

MQSeries®



Application Programming Guide

Note!

Before using this information and the product it supports, be sure to read the general information under “Appendix G. Notices” on page 549.

Thirteenth edition (November 2000)

This edition applies to the following products:

- 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 for Compaq Tru64 UNIX, V5.1
- MQSeries for HP-UX, V5.1
- | • MQSeries for OS/390[®], V5.2
- MQSeries for OS/2[®] Warp, V5.1
- MQSeries for SINIX and DC/OSx, V2.2
- MQSeries for Sun Solaris, V5.1
- | • MQSeries for Sun Solaris, Intel[®] Platform Edition, V5.1
- MQSeries for Tandem NonStop Kernel, V2.2.0.1
- MQSeries for VSE/ESA[™], V2.1
- MQSeries for Windows[®], V2.0
- MQSeries for Windows, V2.1
- MQSeries for Windows NT[®], V5.1

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

© Copyright International Business Machines Corporation 1993, 2000. All rights reserved.

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

Contents

Figures xi

Tables xiii

About this book xv

Who this book is for xv

What you need to know to understand this book. . . xv

How to use this book. xvi

 Appearance of text in this book xvi

 Terms used in this book xvi

Summary of changes xix

Changes for this edition (SC33-0807-12) xix

Changes for the twelfth edition (SC33-0807-11) . . . xix

Changes for the eleventh edition (SC33-0807-10) . . . xix

Changes for the tenth edition (SC33-0807-09) . . . xix

Part 1. Designing applications that use MQSeries 1

Chapter 1. Introduction to message queuing 3

What is message queuing?. 3

What is a message? 4

 Message descriptor 4

 Message channel agent 4

What is a message queue?. 4

What is a queue manager?. 5

What is a cluster?. 5

What is a shared queue, a queue-sharing group, and

intra-group queuing? 6

What is an MQSeries client? 6

Main features of message queuing 6

 MQSeries clients and servers 9

Benefits of message queuing to the application

designer and developer. 9

What can you do with MQSeries products? 9

 MQSeries for OS/390 9

 MQSeries for non-OS/390 platforms 10

Chapter 2. Overview of application design 13

Planning your design 13

Using MQSeries objects 14

Designing your messages. 15

MQSeries techniques 16

 Waiting for messages 16

 Correlating replies 16

 Setting and using context information 16

 Starting MQSeries programs automatically 17

 Generating MQSeries reports 17

 Clusters and message affinities 17

Application programming 17

 Call interface 18

 Design for performance - hints and tips 18

 Programming platforms 18

 Applications for more than one platform 20

Testing MQSeries applications 20

Chapter 3. MQSeries messages 23

Message descriptor 24

Types of message 24

 Datagrams. 24

 Request messages 25

 Reply messages 25

 Report messages. 25

 Reports and segmented messages 27

Format of message control information and message

data 30

 Format of message control information 30

 Format of message data 30

 Application data conversion. 31

Message priorities 32

Message groups 32

Message persistence 34

Selecting messages from queues 34

Messages that fail to be delivered 35

Messages that are backed out 35

Reply-to queue and queue manager 36

Message context 36

 Identity context 37

 Origin context 37

Chapter 4. MQSeries objects 39

Queue managers 39

 Attributes of queue managers 39

 Queue managers and workload management 40

Queue-sharing groups 40

Queues 41

 Types of queue 41

 Attributes of queues 43

 Remote queues 44

 Alias queues 44

 Model queues 45

 Dynamic queues. 45

 Transmission queues 47

 Initiation queues. 47

 Dead-letter (undelivered-message) queues 48

 System command queues. 48

 System default queues. 48

Namelist 48

Process definitions 48

Channels 49

Storage classes 49

Rules for naming MQSeries objects 49

 Queue names. 50

 Process definition and namelist names 50

 Channel names 51

Reserved object names	51
Chapter 5. Handling program errors	53
Locally determined errors	53
Failure of an MQI call	53
System interruptions	53
Messages containing incorrect data	55
Using report messages for problem determination	55
Creating report messages	55
Remotely determined errors	56
Problems delivering a message	56
Using the dead-letter (undelivered-message) queue	57

Part 2. Writing an MQSeries application 61

Chapter 6. Introducing the Message Queue Interface	65
What is in the MQI?	65
Calls	66
Syncpoint calls	67
Data conversion	68
Structures	68
Elementary data types	68
MQSeries data definitions	69
MQSeries stub programs and library files	69
Parameters common to all the calls	75
Using connection and object handles	75
Understanding return codes	75
Specifying buffers	76
Programming language considerations	76
Coding in C	77
Coding in COBOL	79
Coding in System/390® assembler language	80
Specifying the form of a structure	81
Coding in RPG	82
Coding in PL/I	83
Coding in TAL	83
OS/390 batch considerations	84
UNIX signal handling on MQSeries Version 5 products	84
Unthreaded applications	86
Threaded applications	86
Fastpath (trusted) applications	87
MQI function calls within signal handlers	87
Signals during MQI calls	87
User exits and installable services	88

Chapter 7. Connecting and disconnecting a queue manager.	89
Connecting to a queue manager using the MQCONN call	90
Scope of MQCONN	91
Connecting to a queue manager using the MQCONNX call	92
Restrictions	93
Environment variable	94

Disconnecting programs from a queue manager using MQDISC	95
Authority checking	96

Chapter 8. Opening and closing objects 97	
Opening objects using the MQOPEN call	98
Scope of an object handle	98
Identifying objects (the MQOD structure)	99
Name resolution	99
Using the options of the MQOPEN call	102
Creating dynamic queues	104
Opening remote queues	105
Closing objects using the MQCLOSE call	105

Chapter 9. Putting messages on a queue.	107
Putting messages on a local queue using the MQPUT call	107
Specifying handles	108
Defining messages using the MQMD structure	108
Specifying options using the MQPMO structure	108
The data in your message	111
Putting messages on a remote queue	112
Controlling context information	112
Passing identity context	112
Passing all context	113
Setting identity context	113
Setting all context	113
Putting one message on a queue using the MQPUT1 call	113
Distribution lists	115
Opening distribution lists	116
Putting messages to a distribution list	118
Some cases where the put calls fail	120

Chapter 10. Getting messages from a queue.	121
Getting messages from a queue using the MQGET call	121
Specifying connection handles	122
Describing messages using the MQMD structure and the MQGET call	122
Specifying MQGET options using the MQGMO structure	122
Specifying the size of the buffer area	125
The order in which messages are retrieved from a queue	126
Priority	126
Logical and physical ordering	126
Getting a particular message	133
Type of index	134
Handling messages greater than 4 MB long	135
Increasing the maximum message length	135
Message segmentation	136
Reference messages	139
Waiting for messages	141
Signaling	142
To set a signal	142
When the message arrives	143
Skipping backout	144

Application data conversion	147
Conversion of EBCDIC newline characters	148
Browsing messages on a queue	149
The browse cursor.	149
Browsing messages when message length unknown.	150
Removing a message you have browsed	151
Browsing messages in logical order	151
Browsing messages in groups	152
Some cases where the MQGET call fails	154

Chapter 11. Writing data-conversion exits 155

Invoking the data-conversion exit	156
Data conversion on OS/390	156
Writing a data-conversion exit program	157
Skeleton source file	157
Convert characters call	158
Utility for creating conversion-exit code	158
Valid syntax.	159
Writing a data-conversion exit program for MQSeries for AS/400.	161
Writing a data-conversion exit for MQSeries for OS/2 Warp	162
Writing a data-conversion exit program for MQSeries for OS/390.	164
Writing a data-conversion exit for MQSeries for Tandem NonStop Kernel	165
Reusing data-conversion exit programs.	165
Writing a data-conversion exit for MQSeries on UNIX systems and Compaq (DIGITAL) OpenVMS	166
UNIX environment	167
Compiling data-conversion exits on Digital OpenVMS	168
Compiling data-conversion exits on UNIX.	168
Writing a data-conversion exit for MQSeries for Windows NT	171

Chapter 12. Inquiring about and setting object attributes 173

Inquiring about the attributes of an object	174
Some cases where the MQINQ call fails	175
Setting queue attributes	175

Chapter 13. Committing and backing out units of work 177

Syncpoint considerations in MQSeries applications	178
Syncpoints in MQSeries for OS/390 applications	179
Syncpoints in CICS Transaction Server for OS/390 and CICS for MVS/ESA applications.	179
Syncpoints in IMS applications	180
Syncpoints in OS/390 batch applications	180
Syncpoints in CICS for AS/400 applications	182
Syncpoints in MQSeries for OS/2 Warp, MQSeries for Windows NT, MQSeries for AS/400, and MQSeries on UNIX systems	182
Local units of work	183
Global units of work	183
Interfaces to external syncpoint managers	185

Interfaces to the AS/400 external syncpoint manager	187
Syncpoints in MQSeries for Tandem NonStop Kernel applications	188
General XA support	190

Chapter 14. Starting MQSeries applications using triggers 191

What is triggering?	191
Prerequisites for triggering	196
Conditions for a trigger event	198
Controlling trigger events	202
Example of the use of trigger type EVERY.	203
Example of the use of trigger type FIRST	203
Example of the use of trigger type DEPTH	203
Special case of trigger type FIRST	203
Designing an application that uses triggered queues	204
Trigger messages and units of work	204
Getting messages from a triggered queue	205
Trigger monitors	206
MQSeries for OS/390 trigger monitors	206
MQSeries for OS/2 Warp, Digital OpenVMS, Tandem NSK, UNIX systems, AS/400, and Windows NT trigger monitors.	206
MQSeries for AS/400 trigger monitors	208
Properties of trigger messages	209
Persistence and priority of trigger messages	209
Queue manager restart and trigger messages	209
Trigger messages and changes to object attributes.	209
Format of trigger messages.	209
When triggering does not work	210
How CKTI detects errors	211
How CSQQTRMN detects errors	211
How RUNMQTRM detects errors	211

Chapter 15. Using and writing applications on MQSeries for OS/390 213

Environment-dependent MQSeries for OS/390 functions	213
Program debugging facilities	214
Syncpoint support.	214
Recovery support	215
The MQSeries for OS/390 interface with the application environment.	215
The batch adapter	215
RRS batch adapter.	215
The CICS adapter	216
The IMS adapter	219
Writing OS/390 OpenEdition applications.	220
The API-crossing exit for OS/390.	221
Using the API-crossing exit.	221
Writing your own exit program	222
The sample API-crossing exit program, CSQCAPX	224
Preparing and using the API-crossing exit	224
MQSeries Workflow	225
Application programming with shared queues	226
Serializing your applications	226

Applications that are not suitable for use with shared queues	227
Deciding whether to share non-application queues	228
Migrating your existing applications to use shared queues	228

Chapter 16. Using and writing MQSeries-CICS bridge applications for OS/390. 231

Using CICS DPL programs with the bridge	232
CICS DPL bridge message structure	232
Application programming for the CICS DPL bridge	233
Using CICS transactions with the bridge	234
Using CICS bridge vectors	235
CICS 3270 bridge message structure	235
Application programming for the CICS 3270 bridge	237
Transactions with start data	249
Programming for the distributed environment	249
CICS transactions in the distributed environment	249
CICS DPL programs in the distributed environment	250
From 3270 legacy to 3270 bridge - an example	250
Exact emulation - no optimization	251
Improved emulation, with optimization	252
Setting fields in the MQMD and MQCIH structures	253
Setting the MQMD fields	253
Setting the MQCIH fields	253
Setting the open options and put message options for the bridge request queue	256
Managing MsgId and CorrelId in a unit of work	256
Error handling by the CICS bridge	258
Debugging CICS bridge applications	260
Application data structure terminology	262

Chapter 17. IMS and IMS Bridge applications on MQSeries for OS/390 . 265

Writing IMS applications using MQSeries	265
Syncpoints in IMS applications	265
MQI calls in IMS applications	265
Writing MQSeries-IMS bridge applications	269
How the MQSeries-IMS bridge deals with messages	269
Writing your program	275

Chapter 18. Object-oriented programming with MQSeries. 279

What is in the MQSeries Object Model?	279
Classes	279
Object references	280
Return codes	280
Programming language considerations	280
Coding in C++	280
Coding in Java	281
Coding in LotusScript	281
Coding in ActiveX.	281

Part 3. Building an MQSeries application 283

Chapter 19. Building your application on AIX 285

Preparing C programs	285
Linking libraries	285
Preparing COBOL programs	286
Preparing COBOL programs using IBM COBOL SET for AIX	287
Preparing COBOL programs using Micro Focus COBOL	287
Preparing PL/I programs	287
Preparing CICS programs	287
CICS on Open Systems support	288

Chapter 20. Building your application on AS/400 291

Preparing C programs	291
Preparing COBOL programs	292
Preparing CICS programs	293
Preparing RPG programs	293
SQL programming considerations	294
AS/400 programming considerations	294
QM/QM activation group	294

Chapter 21. Building your application on AT&T GIS UNIX 295

Preparing C programs	295
C compiler flags	295
Linking libraries	295

Chapter 22. Building your application on Digital OpenVMS 297

Preparing C programs	297
C compiler version	297
C compiler flags	297
Linking libraries	297
Preparing COBOL programs	297
COBOL compiler flags	298
Linking libraries	298

Chapter 23. Building your application on Compaq Tru64 UNIX 299

Building applications in C on Compaq Tru64 UNIX Version 4.0	299
C language include files	299
Preparing C programs	299
Linking libraries	300
Building applications in C on Compaq Tru64 UNIX Version 5.0	300
C language include files	300
Preparing C programs	301
Linking libraries	301
Building applications in C++ on Compaq Tru64 UNIX Version 4.0	301
C++ language include files	301
Preparing C++ programs	302

	Linking libraries	302
	Building applications in C++ on Compaq Tru64 UNIX Version 5.0	302
	C++ language include files	302
	Preparing C++ programs	303
	Linking libraries	303
	Building applications in COBOL	303
	Preparing COBOL programs	303
	Linking libraries	304
	Building applications in Java	304
	Preparing Java programs	304

Chapter 24. Building your application on HP-UX 305

Preparing C programs	305
Preparing C programs on HP-UX V10.20	305
Preparing C programs on HP-UX V11.00	306
Linking libraries	306
Preparing COBOL programs	307
Programs to run in the MQSeries client environment.	307
Preparing CICS programs	308
CICS on Open Systems support	308

Chapter 25. Building your application on OS/390 311

Preparing your program to run	311
Building OS/390 batch applications	312
Building CICS applications	313
Building IMS (BMP or MPP) applications	314
Dynamically calling the MQSeries stub	315
Debugging your programs	319
Debugging CICS programs	320
Debugging TSO programs	322

Chapter 26. Building your application on OS/2 Warp 323

Preparing C programs	323
Preparing CICS and Transaction Server programs.	324
Preparing COBOL programs	325
Preparing Transaction Server programs.	326
Preparing PL/I programs	326

Chapter 27. Building your application on SINIX or DC/OSx 327

Preparing C programs	327
C compiler flags	327
Preparing COBOL programs	328
Compiling COBOL programs	328
Preparing CICS programs	329
CICS on Open Systems support	329
CICS sample transaction.	329
Linking libraries	330

Chapter 28. Building your application on Sun Solaris 331

Preparing C programs	331
Linking libraries	332

Preparing COBOL programs	332
Preparing CICS programs	333
CICS on Open Systems support	333

Chapter 29. Building your application on Tandem NSK 335

Unit of work (transaction) management	335
General design considerations	335
MQGMO_BROWSE_* with MQGMO_LOCK	335
Triggered applications	335
Compiling and binding applications.	336
Running applications.	337

Chapter 30. Building your application on VSE/ESA 339

Linking library	339
Using the batch interface	339
Preparing C programs	339
Preparing COBOL programs	339
Preparing PL/I programs	339

Chapter 31. Building your application on Windows 341

Linking libraries	341
Preparing Visual Basic programs	341

Chapter 32. Building your application on Windows NT 343

Preparing C programs	343
Preparing CICS and Transaction Server programs.	344
Preparing COBOL programs	345
Preparing CICS and Transaction Server programs.	346
Preparing PL/I programs	347
Preparing Visual Basic programs	347

Chapter 33. Using lightweight directory access protocol services with MQSeries for Windows NT 349

What is a directory service?	349
What is LDAP?.	349
Using LDAP with MQSeries	350
LDAP sample program	351
Building the sample program	351
Configuring the directory	351
Configuring the IBM eNetwork LDAP server	351
Configuring the Netscape directory server.	353
Running the sample program	353
Program design	354

Part 4. Sample MQSeries programs 357

Chapter 34. Sample programs (all platforms except OS/390) 359

Features demonstrated in the sample programs	360
--	-----

Samples for Compaq (DIGITAL) OpenVMS and UNIX systems	360
Samples for OS/2 Warp and Windows NT PL/I samples for AIX, OS/2 Warp, and Windows NT	363
Visual Basic samples for Windows NT	364
Samples for AS/400	364
Samples for Tandem NonStop Kernel	366
Samples for VSE/ESA	366
Preparing and running the sample programs	367
AS/400	367
UNIX systems	367
Digital OpenVMS	367
OS/2 and Windows NT	368
Tandem NSK	369
Windows	371
Running the sample programs	371
The Put sample programs	373
Running the amqsput and amqsputc samples	373
Running the amqsputw sample	373
Running the amq0put sample	374
Running the AMQSPUT4 C sample	374
Running the AMQ0PUT4 COBOL sample	375
Design of the Put sample program	375
The Distribution List sample program	375
Running the Distribution List sample, amqsptl0	375
Design of the Distribution List sample	375
The Browse sample programs	376
OS/2, UNIX systems, Digital OpenVMS, and Windows NT	376
AS/400	377
Design of the Browse sample program	377
The Browser sample program	378
The Get sample programs	378
Running the amqsget and amqsgetc samples	378
Running the amqsgetw sample	379
Running the amq0get sample	379
Running the AMQSGET4 and the AMQ0GET4 samples	380
Design of the Get sample program	380
The Reference Message sample programs	380
Notes for AS/400 users	381
Running the Reference Message samples	382
Design of the Put Reference Message sample (amqsprma.c, AMQSPRM4)	386
Design of the Reference Message Exit sample (amqsxrma.c, AMQSXRM4)	386
Design of the Get Reference Message sample (amqsgrma.c, AMQSGRM4)	388
The Request sample programs	388
Running the amqsreq0.c, amqsreq, and amqsreqc samples	389
Running the amq0req0.cbl sample	389
Running the AMQSREQ4 sample	389
Running the AMQ0REQ4 sample	389
Running the Request sample using triggering	390
Design of the Request sample program	393
The Inquire sample programs	394
Design of the Inquire sample program	395
The Set sample programs	395
Design of the Set sample program	396

The Echo sample programs	397
Design of the Echo sample programs	397
The Data-Conversion sample program	398
Design of the data-conversion sample	398
The Triggering sample programs	398
Running the amqstrg0.c, amqstrg, and amqstrgc samples	398
Running the AMQSTRG4 sample	398
Design of the triggering sample	399
Running the AMQSERV4 sample	399
Design of the trigger server	400
Ending the triggering sample programs on AS/400	400
Running the samples using remote queues	400
Database coordination samples	400
Creating the databases and tables	402
Precompiling, compiling, and linking the samples	403
Running the samples	405
The CICS transaction sample	407
TUXEDO samples	407
Building the server environment	407
Server sample program for TUXEDO	416
Put sample program for TUXEDO	417
Get sample for TUXEDO	417
Encina sample program	418
Building the AMQSXAE0.C sample	418
Dead-letter queue handler sample	419
The Connect sample program	419
Running the amqscnxc sample	419

Chapter 35. Sample programs for MQSeries for OS/390 421

Features demonstrated in the sample applications	421
Put samples	421
Get samples	422
Browse sample	422
Print Message sample	422
Queue Attributes sample	423
Mail Manager sample	423
Credit Check sample	423
The Message Handler sample	424
Distributed queuing exit samples	425
Data-conversion exit samples	425
Preparing and running sample applications for the batch environment	425
Names of the sample batch applications	426
Preparing sample applications for the TSO environment	427
Names of the sample TSO applications	427
Preparing the sample applications for the CICS environment	428
QLOP abend	429
Names of the sample CICS applications	429
Preparing the sample application for the IMS environment	431
Names of the sample IMS application	432
The Put samples	432
Design of the Put sample	432
The Put samples for the batch environment	433
The Put samples for the CICS environment	434

The Get samples	435
Design of the Get sample	435
The Get samples for the CICS environment	436
The Browse sample	437
Design of the Browse sample	438
Language-dependent design considerations	439
The Print Message sample	439
Design of the sample	441
The Queue Attributes sample	443
Design of the sample	443
The Mail Manager sample	444
Preparing the sample	444
Running the sample	445
Design of the sample	447
The Credit Check sample	451
Preparing and running the Credit Check sample	452
Design of the sample	453
Design considerations	460
The Credit Check sample with multiple queue managers	462
The IMS extension to the Credit Check sample	462
The Message Handler sample	464
Preparing and running the sample	464
Using the sample	464
Design of the sample	466

Part 5. Appendixes 469

Appendix A. Language compilers and assemblers 471

Appendix B. C language examples 475

Connecting to a queue manager	476
Disconnecting from a queue manager	476
Creating a dynamic queue	477
Opening an existing queue	478
Closing a queue	479
Putting a message using MQPUT.	479
Putting a message using MQPUT1	480
Getting a message	482
Getting a message using the wait option	483
Getting a message using signaling	484
Inquiring about the attributes of an object	486
Setting the attributes of a queue	487

Appendix C. COBOL examples 491

Connecting to a queue manager	492
Disconnecting from a queue manager	492
Creating a dynamic queue	493
Opening an existing queue	494
Closing a queue	496
Putting a message using MQPUT.	496
Putting a message using MQPUT1	498
Getting a message	499
Getting a message using the wait option	501
Getting a message using signaling	502
Inquiring about the attributes of an object	505
Setting the attributes of a queue	506

Appendix D. System/390 assembler-language examples 509

Connecting to a queue manager	510
Disconnecting from a queue manager	511
Creating a dynamic queue	511
Opening an existing queue	513
Closing a queue	514
Putting a message using MQPUT.	514
Putting a message using MQPUT1	516
Getting a message	517
Getting a message using the wait option	519
Getting a message using signaling	520
Inquiring about and setting the attributes of a queue	522

Appendix E. PL/I examples 525

Connecting to a queue manager	526
Disconnecting from a queue manager	526
Creating a dynamic queue	527
Opening an existing queue	528
Closing a queue	529
Putting a message using MQPUT.	529
Putting a message using MQPUT1	531
Getting a message	532
Getting a message using the wait option	533
Getting a message using signaling	534
Inquiring about the attributes of an object	537
Setting the attributes of a queue	538

Appendix F. MQSeries data definition files 541

C language include files	542
Visual Basic module files	542
COBOL copy files	543
System/390 assembler-language macros	546
PL/I include files	547

Appendix G. Notices 549

Programming interface information	550
Trademarks	551

Glossary of terms and abbreviations 553

Bibliography. 565

MQSeries cross-platform publications	565
MQSeries platform-specific publications	565
Softcopy books	566
HTML format	566
Portable Document Format (PDF)	566
BookManager® format	567
PostScript format	567
Windows Help format	567
MQSeries information available on the Internet	567
Related publications	567
CICS	567
IMS	567
MVS/ESA	568
Design	568
C	568

C++	568
COBOL	568
LDAP	568
RPG	568

Index	569
Sending your comments to IBM	581

Figures

1.	Message queuing compared with traditional communication.	7	27.	Dynamic linking using assembler language in the CICS environment	318
2.	Representation of a message	23	28.	Dynamic linking using assembler language in the IMS environment	318
3.	Group of logical messages	33	29.	Dynamic linking using C language in the batch environment	318
4.	Segmented messages	34	30.	Dynamic linking using C language in the CICS environment	318
5.	How distribution lists work.	116	31.	Dynamic linking using C language in the IMS environment	319
6.	Opening a distribution list in C	118	32.	Dynamic linking using PL/I in the batch environment	319
7.	Opening a distribution list in COBOL	118	33.	Dynamic linking using PL/I in the IMS environment	319
8.	Putting a message to a distribution list in C	119	34.	Running the reference message samples	382
9.	Putting a message to a distribution list in COBOL	120	35.	Request and Inquire samples using triggering	391
10.	Logical order on a queue	127	36.	Sample Client/Server (Echo) program flowchart	394
11.	Physical order on a queue	128	37.	The database coordination samples	401
12.	Skipping backout using MQGMO_MARK_SKIP_BACKOUT	146	38.	Example of ubbstxcx.cfg file for UNIX systems	412
13.	Sample JCL used to invoke the CSQUCVX utility	159	39.	Example of ubbstxcn.cfg file for Windows NT	414
14.	Flow of application and trigger messages	194	40.	Sample TUXEDO makefile for MQSeries for Windows NT	415
15.	Relationship of queues within triggering	195	41.	How TUXEDO samples work together	417
16.	Setting of key fields for many CICS user programs in a unit of work	257	42.	Example of a report from the Print Message sample application.	440
17.	Setting of key fields: MQSeries - conversational 3270 transaction	258	43.	Programs and panels for the TSO versions of the Mail Manager	447
18.	User program abends (only program in the unit of work)	259	44.	Programs and panels for the CICS version of the Mail Manager	448
19.	Fragments of JCL to link-edit the object module in the batch environment, using single-phase commit	312	45.	Example of a panel showing a list of waiting messages	449
20.	Fragments of JCL to link-edit the object module in the batch environment, using two-phase commit	313	46.	Example of a panel showing the contents of a message	450
21.	Fragments of JCL to link-edit the object module in the CICS environment	314	47.	Immediate Inquiry panel for the Credit Check sample application.	452
22.	Fragments of JCL to link-edit the object module in the IMS environment	315	48.	Programs and queues for the Credit Check sample application (COBOL programs only)	454
23.	Dynamic linking using COBOL in the batch environment	316	49.	Initial screen for Message Handler sample	464
24.	Dynamic linking using COBOL in the CICS environment	317	50.	Message list screen for Message Handler sample.	465
25.	Dynamic linking using COBOL in the IMS environment	317	51.	Chosen message is displayed	466
26.	Dynamic linking using assembler language in the batch environment	317			

Tables

1.	Visual Basic equivalents of the C elementary data types	69	29.	MQSeries for AS/400 sample programs demonstrating use of the MQI	364
2.	Environment variable	94	30.	MQSeries for Tandem NonStop Kernel C and COBOL sample programs demonstrating use of the MQI	366
3.	Resolving queue names when using MQOPEN.	100	31.	MQSeries for Tandem NonStop Kernel TAL sample programs demonstrating use of the MQI	366
4.	How queue attributes and options of the MQOPEN call affect access to queues	103	32.	MQSeries for VSE/ESA COBOL sample programs demonstrating use of the MQI	366
5.	Using message and correlation identifiers	133	33.	Where to find the samples for MQSeries on UNIX systems	367
6.	Using the group identifier	133	34.	Where to find the samples for MQSeries for Compaq (DIGITAL) OpenVMS.	367
7.	Skeleton source files	157	35.	Where to find the samples for MQSeries for OS/2 Warp and MQSeries for Windows NT	368
8.	Linking MQSeries for OS/2 Warp with CICS Version 3 applications.	186	36.	Source for the distributed queuing exit samples	425
9.	Essential Code for CICS applications	190	37.	Source for the data conversion exit samples (Assembler language only)	425
10.	OS/390 environmental features	214	38.	Source and JCL for the Put and Get samples	426
11.	When to use a shared-initiation queue	229	39.	Source and JCL for the Browse sample	426
12.	Mapping MQSeries messages to IMS transaction types	270	40.	Source for the Print Message sample (C language only)	426
13.	Essential Code for CICS applications (AIX)	287	41.	Source and JCL for the Mail Manager (TSO) sample.	427
14.	Example of CRTPGM in the nonthreaded environment	291	42.	Source for the Message Handler sample	428
15.	Example of CRTPGM in the threaded environment	292	43.	Source and JCL for the Put and Get samples	429
16.	C include files for MQSeries (Compaq Tru64 UNIX Version 4.0)	299	44.	Source for the Queue Attributes sample	430
17.	C include files for MQSeries (Compaq Tru64 UNIX Version 5.0)	300	45.	Source and JCL for the Mail Manager (CICS) sample (COBOL only).	430
18.	C++ include files for MQSeries (Compaq Tru64 UNIX Version 4.0)	301	46.	Source and JCL for the Credit Check CICS sample.	430
19.	C++ include files for MQSeries (Compaq Tru64 UNIX Version 5.0)	302	47.	Source and JCL for the Credit Check IMS sample (C only).	432
20.	Essential Code for CICS applications (HP-UX)	308	48.	Language compilers and assemblers	471
21.	Call names for dynamic linking	315	49.	C include files for MQSeries	542
22.	CICS adapter trace entries	320	50.	Visual Basic module files for MQSeries for Windows V2.0	542
23.	Essential Code for CICS applications (SINIX)	329	51.	Visual Basic module files for MQSeries for Windows V2.1	543
24.	Essential Code for CICS applications (Sun Solaris).	333	52.	Visual Basic module files for MQSeries for Windows NT, V5.1.	543
25.	MQSeries on UNIX and Digital OpenVMS sample programs demonstrating use of the MQI	360	53.	COBOL copy files	544
26.	MQSeries for OS/2 Warp and Windows NT sample programs demonstrating use of the MQI	362	54.	System/390 assembler-language macros	546
27.	MQSeries for AIX, OS/2 Warp, and Windows NT sample programs demonstrating use of the MQI	364	55.	PL/I include files	547
28.	MQSeries for Windows NT sample programs demonstrating use of the MQI	364			

About this book

This book introduces the concepts of *messages* and *queues*, and shows you in detail how to design and write applications that use the services that MQSeries provides.

The IBM® MQSeries Level 2 products comprise:

- MQSeries for AIX
- MQSeries for AS/400 (formerly known as MQSeries for OS/400)
- MQSeries for AT&T GIS UNIX¹
- MQSeries for Compaq (DIGITAL) OpenVMS
- MQSeries for Compaq Tru64 UNIX
- MQSeries for HP-UX
- MQSeries for OS/390 (formerly known as MQSeries for MVS/ESA)
- MQSeries for OS/2 Warp
- MQSeries for SINIX and DC/OSx
- MQSeries for Sun Solaris
- MQSeries for Tandem NonStop Kernel
- MQSeries for VSE/ESA
- MQSeries for Windows
- MQSeries for Windows NT

They are referred to in this book collectively as MQSeries. They provide application programming services that enable you to write applications in which the constituent programs communicate with each other using *message queues*.

For a full description of the MQSeries programming interface, see the *MQSeries Application Programming Reference* manual for your platform. The manuals are:

- *MQSeries Application Programming Reference* manual, SC33-1673
- *MQSeries for AS/400 Application Programming Reference (ILE RPG)*, SC 34-5559

For information on the use of C++, see the *MQSeries Using C++* book.

IBM ships sample programs with MQSeries, which are explained in “Part 4. Sample MQSeries programs” on page 357. Also, appendixes “Appendix B. C language examples” on page 475 through “Appendix E. PL/I examples” on page 525 give programming examples in the various languages. You may find it useful to refer to these.

Who this book is for

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

What you need to know to understand this book

To write message queuing applications using MQSeries, you need to know how to write programs in at least one of the programming languages that MQSeries supports. “Appendix A. Language compilers and assemblers” on page 471 contains details of supported compilers and assemblers listed by MQSeries platform.

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

About this book

If the applications you are writing will run within a CICS® or IMS system, you must also be familiar with CICS or IMS, and their application programming interfaces.

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

How to use this book

This book contains guidance information to help you design an application, and procedural information to help you to write an application.

The book is divided into five parts:

“Part 1. Designing applications that use MQSeries” on page 1

Introduces the message queuing style of application design, describes MQSeries messages and queues, and shows how to design a message queuing application.

“Part 2. Writing an MQSeries application” on page 61

Describes how to use the IBM Message Queue Interface (MQI) to write the programs that comprise a message queuing application. The chapters guide you through the coding of each MQI call, showing you what information to supply as input and what returns to expect. These chapters first describe simple uses of the MQI calls, then go on to describe how to use all the features of each call.

Read “Part 1. Designing applications that use MQSeries” on page 1 to understand the concepts involved when designing MQSeries applications. The second part is self-contained: use an individual chapter when you are performing the task described in it.

“Part 3. Building an MQSeries application” on page 283

Explains how to build your MQSeries application on each platform.

“Part 4. Sample MQSeries programs” on page 357

Lists and explains how the sample programs work, for all platforms.

The appendixes

Contain examples of how to use the MQI calls in each of the programming languages supported by MQSeries.

There is a glossary and a bibliography at the back of the book.

Appearance of text in this book

This book uses the following type style:

CompCode

Example of the name of a parameter of a call, or the attribute of an object

Terms used in this book

In the body of this book, the following shortened names are used for these products and a qualifier:

CICS CICS for AS/400, CICS for MVS/ESA™, CICS for VSE/ESA, CICS Transaction Server for OS/2, CICS Transaction Server for OS/390, TXSeries™ for AIX, TXSeries for HP-UX, TXSeries for Sun Solaris, and TXSeries for Windows NT products.

IMS The IMS/ESA product.

MQSeries

MQSeries for AIX, MQSeries for AS/400, MQSeries for AT&T GIS UNIX, MQSeries for Compaq (DIGITAL) OpenVMS, MQSeries for Compaq Tru64 UNIX, MQSeries for HP-UX, MQSeries for OS/2 Warp, MQSeries for OS/390, MQSeries for SINIX and DC/OSx, MQSeries for Sun Solaris, MQSeries for Tandem NonStop Kernel, MQSeries for VSE/ESA, MQSeries for Windows, and MQSeries for Windows NT.

MQSeries on UNIX systems

MQSeries for AIX, MQSeries for AT&T GIS UNIX, MQSeries for Compaq Tru64 UNIX, MQSeries for HP-UX, MQSeries for SINIX and DC/OSx, and MQSeries for Sun Solaris (SPARC and Intel Platform Editions).

MQSeries for Sun Solaris

MQSeries for Sun Solaris (SPARC and Intel Platform Editions).

MQSeries Version 5 products

Version 5 of MQSeries for AIX, AS/400, Compaq Tru64 UNIX, HP-UX, OS/2 Warp, OS/390, Sun Solaris, and Windows NT.

OS/390

The OS/390 System Product.

thlqual

The high-level qualifier of the installation library on OS/390.

About this book

Summary of changes

This section describes changes in this edition of *MQSeries Application Programming Guide*. Changes since the previous edition of the book are marked by vertical lines to the left of the changes.

Changes for this edition (SC33-0807-12)

The main changes to this edition are the addition of:

- MQSeries for OS/390, V5.2
- MQSeries for Compaq Tru64 UNIX, V5.1
- MQSeries for Sun Solaris, Intel Platform Edition, V5.1

The changes to the book include:

- Addition of support for large messages on MQSeries for OS/390.
- Addition of support for shared queues on MQSeries for OS/390.
- Addition of support for MQCONN on MQSeries for OS/390.
- Information about the MQSeries-CICS bridge has been re-written, and moved to its own chapter: “Chapter 16. Using and writing MQSeries-CICS bridge applications for OS/390” on page 231.
- Information about IMS has been moved to its own chapter: “Chapter 17. IMS and IMS Bridge applications on MQSeries for OS/390” on page 265.
- The Application Messaging Interface (AMI) is suggested as a simpler and higher-level programming interface than the Message Queue Interface (MQI).

Changes for the twelfth edition (SC33-0807-11)

The twelfth edition was not published.

Changes for the eleventh edition (SC33-0807-10)

The main change to this edition of the Application Programming Guide is the enhancement of MQSeries for AS/400 bringing it to the same level of function as the other MQSeries Version 5 Release 1 products.

Also included in this edition is a section regarding SQL programming considerations on MQSeries for AS/400. There is a new section about building CICS applications on MQSeries for AS/400. See “Chapter 20. Building your application on AS/400” on page 291 for these new sections.

Table 29 on page 364 has been expanded to include information on RPG samples for MQSeries for AS/400.

Changes for the tenth edition (SC33-0807-09)

In this edition, the book has been updated to reflect the new function in the following new versions of the MQSeries products:

- V5.1 of MQSeries for AIX, Compaq Tru64 UNIX, HP-UX, OS/2 Warp, Sun Solaris, and Windows NT

Changes

- V2.1 of MQSeries for OS/390, MQSeries for VSE/ESA, and MQSeries for Windows
- V4R2M1 of MQSeries for AS/400

The changes to the book include:

- Queue manager clusters (applicable to the V5.1 products, and MQSeries for OS/390 only). There are many references throughout the book to this new function.
- Recoverable resource services (applicable to MQSeries for OS/390 in the Batch/TSO environment). This provides two-phase syncpoint support. “Chapter 15. Using and writing applications on MQSeries for OS/390” on page 213 contains information about it.
- UNIX signal handling on MQSeries Version 5 products. “UNIX signal handling on MQSeries Version 5 products” on page 84 explains the changes to signal handling in threaded and non-threaded environments in MQSeries for AIX, MQSeries for HP-UX, and MQSeries for Sun Solaris.
- Writing MQSeries-CICS bridge applications (applicable to MQSeries for OS/390 only).
- MQSeries Workflow is how MQSeries supports the OS/390 workload manager (WLM). “MQSeries Workflow” on page 225 provides information.
- Object-oriented programming with MQSeries. A new chapter, “Chapter 18. Object-oriented programming with MQSeries” on page 279, gives an introduction to the MQSeries object model and the base classes. There are further references to language-specific information.
- Using lightweight directory access protocol services (LDAP) with MQSeries for Windows NT. A new chapter, “Chapter 33. Using lightweight directory access protocol services with MQSeries for Windows NT” on page 349, is an introduction to LDAP and contains an example of an MQSeries application using an LDAP directory.
- A new chapter, “Chapter 30. Building your application on VSE/ESA” on page 339 is included.

Part 1. Designing applications that use MQSeries

Chapter 1. Introduction to message queuing . . . 3

What is message queuing?	3
What is a message?	4
Message descriptor	4
Message channel agent	4
What is a message queue?	4
What is a queue manager?	5
What is a cluster?	5
What is a shared queue, a queue-sharing group, and intra-group queuing?	6
What is an MQSeries client?	6
Main features of message queuing	6
MQSeries clients and servers	9
Benefits of message queuing to the application designer and developer.	9
What can you do with MQSeries products?	9
MQSeries for OS/390	9
MQSeries for non-OS/390 platforms	10

Chapter 2. Overview of application design . . . 13

Planning your design	13
Using MQSeries objects	14
Designing your messages.	15
MQSeries techniques	16
Waiting for messages	16
Correlating replies	16
Setting and using context information	16
Starting MQSeries programs automatically	17
Generating MQSeries reports	17
Clusters and message affinities	17
Application programming	17
Call interface	18
Design for performance - hints and tips	18
Programming platforms	18
Applications for more than one platform	20
Testing MQSeries applications	20

Chapter 3. MQSeries messages 23

Message descriptor	24
Types of message	24
Datagrams.	24
Request messages	25
Reply messages	25
Report messages.	25
Types of report message	25
Report message options	26
Reports and segmented messages	27
MQSeries-generated reports	27
Application-generated reports	28
Retrieval of reports	28
Back-level queue managers	29
Format of message control information and message data	30
Format of message control information	30
Format of message data	30
Application data conversion	31

Conversion at the sending queue manager	31
Conversion at the receiving queue manager	31
Coded character sets	31
Message priorities	32
Message groups	32
Message persistence	34
Selecting messages from queues	34
Messages that fail to be delivered	35
Messages that are backed out	35
Reply-to queue and queue manager	36
Message context	36
Identity context	37
Origin context	37

Chapter 4. MQSeries objects 39

Queue managers	39
Attributes of queue managers	39
Queue managers and workload management	40
Queue-sharing groups	40
Queues.	41
Types of queue	41
Types of local queue	42
Attributes of queues	43
Remote queues	44
Alias queues	44
Model queues	45
Dynamic queues.	45
Properties of temporary dynamic queues	45
Properties of permanent dynamic queues	46
Uses of dynamic queues	46
Recommendations for uses of dynamic queues	47
Transmission queues	47
Initiation queues.	47
Dead-letter (undelivered-message) queues	48
System command queues.	48
System default queues.	48
Namelist	48
Process definitions	48
Channels	49
Storage classes	49
Rules for naming MQSeries objects	49
Queue names.	50
Process definition and namelist names	50
Channel names	51
Reserved object names.	51

Chapter 5. Handling program errors 53

Locally determined errors	53
Failure of an MQI call	53
System interruptions	53
Messages containing incorrect data	55
Using report messages for problem determination	55
Creating report messages.	55
Requesting and receiving (MQGET) report messages	56
Remotely determined errors	56

Problems delivering a message	56
Retry message delivery	57
Return message to sender	57
Using the dead-letter (undelivered-message)	
queue	57
Dead-letter queue processing	58

Chapter 1. Introduction to message queuing

The MQSeries products enable programs to communicate with one another across a network of unlike components – processors, operating systems, subsystems and communication protocols – using a consistent application programming interface.

Applications designed and written using this interface are known as *message queuing* applications, as they use the *messaging* and *queuing* style:

Messaging

Programs communicate by sending each other data in messages rather than calling each other directly.

Queuing

Messages are placed on queues in storage, allowing programs to run independently of each other, at different speeds and times, in different locations, and without having a logical connection between them.

This chapter introduces messaging and queuing concepts, under these headings:

- “What is message queuing?”
- “What is a message?” on page 4
- “What is a message queue?” on page 4
- “What is a queue manager?” on page 5
- “What is a cluster?” on page 5
- “What is an MQSeries client?” on page 6
- “Main features of message queuing” on page 6
- “Benefits of message queuing to the application designer and developer” on page 9
- “What can you do with MQSeries products?” on page 9

What is message queuing?

Message queuing has been used in data processing for many years. It is most commonly used today in electronic mail. Without queuing, sending an electronic message over long distances requires every node on the route to be available for forwarding messages, and the addressees to be logged on and conscious of the fact that you are trying to send them a message. In a queuing system, messages are stored at intermediate nodes until the system is ready to forward them. At their final destination they are stored in an electronic mailbox until the addressee is ready to read them.

Even so, many complex business transactions are processed today without queuing. In a large network, the system might be maintaining many thousands of connections in a ready-to-use state. If one part of the system suffers a problem, many parts of the system become unusable.

You can think of message queuing as being electronic mail for programs. In a message queuing environment, each program from the set that makes up an application suite is designed to perform a well-defined, self-contained function in response to a specific request. To communicate with another program, a program must put a message on a predefined queue. The other program retrieves the message from the queue, and processes the requests and information contained in the message. So message queuing is a style of program-to-program communication.

Definition of terms

Queuing is the mechanism by which messages are held until an application is ready to process them. Queuing allows you to:

- Communicate between programs (which may each be running in different environments) without having to write the communication code.
- Select the order in which a program processes messages.
- Balance loads on a system by arranging for more than one program to service a queue when the number of messages exceeds a threshold.
- Increase the availability of your applications by arranging for an alternative system to service the queues if your primary system is unavailable.

What is a message?

In message queuing, a *message* is simply a collection of data sent by one program and intended for another program.

MQSeries defines four types of message:

Datagram	A simple message for which no reply is expected
Request	A message for which a reply is expected
Reply	A reply to a request message
Report	a message that describes an event such as the occurrence of an error

See “Types of message” on page 24 for more information about these messages.

Message descriptor

An MQSeries message consists of control information and application data. The control information is defined in a *message descriptor* structure (MQMD) and contains such things as:

- The type of the message
- An identifier for the message
- The priority for delivery of the message

The structure and content of the application data is determined by the participating programs, not by MQSeries.

Message channel agent

A message channel agent moves messages from one queue manager to another. References are made to them in this book when dealing with report messages and you will need to consider them when designing your application. See the *MQSeries Intercommunication* book for more information.

What is a message queue?

A *message queue*, known simply as a queue, is a named destination to which messages can be sent. Messages accumulate on queues until they are retrieved by programs that service those queues.

Queues reside in, and are managed by, a queue manager (see “What is a queue manager?” on page 5). The physical nature of a queue depends on the operating system on which the queue manager is running. A queue can either be a volatile buffer area in the memory of a computer, or a data set on a permanent storage device (such as a disk). The physical management of queues is the responsibility of the queue manager and is not made apparent to the participating application programs.

Programs access queues only through the external services of the queue manager. They can open a queue, put messages on it, get messages from it, and close the queue. They can also set, and inquire about, the attributes of queues.

What is a queue manager?

A *queue manager* is a system program that provides queuing services to applications. It provides an application programming interface so that programs can put messages on, and get messages from, queues. A queue manager provides additional functions so that administrators can create new queues, alter the properties of existing queues, and control the operation of the queue manager.

For MQSeries message queuing services to be available on a system, there must be a queue manager running:

- On OS/400, OS/390, OS/2, Windows NT, Digital OpenVMS, and UNIX systems, you can have more than one queue manager running on a single system (for example, to separate a test system from a “live” system). To an application, each queue manager is identified by a *connection handle (Hconn)*.
- On the VSE/ESA and Windows platforms, you can have only one queue manager running on a single system. *Hconn* is still used, but only to give compatibility with other MQSeries platforms.

Many different applications can make use of the queue manager’s services at the same time and these applications can be entirely unrelated. For a program to use the services of a queue manager, it must establish a connection to that queue manager.

For applications to be able to send messages to applications that are connected to other queue managers, the queue managers must be able to communicate among themselves. MQSeries implements a *store-and-forward* protocol to ensure the safe delivery of messages between such applications.

What is a cluster?

A *cluster* is a network of queue managers that are logically associated in some way. Clustering is available to queue managers on the following MQSeries Version 5 products:

- MQSeries for AIX
- MQSeries for AS/400
- MQSeries for HP-UX
- MQSeries for OS/2 Warp
- MQSeries for OS/390
- MQSeries for Sun Solaris
- MQSeries for Compaq Tru64 UNIX
- MQSeries for Windows NT

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

If you group queue managers in a cluster, the queue managers can make the queues that they host available to every other queue manager in the cluster. Then, assuming you have the necessary network infrastructure in place, any queue

Definition of terms

manager can send a message to any other queue manager in the same cluster without the need for explicit channel definitions, remote queue definitions, or transmission queues.

There are two quite different reasons for using clusters: to reduce system administration and to improve availability and workload balancing.

As soon as you establish even the smallest cluster you will benefit from simplified system administration. Queue managers that are part of a cluster need fewer definitions and so the risk of making an error in your definitions is reduced.

For details of all aspects of clustering, see the *MQSeries Queue Manager Clusters* book.

What is a shared queue, a queue-sharing group, and intra-group queuing?

Shared queues, queue-sharing groups, and intra-group queuing are only available on MQSeries for OS/390.

A *shared queue* is a type of local queue whose messages can be accessed by one or more queue managers that are in a sysplex. (This is not the same as a queue being "shared" by more than one application, via the same queue manager.)

The queue managers that can access the same set of shared queues form a group called a *queue-sharing group* (QSG). They communicate with each other by means of a coupling facility (CF) that stores the shared queues. See the *MQSeries for OS/390 Concepts and Planning Guide* for a full discussion of queue-sharing groups.

Message transfer between queue managers in a queue-sharing group is called *intra-group queuing* (IGQ), and lets you perform fast message transfer without defining channels.

What is an MQSeries client?

An *MQSeries client* is an independently installable component of an MQSeries product. It allows you to run MQSeries applications, by means of a communications protocol, to interact with one or more Message Queue Interface (MQI) servers on other platforms and to connect to their queue managers.

For full details on how to install the MQSeries client component and use the environment, see the *MQSeries Clients* book.

Main features of message queuing

The main features of applications that use message queuing techniques are:

- There are no direct connections between programs.
- Communication between programs can be time-independent.
- Work can be carried out by small, self-contained programs.
- Communication can be driven by events.
- Applications can assign a priority to a message.
- Security.
- Data integrity.
- Recovery support.

No direct connections between programs

Message queuing is a technique for indirect program-to-program communication. It can be used within any application where programs communicate with each other. Communication occurs by one program putting messages on a queue (owned by a queue manager) and another program getting the messages from the queue.

Programs can get messages that were put on a queue by other programs. The other programs can be connected to the same queue manager as the receiving program, or to another queue manager. This other queue manager might be on another system, a different computer system, or even within a different business or enterprise.

There are no physical connections between programs that communicate using message queues. A program sends messages to a queue owned by a queue manager, and another program retrieves messages from the queue (see Figure 1).

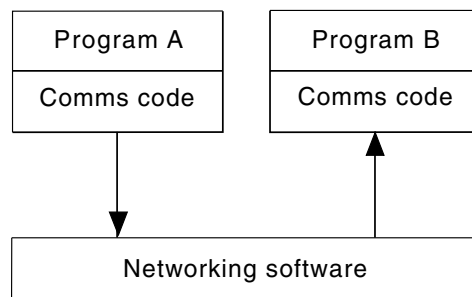
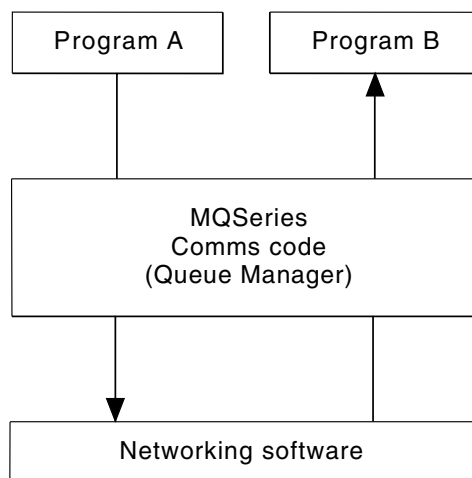
Traditional communication between programs**Communication by message queuing**

Figure 1. Message queuing compared with traditional communication

As with electronic mail, the individual messages that may be part of a transaction, travel through a network on a store-and-forward basis. If a link between nodes fails, the message is kept until the link is restored, or the operator or program redirects the message.

Main features

The mechanism by which a message moves from queue to queue is hidden from the programs. Therefore the programs are simpler.

Time-independent communication

Programs requesting others to do work do not have to wait for the reply to a request. They can do other work, and process the reply either when it arrives or at a later time. When writing a messaging application, you need not know (or be concerned) when a program sends a message, or when the target is able to receive the message. The message is not lost; it is retained by the queue manager until the target is ready to process it. The message stays on the queue until it is removed by a program.

Small programs

Message queuing allows you to exploit the advantages of using small, self-contained programs. Instead of a single, large program performing all the parts of a job sequentially, you can spread the job over several smaller, independent programs. The requesting program sends messages to each of the separate programs, asking them to perform their function; when each program is complete, the results are sent back as one or more messages.

Event-driven processing

Programs can be controlled according to the state of queues. For example, you can arrange for a program to start as soon as a message arrives on a queue, or you can specify that the program does not start until there are, for example, 10 messages above a certain priority on the queue, or 10 messages of any priority on the queue.

Message priority

A program can assign a priority to a message when it puts the message on a queue. This determines the position in the queue at which the new message is added.

Programs can get messages from a queue either in the order in which the messages appear in the queue, or by getting a specific message. (A program may want to get a specific message if it is looking for the reply to a request it sent earlier.)

Security

Authorization checks are carried out on each resource, using the tables that are set up and maintained by the MQSeries administrator.

- RACF or other external security managers may be used within MQSeries for OS/390.
- There is no authorization checking within MQSeries for OS/2 Warp; however, an interface is provided to enable you to build and install your own.
- Within MQSeries on UNIX systems, AS/400, Compaq (DIGITAL) OpenVMS, Tandem NonStop Kernel, and Windows NT, a security manager, the Object Authority Manager (OAM), is provided as an installable service. By default, the OAM is active.
- On VSE/ESA, security is provided by CICS.

Data integrity

Data integrity is provided via units of work. The synchronization of the start and end of units of work is fully supported as an option on each MQGET/MQPUT, allowing the results of the unit of work to be committed or rolled back. Syncpoint support operates either internally or externally to MQSeries depending on the form of syncpoint coordination selected for the application.

Recovery support

For recovery to be possible, all persistent MQSeries updates are logged. Hence, in the event that recovery is necessary, all persistent messages will be restored, all in-flight transactions will be rolled back and any syncpoint commit and backouts will be handled in the normal way of the syncpoint manager in control. For more information on persistent messages, see “Message persistence” on page 34.

MQSeries clients and servers

A server application will not have to be changed to be able to support additional MQSeries clients on new platforms.

Similarly, the MQSeries client will, without change, be able to function with additional types of server. See the *MQSeries Clients* book for more information.

Benefits of message queuing to the application designer and developer

Some of the benefits of message queuing are:

- You can design applications using small programs that you can share between many applications.
- You can quickly build new applications by reusing these building blocks.
- Applications written to use message queuing techniques are not affected by changes in the way queue managers work.
- You do not need to use any communication protocols. The queue manager deals with all aspects of communication for you.
- Programs that receive messages need not be running at the time messages are sent to them. The messages are retained on queues.

Designers can reduce the cost of their applications because development is faster, fewer developers are needed, and demands on programming skill are lower than those for applications that do not use message queuing.

What can you do with MQSeries products?

MQSeries products are queue managers and application enablers. They support the IBM Message Queue Interface (MQI) through which programs can put messages on a queue and get messages from a queue.

MQSeries for OS/390

With MQSeries for OS/390 you can write applications that:

- Use message queuing within CICS or IMS.
- Send messages between batch, CICS, and IMS applications, selecting the most appropriate environment for each function.
- Send messages to applications that run on other MQSeries platforms.
- Process several messages together as a single unit of work that can be committed or backed out.
- Send messages to and interact with IMS applications by means of the IMS bridge.
- Participate in units of work coordinated by RRS.

Uses of MQSeries

See “Appendix A. Language compilers and assemblers” on page 471 for details of the supported programming languages.

Each environment within OS/390 has its own characteristics, advantages, and disadvantages. The advantage of MQSeries for OS/390 is that applications are not tied to any one environment, but can be distributed to take advantage of the benefits of each environment. For example, you can develop end-user interfaces using TSO or CICS, you can run processing-intensive modules in OS/390 batch, and you can run database applications in IMS or CICS. In all cases, the various parts of the application can communicate using messages and queues.

Designers of MQSeries applications must be aware of the differences and limitations imposed by these environments. For example:

- MQSeries provides facilities that allow intercommunication between queue managers (this is known as *distributed queuing*).
- Methods of committing and backing out changes differ between the batch and CICS environments.
- MQSeries for OS/390 provides support in the IMS environment for online message processing programs (MPPs), interactive fast path programs (IFPs), and batch message processing programs (BMPs). If you are writing batch DL/I programs, follow the guidance given in this book for OS/390 batch programs.
- Although multiple instances of MQSeries for OS/390 can exist on a single OS/390 system, a CICS region can connect to only one queue manager at a time. However, more than one CICS region can be connected to the same queue manager. In the IMS and OS/390 batch environments, programs can connect to more than one queue manager.
- MQSeries for OS/390 allows local queues to be shared by a group of queue managers, giving improved throughput and availability. Such queues are called *shared queues*, and the queue managers form a *queue-sharing group*, which can process messages on the same shared queues. Batch applications can connect to one of several queue managers within a queue-sharing group by specifying the queue-sharing group name, instead of a particular queue manager name. This is known as *group batch attach*, or more simply *group attach*. See the *MQSeries for OS/390 Concepts and Planning Guide* for a full discussion of queue-sharing groups.

The differences between the supported environments, and their limitations, are discussed further in “Chapter 15. Using and writing applications on MQSeries for OS/390” on page 213.

MQSeries for non-OS/390 platforms

With MQSeries for non-OS/390 platforms you can write applications that:

- Send messages to other applications running under the same operating systems. The applications can be on either the same or another system.
- Send messages to applications that run on other MQSeries platforms.
- Use message queuing from within CICS Transaction Server for OS/2, CICS for AS/400, TXSeries for AIX, TXSeries for HP-UX, CICS for Siemens Nixdorf SINIX, TXSeries for Sun Solaris, and TXSeries for Windows NT, DOS, and Windows 3.1 applications.
- Use message queuing from within Encina for AIX, HP-UX, SINIX, Sun Solaris, and Windows NT.
- Use message queuing from within Sybase for AIX, Sun Solaris, and Windows NT.

- Use message queuing from within Tuxedo for AIX, AT&T, HP-UX, SINIX and DC/OSx, Sun Solaris, Compaq Tru64 UNIX, and Windows NT.
- MQSeries can act as a transaction manager, and will coordinate updates made by external resource managers within MQSeries units of work. These external resource managers must comply to the X/OPEN XA interface.
- Process several messages together as a single unit of work that can be committed or backed out.
- Run from a full MQSeries environment, or run from an MQSeries client environment on the following platforms:
 - AS/400
 - Digital OpenVMS
 - DOS
 - OS/2
 - UNIX systems
 - VM/ESA[®]
 - Windows NT
 - Windows 3.1
 - Windows 95 and Windows 98

See “Appendix A. Language compilers and assemblers” on page 471 for details of the supported programming languages.

Chapter 2. Overview of application design

This chapter introduces the design of MQSeries applications, under these headings:

- “Planning your design”
- “Using MQSeries objects” on page 14
- “Designing your messages” on page 15
- “MQSeries techniques” on page 16
- “Application programming” on page 17
- “Testing MQSeries applications” on page 20

These subjects are discussed in greater detail in the remaining chapters of this book.

Planning your design

When you have decided how your applications are able to take advantage of the platforms and environments available to you, you need to decide how to use the features offered by MQSeries. Some of the key aspects are:

What types of queue should you use?

Do you want to create a queue each time you need one, or do you want to use queues that have already been set up? Do you want to delete a queue when you have finished using it, or is it going to be used again? Do you want to use alias queues for application independence? To see what types of queues are supported, refer to “Queues” on page 41.

Should you make use of shared queues and queue-sharing groups?

Only available on MQSeries for OS/390.

You may want to take advantage of the increased availability, scalability, and work-load balancing that are possible when you use shared queues with queue-sharing groups. See the *MQSeries for OS/390 Concepts and Planning Guide* for a full discussion of this topic.

What types of message should you use?

You may want to use datagrams for simple messages, but request messages (for which you expect replies) for other situations. You may want to assign different priorities to some of your messages.

How can you control your MQSeries programs?

You may want to start some programs automatically or make programs wait until a particular message arrives on a queue, (using the MQSeries *triggering* feature, see “Chapter 14. Starting MQSeries applications using triggers” on page 191). Alternatively, you may want to start up another instance of an application when the messages on a queue are not getting processed fast enough (using the MQSeries *instrumentation events* feature as described in the *MQSeries Event Monitoring* book).

Will your application run on an MQSeries client?

The full MQI is supported in the client environment and this enables almost any MQSeries application to be relinked to run on an MQSeries client. Link the application on the MQSeries client to the MQIC library, rather than to the MQI library. The exceptions are:

- An application that needs syncpoint coordination with other resource managers.

Planning your design

- Get(signal) on OS/390 is not supported.

Note: An application running on an MQSeries client may connect to more than one queue manager concurrently, or use a queue manager name with an asterisk (*) on an MQCONN or MQCONNX call. The application will have to be changed if you want to link to the queue manager libraries instead of the client libraries, as this function will not be available.

See the *MQSeries Clients* book for more information.

How can you secure your data and maintain its integrity?

You can use the context information that is passed with a message to test that the message has been sent from an acceptable source. You can use the syncpointing facilities provided by MQSeries or your operating system to ensure that your data remains consistent with other resources (see “Chapter 13. Committing and backing out units of work” on page 177 for further details). You can use the *persistence* feature of MQSeries messages to assure the delivery of important messages.

How should you handle exceptions and errors?

You need to consider how to process messages that cannot be delivered, and how to resolve error situations that are reported to you by the queue manager. For some reports, you must set report options on MQPUT.

The remainder of this chapter introduces the features and techniques that MQSeries provides to help you answer questions like these.

Using MQSeries objects

The MQI uses the following types of object:

- Queue managers
- Queues
- Namelists (MQSeries Version 5 products only)
- Process definitions
- Channels
- Storage classes (MQSeries for OS/390 only)

These objects, and queue-sharing groups (only on MQSeries for OS/390) which are not strictly objects, are discussed in “Chapter 4. MQSeries objects” on page 39.

Each object is identified by an *object descriptor* (MQOD), which you use when you write MQSeries programs. However, with the exception of dynamic queues, these objects must be defined to the queue manager before you can work with them.

You define objects using:

- The PCF commands described in the *MQSeries Programmable System Management* book (not on OS/390 or VSE/ESA)
- The MQSC commands described in the *MQSeries MQSC Command Reference* manual (not on VSE/ESA)
- The MQSeries for OS/390 operations and control panels, described in the *MQSeries for OS/390 System Administration Guide*
- The MQSeries Explorer or MQSeries Web Administration (Windows NT only)
- The MQSeries Master Terminal (MQMT) transaction (VSE/ESA only)

You can also display or alter the attributes of objects, or delete the objects.

Alternatively, for sequences of MQSeries for OS/390 commands that you use regularly, you can write administration programs that create messages containing commands and that put these messages on the system-command input queue. The queue manager processes the messages on this queue in the same way that it processes commands entered from the command line or from the operations and control panels. This technique is described in the *MQSeries for OS/390 System Administration Guide*, and demonstrated in the Mail Manager sample application delivered with MQSeries for OS/390. For a description of this sample, see “Chapter 35. Sample programs for MQSeries for OS/390” on page 421.

For sequences of MQSeries for AS/400 commands you use regularly you can write CL programs.

For sequences of MQSeries commands on OS/2, Windows NT, and UNIX systems, you can use the MQSC facility to run a series of commands held in a file. For information on how to use this facility, see the *MQSeries MQSC Command Reference* manual.

Designing your messages

You create a message when you use an MQI call to put the message on a queue. As input to the call, you supply some control information in a *message descriptor* (MQMD) and the data that you want to send to another program. But at the design stage, you need to consider the following questions, because they affect the way you create your messages:

What type of message should I use?

Are you designing a simple application in which you can send a message, then take no further action? Or are you asking for a reply to a question? If you are asking a question, you may include in the message descriptor the name of the queue on which you want to receive the reply.

Do you want your request and reply messages to be synchronous? This implies that you set a timeout period for the reply to answer your request, and if you do not receive the reply within that period, it is treated as an error.

Or would you prefer to work asynchronously, so that your processes do not have to depend upon the occurrence of specific events, such as common timing signals?

Another consideration is whether you have all your messages inside a unit of work.

Should I assign different priorities to some of the messages I create?

You can assign a priority value to each message, and define the queue so that it maintains its messages in order of their priority. If you do this, when another program retrieves a message from the queue, it always gets the message with the highest priority. If the queue does not maintain its messages in priority order, a program that retrieves messages from the queue will retrieve them in the order in which they were added to the queue.

Programs can also select a message using the identifier that the queue manager assigned when the message was put on the queue. Alternatively, you can generate your own identifiers for each of your messages.

Will my messages be discarded when the queue manager restarts?

The queue manager preserves all persistent messages, recovering them

Message design

when necessary from the MQSeries log files, when it is restarted. Nonpersistent messages and temporary dynamic queues are not preserved. Any messages that you do not want discarded must be defined as persistent at the time they are created. When writing an application for MQSeries for OS/2 Warp, MQSeries for Windows NT, or MQSeries on UNIX systems, make sure that you know how your system has been set up in respect of log file allocation to reduce the risk of designing an application that will run to the log file limits.

Note that messages on shared queues (only available on MQSeries for OS/390) cannot be made persistent, and so are not preserved when the queue manager restarts. However, messages on shared queues are held in the Coupling Facility (CF), and are lost only if the CF fails.

Do I want to give information about myself to the recipient of my messages?

Normally, the queue manager sets the user ID, but suitably authorized applications can also set this field, so that you can include your own user ID and other information that the receiving program can use for accounting or security purposes.

MQSeries techniques

For a simple MQSeries application, you need to decide which MQSeries objects to use in your application, and which types of message you want to use. For a more advanced application, you may want to use some of the techniques introduced in the following sections.

Waiting for messages

A program that is serving a queue can await messages by:

- Making periodic calls on the queue to see whether a message has arrived (*polling*).
- Waiting until either a message arrives, or a specified time interval expires (see “Waiting for messages” on page 141).
- Setting a signal so that the program is informed when a message arrives (MQSeries for OS/390 and MQSeries for Windows V2.1 only). For information about this, see “Signaling” on page 142.

Correlating replies

In MQSeries applications, when a program receives a message that asks it to do some work, the program usually sends one or more reply messages to the requester. To help the requester to associate these replies with its original request, an application can set a *correlation identifier* field in the descriptor of each message. Programs should copy the message identifier of the request message into the correlation identifier field of their reply messages.

Setting and using context information

Context information is used for associating messages with the user who generated them, and for identifying the application that generated the message. Such information is useful for security, accounting, auditing, and problem determination.

When you create a message, you can specify an option that requests that the queue manager associates default context information with your message.

For more information on using and setting context information, see “Message context” on page 36.

Starting MQSeries programs automatically

MQSeries *triggering* enables a program to be started automatically when messages arrive on a queue. You can set trigger conditions on a queue so that a program is started to process that queue:

- Every time a message arrives on the queue
- When the first message arrives on the queue
- When the number of messages on the queue reaches a predefined number

For more information on triggering, see “Chapter 14. Starting MQSeries applications using triggers” on page 191.

Note: Triggering is just one way of starting a program automatically. For example, you can start a program automatically on a timer using non-MQSeries facilities.

Generating MQSeries reports

You can request the following reports within an application:

- Exception reports
- Expiry reports
- Confirm-on-arrival (COA) reports
- Confirm-on-delivery (COD) reports
- Positive action notification (PAN) reports
- Negative action notification (NAN) reports

These are described in “Report messages” on page 25.

Clusters and message affinities

Before starting to use clusters with multiple definitions for the same queue, you must examine your applications to see whether there are any that require an exchange of related messages. Within a cluster, a message may be routed to *any* queue manager that hosts an instance of the appropriate queue. Therefore, the logic of applications with message affinities may be upset.

For example, you may have two applications that rely on a series of messages flowing between them in the form of questions and answers. It may be important that all the questions are sent to the same queue manager and that all the answers are sent back to the other queue manager. In this situation, it is important that the workload management routine does not send the messages to any queue manager that just happens to host an instance of the appropriate queue.

You should attempt, where possible, to remove the affinities. Removing message affinities improves the availability and scalability of applications.

For more information see the *MQSeries Queue Manager Clusters* book.

Application programming

MQSeries supports the IBM Message Queue Interface (MQI) and the Application Messaging Interface (AMI). The MQI includes a set of calls with which you can send and receive messages, and manipulate MQSeries objects. The Application Messaging Interface (AMI), a simpler interface than the MQI, may be sufficient in some cases. Further information about the AMI is outside the scope of this book. For full details, refer to the *MQSeries Application Messaging Interface* book.

Application programming

Call interface

The MQI calls allow you to:

- Connect programs to, and disconnect programs from, a queue manager
- Open and close objects (such as queues, queue managers, namelists, and processes)
- Put messages on queues
- Receive messages from a queue, or browse them (leaving them on the queue)
- Inquire about the attributes (or properties) of MQSeries objects, and set some of the attributes of queues
- Commit and back out changes made within a unit of work, in environments where there is no natural syncpoint support, for example, OS/2 and UNIX systems
- Coordinate queue manager updates and updates made by other resource managers

The MQI provides *structures* (groups of fields) with which you supply input to, and get output from, the calls. It also provides a large set of named constants to help you supply options in the parameters of the calls. The definitions of the calls, structures, and named constants are supplied in data definition files for each of the supported programming languages. Also, default values are set within the MQI calls.

Design for performance - hints and tips

Here are a few ideas to help you design efficient applications:

- Design your application so that processing goes on in parallel with a user's thinking time:
 - Display a panel and allow the user to start typing while the application is still initializing.
 - Don't be afraid to get the data you need in parallel from different servers.
- Keep connections and queues open if you are going to reuse them instead of repeatedly opening and closing, connecting and disconnecting.

Note: However, a server application which is putting only one message should use MQPUT1.

- Keep your messages within a unit of work, so that they can be committed or backed out simultaneously.
- Use the nonpersistent option for messages that do not need to be recoverable.

Programming platforms

MQSeries for OS/390

MQSeries for OS/390 operates under OS/390 Version 2.6 and subsequent compatible releases. You can run MQSeries for OS/390 programs in the CICS Transaction Server for OS/390, CICS for MVS/ESA, IMS/ESA, and OS/390 environments. See "Appendix A. Language compilers and assemblers" on page 471 for details of the programming languages supported by MQSeries for OS/390. See *MQSeries for OS/390 Concepts and Planning Guide* for information on prerequisites.

UNIX systems

MQSeries for AIX operates under AIX Version 4.2, Version 4.3.x, and subsequent compatible releases. You can run MQSeries for AIX programs

Application programming

from within CICS for AIX, TXSeries for AIX, Encina for AIX, and Tuxedo for AIX. Applications using threads are supported by MQSeries for AIX.

MQSeries for AT&T GIS UNIX operates under AT&T GIS UNIX Version 3² and subsequent compatible releases. You can run MQSeries for AT&T GIS UNIX programs from within Tuxedo for AT&T.

MQSeries for Compaq (DIGITAL) OpenVMS operates under VMS Version 6.2 and VMS Version 7.1.

MQSeries for Compaq Tru64 UNIX operates under DIGITAL UNIX (Compaq Tru64 UNIX) Version 4.0D or Version 4.0E, or Compaq Tru64 UNIX Version 4.0F or Version 5.0. Applications using threads are supported by MQSeries for Compaq Tru64 UNIX.

MQSeries for HP-UX operates under HP-UX Version 10.20 and Version 11.0. You can run MQSeries for HP-UX programs from within TXSeries, Encina, and Tuxedo for HP-UX. Applications using threads are supported by MQSeries for HP-UX.

MQSeries for SINIX and DC/OSx operates under SINIX and DC/OSx Version 2.1 and subsequent compatible releases. You can run MQSeries for SINIX and DC/OSx programs from within CICS for Siemens Nixdorf SINIX, and Tuxedo for SINIX and DC/OSx. You can also run MQSeries for SINIX programs from within Encina for SINIX.

MQSeries for Sun Solaris operates under Sun Solaris Version 2.6 (with patches 105210-13 and 105568-10), Version 7, and subsequent compatible releases. You can run MQSeries for Sun Solaris programs from within CICS, TXSeries, Encina, and Tuxedo for Sun Solaris. Applications using threads are supported by MQSeries for Sun Solaris.

See “Appendix A. Language compilers and assemblers” on page 471 for details of the programming languages supported by MQSeries on UNIX systems.

MQSeries for AS/400

MQSeries for AS/400 operates under OS/400 Version 4 Release 4 and subsequent compatible releases. You can run MQSeries for AS/400 programs in the CICS for AS/400 environment. See “Appendix A. Language compilers and assemblers” on page 471 for details of the programming languages supported by MQSeries for AS/400. Applications using threads are supported by MQSeries for AS/400.

MQSeries for OS/2 Warp

MQSeries for OS/2 Warp operates under OS/2 Warp Version 4.0, OS/2 Warp Server V4.0, OS/2 Warp Server Advanced SMP feature, OS/2 Workspace On-Demand, OS/2 e-business Server, and subsequent compatible releases. You can run MQSeries for OS/2 Warp programs in the CICS and CICS Transaction Server environment. See “Appendix A. Language compilers and assemblers” on page 471 for details of the programming languages supported by MQSeries for OS/2 Warp.

MQSeries for Tandem NonStop Kernel

MQSeries for Tandem NonStop Kernel operates under Tandem NSK operating system version D3x, D4x, G02, or later G0x with TMF and

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

Application programming

PATHWAY. See “Appendix A. Language compilers and assemblers” on page 471 for details of the programming languages supported by MQSeries for Tandem NonStop Kernel.

MQSeries for VSE/ESA

MQSeries for VSE/ESA, V2.1 operates under VSE/ESA V2.3 and subsequent compatible releases, with CICS for VSE/ESA V2.3. See “Appendix A. Language compilers and assemblers” on page 471 for details of the programming languages supported by MQSeries for VSE/ESA.

MQSeries for Windows

MQSeries for Windows, V2.0 operates under Windows Version 3.1, Windows 95, and the WIN-OS/2 environment within OS/2. MQSeries for Windows V2.1 operates under Windows 95, Windows 98, and Windows NT V4. See “Appendix A. Language compilers and assemblers” on page 471 for details of the programming languages supported by MQSeries for Windows.

MQSeries for Windows NT

MQSeries for Windows NT operates under Windows NT Version 4.0 (service pack 4) and subsequent compatible releases. You can run MQSeries for Windows NT programs from within CICS, TXSeries, Encina, and Tuxedo for Windows NT. See “Appendix A. Language compilers and assemblers” on page 471 for details of the programming languages supported by MQSeries for Windows NT.

Applications for more than one platform

Will your application run on more than one platform? Do you have a strategy to move to a different platform from the one you use today? If the answer to either of these questions is “yes,” you need to make sure that you code your programs for platform independence.

If you are using C, make sure that you code in ANSI standard C. Use a standard C library function rather than an equivalent platform-specific function even if the platform-specific function is faster or more efficient. The exception is when efficiency in the code is paramount, when you should code for both situations using `#ifdef`. For example:

```
#ifdef _OS2
    OS/2 specific code
#else
    generic code
#endif
```

When the time comes to move the code to another platform, you can now search the source for `#ifdef` with the platform specific identifiers, in this example `_OS2`, and add or change code as necessary.

It is worth considering keeping portable code in separate source files from the platform-specific code, and using a simple naming convention to split the categories.

Testing MQSeries applications

The application development environment for MQSeries programs is no different from that for any other application, so you can use the same development tools as well as the MQSeries trace facilities. This is most noticeable on OS/2 and UNIX systems where there is a wide selection.

When testing CICS applications with MQSeries for OS/390, you can use the CICS Execution Diagnostic Facility (CEDF). CEDF traps the entry and exit of every MQI call as well as calls to all CICS services. Also, in the CICS environment, you can write an API-crossing exit program to provide diagnostic information before and after every MQI call. For information on how to do this, see “Chapter 15. Using and writing applications on MQSeries for OS/390” on page 213.

When testing AS/400 applications, you can use the Extended Program Model Debugger. To start this, use the STRDBG command.

Chapter 3. MQSeries messages

An MQSeries message consists of two parts:

- Message descriptor
- Application data

Figure 2 represents a message and shows how it is logically divided into message data and application data.

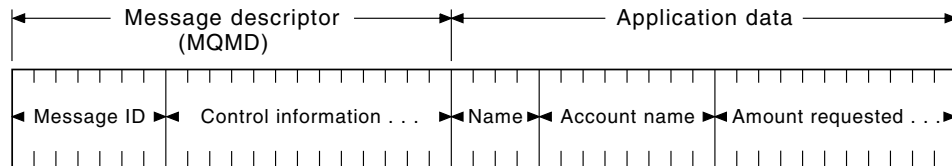


Figure 2. Representation of a message

The application data carried in an MQSeries message is not changed by a queue manager unless data conversion is carried out on it. Also, MQSeries does not put any restrictions on the content of this data. The length of the data in each message cannot exceed the value of the *MaxMsgLength* attribute of both the queue and queue manager.

In MQSeries for AIX, MQSeries for AS/400, MQSeries for HP-UX, MQSeries for OS/2 Warp, MQSeries for Sun Solaris, MQSeries for Compaq Tru64 UNIX, and MQSeries for Windows NT, the *MaxMsgLength* defaults to 100 MB (104 857 600 bytes).

Note: If you are intending to use MQSeries messages greater than 15 MB on an AS/400 machine, see “Chapter 20. Building your application on AS/400” on page 291.

In MQSeries for OS/390, the *MaxMsgLength* attribute of the queue manager is fixed at 100 MB and the *MaxMsgLength* attribute of the queue defaults to 4 MB (4 194 304 bytes) which you can amend up to 100 MB if required. However, the limit is 63 KB when using shared queues accessed by queue managers in a queue-sharing group.

In MQSeries for AT&T GIS UNIX, MQSeries for Compaq (DIGITAL) OpenVMS, MQSeries for SINIX and DC/OSx, MQSeries for Tandem NonStop Kernel, MQSeries for VSE/ESA, MQSeries for Windows, V2.0, and MQSeries for Windows, V2.1, the *MaxMsgLength* defaults to 4 MB.

However, you should make your messages slightly shorter than the value of the *MaxMsgLength* attribute in some circumstances. See “The data in your message” on page 111 for more information.

You create a message when you use the MQPUT or MQPUT1 MQI call. As input to these calls, you supply the control information (such as the priority of the message, and the name of a reply queue) and your data. These calls put the message on a queue. See the *MQSeries Application Programming Reference* manual for more information on these calls.

Messages

This chapter introduces MQSeries messages, under these headings:

- “Message descriptor”
- “Types of message”
- “Format of message control information and message data” on page 30
- “Message priorities” on page 32
- “Message groups” on page 32
- “Message persistence” on page 34
- “Selecting messages from queues” on page 34
- “Messages that fail to be delivered” on page 35
- “Messages that are backed out” on page 35
- “Reply-to queue and queue manager” on page 36
- “Message context” on page 36

Message descriptor

You can access message control information using the MQMD structure, which defines the *message descriptor*. For a full description of the MQMD structure, see the *MQSeries Application Programming Reference* manual.

See “Message context” on page 36 for a description of how to use the fields within the MQMD that contain information about the origin of the message.

Additional information for grouping and segmenting messages (see “Message groups” on page 32) is provided in Version 2 of the Message Descriptor (or the MQMDE). This is the same as the Version 1 Message Descriptor but has additional fields as described in the *MQSeries Application Programming Reference* manual.

Types of message

There are four types of message defined by MQSeries:

- Datagram
- Request
- Reply
- Report

Applications can use the first three types of messages to pass information between themselves. The fourth type, report, is for applications and queue managers to use to report information about events such as the occurrence of an error.

Each type of message is identified by an MQMT_* value. You can also define your own types of message. For the range of values you can use, see the description of the *MsgType* field in the *MQSeries Application Programming Reference* manual.

Datagrams

You should use a *datagram* when you do not require a reply from the application that receives the message (that is, gets the message from the queue).

An example of an application that could use datagrams is one that displays flight information in an airport lounge. A message could contain the data for a whole screen of flight information. Such an application is unlikely to request an acknowledgement for a message because it probably does not matter if a message is not delivered. The application will send an update message after a short period of time.

Request messages

You should use a *request message* when you want a reply from the application that receives the message.

An example of an application that could use request messages is one that displays the balance of a checking account. The request message could contain the number of the account, and the reply message would contain the account balance.

If you want to link your reply message with your request message, there are two options:

- You can give your application the responsibility of ensuring that it puts information into the reply message that relates to the request message.
- You can use the report field in the message descriptor of your request message to specify the content of the *MsgId* and *CorrelId* fields of the reply message:
 - You can request that either the *MsgId* or the *CorrelId* of the original message is to be copied into the *CorrelId* field of the reply message (the default action is to copy *MsgId*).
 - You can request that either a new *MsgId* is generated for the reply message, or that the *MsgId* of the original message is to be copied into the *MsgId* field of the reply message (the default action is to generate a new message identifier).

Reply messages

You should use a *reply message* when you reply to another message.

When you create a reply message, you should respect any options that were set in the message descriptor of the message to which you are replying. Report options specify the content of the message identifier (*MsgId*) and correlation identifier (*CorrelId*) fields. These fields allow the application that receives the reply to correlate the reply with its original request.

Report messages

Report messages inform applications about events such as the occurrence of an error when processing a message. They can be generated by:

- A queue manager,
- A message channel agent (for example, if they cannot deliver the message),

or

- An application (for example, if it cannot use the data in the message).

Note that report messages can be generated at any time, and they may arrive on a queue when your application is not expecting them.

Types of report message

When you put a message on a queue, you can select to receive:

- An *exception report message*. This is sent in response to a message that had the exceptions flag set. It is generated by the message channel agent (MCA) or the application.
- An *expiry report message*. This indicates that an application attempted to retrieve a message that had reached its expiry threshold; the message is marked to be discarded. This type of report is generated by the queue manager.
- A *confirmation of arrival (COA) report message*. This indicates that the message has reached its target queue. It is generated by the queue manager.

Types of message

- A *confirmation of delivery (COD) report message*. This indicates that the message has been retrieved by a receiving application. It is generated by the queue manager.
- A *positive action notification (PAN) report message*. This indicates that a request has been successfully serviced (that is, the action requested in the message has been performed successfully). This type of report is generated by the application.
- A *negative action notification (NAN) report message*. This indicates that a request has *not* been successfully serviced (that is, the action requested in the message has *not* been performed successfully). This type of report is generated by the application.

Note: Each type of report message contains one of the following:

- The entire original message
- The first 100 bytes of data in the original message
- No data from the original message

You may request more than one type of report message when you put a message on a queue. If you select the delivery confirmation report message and the exception report message options, in the event that the message fails to be delivered, you will receive an exception report message. However, if you select only the delivery confirmation report message option and the message fails to be delivered, you *will not* get an exception report message.

The report messages you request, when the criteria for generating a particular message are met, are the only ones you will receive.

Report message options

You have the option to *discard* a message after an exception has arisen. If you select the discard option, and have requested an exception report message, the report message goes to the *ReplyToQ* and *ReplyToQMGr*, and the original message is discarded.

Note: A benefit of this is you can reduce the number of messages going to the dead-letter queue. However, it does mean that your application, unless it sends only datagram messages, has to deal with returned messages. When an exception report message is generated, it inherits the persistence of the original message.

If a report message cannot be delivered (if the queue is full, for instance), the report message will be placed on the dead-letter queue.

If you wish to receive a report message, you must specify the name of your reply-to queue in the *ReplyToQ* field; otherwise the MQPUT or MQPUT1 of your original message will fail with MQRC_MISSING_REPLY_TO_Q.

You can use other report options in the message descriptor (MQMD) of a message to specify the content of the *MsgId* and *CorrelId* fields of any report messages that are created for the message:

- You can request that either the *MsgId* or the *CorrelId* of the original message is to be copied into the *CorrelId* field of the report message. The default action is to copy the message identifier. MQRO_COPY_MSG_ID_TO_CORRELID should be used because it enables the sender of a message to correlate the reply or report message with the original message. The correlation identifier of the reply or report message will be identical to the message identifier of the original message.

Types of message

- You can request that either a new *MsgId* is generated for the report message, or that the *MsgId* of the original message is to be copied into the *MsgId* field of the report message. The default action is to generate a new message identifier. MQRO_NEW_MSG_ID should be used because it ensures that each message in the system has a different message identifier, and therefore can be distinguished unambiguously from all other messages in the system.
- Specialized applications may need to use MQRO_PASS_MSG_ID and, or MQRO_PASS_CORREL_ID. However, the application that reads the messages from the queue may need careful design in order to ensure that it will work correctly. In particular when the queue contains multiple messages with the same message identifier.

Server applications should check the settings of these flags in the request message, and set the *MsgId* and *CorrelId* fields in the reply or report message appropriately.

Applications which act as intermediaries between a requester application and a server application should not need to check the settings of these flags. This is because these applications usually need to forward the message to the server application with the *MsgId*, *CorrelId*, and *Report* fields unchanged. This allows the server application to copy the *MsgId* from the original message in the *CorrelId* field of the reply message.

When generating a report about a message, server applications should test to see if any of these options have been set.

For more information on how to use report messages, see the description of the *Report* field in the *MQSeries Application Programming Reference* manual.

To indicate the nature of the report, queue managers use a range of feedback codes. They put these codes in the *Feedback* field of the message descriptor of a report message. Queue managers can also return MQI reason codes in the *Feedback* field. MQSeries defines a range of feedback codes for applications to use.

For more information on feedback and reason codes, see the description of the *Feedback* field in the *MQSeries Application Programming Reference* manual.

An example of a program that could use a feedback code is one that monitors the work loads of other programs serving a queue. If there is more than one instance of a program serving a queue, and the number of messages arriving on the queue no longer justifies this, such a program could send a report message (with the feedback code MQFB_QUIT) to one of the serving programs to indicate that the program should terminate its activity. (A monitoring program could use the MQINQ call to find out how many programs are serving a queue.)

Reports and segmented messages

Segmented messages are supported on MQSeries Version 5 products only, except MQSeries for OS/390.

If a message is segmented (see “Message segmentation” on page 136 for a description of this) and you ask for reports to be generated, you may receive more reports than you would have done had the message not been segmented.

MQSeries-generated reports

If you segment your messages or allow the queue manager to do so, there is only one case in which you can expect to receive a single report for the entire message.

Types of message

This is when you have requested only COD reports, and you have specified MQGMO_COMPLETE_MSG on the getting application.

In other cases your application must be prepared to deal with several reports; usually one for each segment.

Note: If you segment your messages, and you need only the first 100 bytes of the original message data to be returned, you must change the setting of the report options to ask for reports with *no* data for segments that have an offset of 100 or more. If you do not do this, and you leave the setting so that each segment requests 100 bytes of data, and you retrieve the report messages with a single MQGET specifying MQGMO_COMPLETE_MSG, the reports assemble into a large message containing 100 bytes of read data at each appropriate offset. If this happens, you need a large buffer or you need to specify MQGMO_ACCEPT_TRUNCATED_MSG.

Application-generated reports

If your application generates reports, you should always copy the MQSeries headers that are present at the start of the original message data to the report message data. Then add none, 100 bytes, or all of the original message data (or whatever other amount you would normally include) to the report message data.

You can recognize the MQSeries headers that must be copied by looking at the successive Format names, starting with the MQMD and continuing through any headers present. The following Format names indicate these MQSeries headers:

- MQMDE
- MQDLH
- MQXQH
- MQIIH
- MQH*

MQH* means any name starting with the characters MQH.

The Format name occurs at specific positions for MQDLH and MQXQH, but for the other MQSeries headers it occurs at the same position. The length of the header is contained in a field that also occurs at the same position for MQMDE, MQIMS and all MQH* headers.

If you are using a Version 1 of the MQMD, and you are reporting on a segment, or a message in a group, or a message for which segmentation is allowed, the report data must start with an MQMDE. You should set the *OriginalLength* field to the length of the original message data *excluding* the lengths of any MQSeries headers that you find.

Retrieval of reports

If you ask for COA or COD reports, you can ask for them to be reassembled for you with MQGMO_COMPLETE_MSG. An MQGET with MQGMO_COMPLETE_MSG is satisfied when enough report messages (of a single type, for example COA, and with the same *GroupId*) are present on the queue to represent one complete original message. This is true even if the report messages themselves do not contain the complete original data; the *OriginalLength* field in each report message gives the length of original data *represented* by that report message, even if the data itself is not present.

This technique can be used even if there are several different report types present on the queue (for example, both COA and COD), because an MQGET with MQGMO_COMPLETE_MSG reassembles report messages only if they have the

same *Feedback* code. Note, however, that you cannot normally use the technique for exception reports, since in general these have different *Feedback* codes.

You can use this technique to get a positive indication that the entire message has arrived. However, in most circumstances you need to cater for the possibility that some segments arrive while others may generate an exception (or expiry, if you have allowed this). You cannot use `MQGMO_COMPLETE_MSG` in this case because in general you may get different *Feedback* codes for different segments and, as noted above, you may get more than one report for a given segment. You can, however, use `MQGMO_ALL_SEGMENTS_AVAILABLE`.

To allow for this you may need to retrieve reports as they arrive, and build up a picture in your application of what happened to the original message. You can use the *GroupId* field in the report message to correlate reports with the *GroupId* of the original message, and the *Feedback* field to identify the type of each report message. The way in which you do this depends on your application requirements.

One approach is as follows:

- Ask for COD reports and exception reports.
- After a specific time, check whether a complete set of COD reports has been received using `MQGMO_COMPLETE_MSG`. If so, your application knows that the entire message has been processed.
- If not, and exception reports relating to this message are present, the problem should be handled just as for unsegmented messages, though provision must also be made for 'orphan' segments to be cleaned up at some point.
- If there are segments for which there are no reports of any kind, the original segments (or the reports) may be waiting for a channel to be reconnected, or the network might be overloaded at some point. If no exception reports at all have been received (or if you think that the ones you have may be temporary only), you may decide to let your application wait a little longer.

As before, this is similar to the considerations you have when dealing with unsegmented messages, except that you must also consider the possibility of 'orphan' segments which have to be cleaned up.

If the original message is not critical (for example, if it is a query, or a message that can be repeated later), set an expiry time to ensure that orphan segments are removed.

Back-level queue managers

When a report is generated by a queue manager that supports segmentation, but is received on a queue manager that does *not* support segmentation, the MQMDE structure (which identifies the *Offset* and *OriginalLength* represented by the report) is always included in the report data, in addition to zero, 100 bytes, or all of the original data in the message.

However, if a segment of a message passes through a queue manager that does not support segmentation, you should be aware that if a report is generated there, the MQMDE structure in the original message will be treated purely as data. It will not therefore be included in the report data if zero bytes of the original data have been requested. Without the MQMDE, the report message may not be useful.

You should therefore request at least 100 bytes of data in reports if there is a possibility that the message might travel through a back-level queue manager.

Format of message control information and message data

The queue manager is only interested in the format of the control information within a message, whereas applications that handle the message are interested in the format of both the control information and the data.

Format of message control information

Control information in the character-string fields of the message descriptor must be in the character set used by the queue manager. The *CodedCharSetId* attribute of the queue manager object defines this character set. Control information must be in this character set because when applications pass messages from one queue manager to another, message channel agents that transmit the messages use the value of this attribute to determine what data conversion they must perform.

Format of message data

You can specify any of the following:

- The format of the application data
- The character set of the character data
- The format of numeric data

To do this, use these fields:

Format

This indicates to the receiver of a message the format of the application data in the message.

When the queue manager creates a message, in some circumstances it uses the *Format* field to identify the format of that message. For example, when a queue manager cannot deliver a message, it puts the message on a dead-letter (undelivered-message) queue. It adds a header (containing more control information) to the message, and changes the *Format* field to show this.

The queue manager has a number of *built-in formats* with names beginning “MQ”, for example MQFMT_STRING. If these do not meet your needs, you must define your own formats (*user-defined formats*), but you should not use names beginning with “MQ” for these.

When you create and use your own formats, you must write a data-conversion exit to support a program getting the message using MQGMO_CONVERT.

CodedCharSetId

This defines the character set of character data in the message. If you want to set this character set to that of the queue manager, you can set this field to the constant MQCCSI_Q_MGR or MQCCSI_INHERIT.

When you get a message from a queue, you should compare the value of the *CodedCharSetId* field with the value that your application is expecting. If the two values differ, you may need to convert any character data in the message or use a data-conversion message exit if one is available.

Encoding

This describes the format of numeric message data that contains binary integers, packed-decimal integers, and floating point numbers. It is usually encoded according to the particular machine on which the queue manager is running.

When you put a message on a queue, you should normally specify the constant `MQENC_NATIVE` in the *Encoding* field. This means that the encoding of your message data is the same as that of the machine on which your application is running.

When you get a message from a queue, you should compare the value of the *Encoding* field in the message descriptor with the value of the constant `MQENC_NATIVE` on your machine. If the two values differ, you may need to convert any numeric data in the message or use a data-conversion message exit if one is available.

Application data conversion

Application data may need to be converted to the character set and the encoding required by another application where different platforms are concerned. It may be converted at the sending queue manager, or at the receiving queue manager. If the library of built-in formats does not meet your needs, you must define your own. The type of conversion depends on the message format which is specified in the format field of the message descriptor, `MQMD`.

Conversion at the sending queue manager

You must set the `CONVERT` channel attribute to `YES` if you need the sending message channel agent (MCA) to convert the application data.

The conversion is performed at the sending queue manager for certain built-in formats and for user-defined formats if a suitable user exit is supplied.

Built-in formats: These include:

- Messages that are all characters (using the format name `MQFMT_STRING`)
- MQSeries defined messages, for example Programmable Command Formats
MQSeries uses Programmable Command Format messages for administration messages and events (the format name used is `MQFMT_ADMIN` in this case). You can use the same format (using the format name `MQFMT_PCF`) for your own messages, and take advantage of the built-in data conversion.

Note: Messages with `MQFMT_NONE` specified are not converted.

The queue manager built-in formats all have names beginning with `MQFMT`. They are listed and described in the *MQSeries Application Programming Reference* manual under the *Format* field of the Message descriptor (`MQMD`).

Application-defined formats: For user-defined formats, application data conversion must be performed by a data-conversion exit program (for more information, see “Chapter 11. Writing data-conversion exits” on page 155). In a client-server environment, the exit is loaded at the server and conversion takes place there.

Conversion at the receiving queue manager

Application message data may be converted by the receiving queue manager for the built-in formats and user-defined formats. The conversion is performed during the processing of an `MQGET` call if the `MQGMO_CONVERT` option is specified. For details, see the *MQSeries Application Programming Reference* manual.

Coded character sets

MQSeries products support the coded character sets that are provided by the underlying operating system.

Message format

When you create a queue manager, the queue manager coded character set ID (CCSID) used is based on that of the underlying environment. If this is a mixed code page, MQSeries uses the SBCS part of the mixed code page as the queue manager CCSID.

For general data conversion, if the underlying operating system supports DBCS code pages then MQSeries is able to use it.

See the documentation for your operating system for details of the coded character sets that it supports.

You need to consider application data conversion, format names, and user exits when writing applications that span multiple platforms. For details of the MQGET call, the Convert characters call, the MQGMO_CONVERT option, and the built-in formats, see the *MQSeries Application Programming Reference* manual. See “Chapter 11. Writing data-conversion exits” on page 155 for information about invoking and writing data-conversion exits.

Message priorities

You set the priority of a message (in the *Priority* field of the MQMD structure) when you put the message on a queue. You can set a numeric value for the priority, or you can let the message take the default priority of the queue.

The *MsgDeliverySequence* attribute of the queue determines whether messages on the queue are stored in FIFO (first in, first out) sequence, or in FIFO within priority sequence. If this attribute is set to MQMDS_PRIORITY, messages are enqueued with the priority specified in the *Priority* field of their message descriptors; but if it is set to MQMDS_FIFO, messages are enqueued with the default priority of the queue. Messages of equal priority are stored on the queue in order of arrival.

The *DefPriority* attribute of a queue sets the default priority value for messages being put on that queue. This value is set when the queue is created, but it can be changed afterwards. Alias queues, and local definitions of remote queues, may have different default priorities from the base queues to which they resolve. If there is more than one queue definition in the resolution path (see “Name resolution” on page 99), the default priority is taken from the value (at the time of the put operation) of the *DefPriority* attribute of the queue specified in the open command.

The value of the *MaxPriority* attribute of the queue manager is the maximum priority that you can assign to a message processed by that queue manager. You cannot change the value of this attribute. In MQSeries, the attribute has the value 9; you can create messages having priorities between 0 (the lowest) and 9 (the highest).

Message groups

Message groups are supported on MQSeries Version 5 products only, except MQSeries for OS/390.

Messages can occur within groups. This allows ordering of messages (see “Logical and physical ordering” on page 126), and segmentation of large messages (see “Message segmentation” on page 136) within the same group.

The hierarchy within a group is as follows:

Group This is the highest level in the hierarchy and is identified by a *GroupId*. It consists of one or more messages that contain the same *GroupId*. These messages can be stored anywhere on the queue.

Note: The term “message” is used here to denote one item on a queue, such as would be returned by a single MQGET that does not specify MQGMO_COMPLETE_MSG.

Figure 3 shows a group of logical messages:

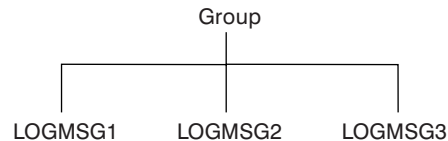


Figure 3. Group of logical messages

Logical message

Logical messages within a group are identified by the *GroupId* and *MsgSeqNumber* fields. The *MsgSeqNumber* starts at 1 for the first message within a group, and if a message is not in a group, the value of the field is 1.

Logical messages within a group can be used to:

- Ensure ordering (if this is not guaranteed under the circumstances in which the message is transmitted).
- Allow applications to group together similar messages (for example, those that must all be processed by the same server instance).

Each message within a group consists of one physical message, unless it is split into segments. Each message is logically a separate message, and only the *GroupId* and *MsgSeqNumber* fields in the MQMD need bear any relationship to other messages in the group. Other fields in the MQMD are independent; some may be identical for all messages in the group whereas others may be different. For example, messages in a group may have different format names, CCSIDs, encodings, and so on.

Segment

Segments are used to handle messages that are too large for either the putting or getting application or the queue manager (including intervening queue managers through which the message passes). For more information about this, see “Message segmentation” on page 136.

A segment of a message is identified by the *GroupId*, *MsgSeqNumber*, and *Offset* fields. The *Offset* field starts at zero for the first segment within a message.

Each segment consists of one physical message that may or may not belong to a group (4 shows an example of messages within a group). A segment is logically part of a single message, so only the *MsgId*, *Offset*, and *SegmentFlag* fields in the MQMD should differ between separate segments of the same message.

Message groups

Figure 4 shows a group of logical messages, some of which are segmented:

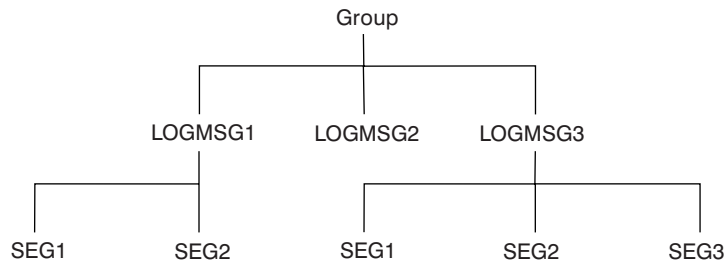


Figure 4. Segmented messages

For a description of logical and physical messages, see “Logical and physical ordering” on page 126. For further information about segmenting messages, see “Message segmentation” on page 136.

Message persistence

Persistent messages are written out to logs and queue data files. If a queue manager is restarted after a failure, it recovers these persistent messages as necessary from the logged data. Messages that are not persistent are discarded if a queue manager stops, whether the stoppage is as a result of an operator command or because of the failure of some part of your system.

When you create a message, if you initialize the message descriptor (MQMD) using the defaults, the persistence for the message will be taken from the *DefPersistence* attribute of the queue specified in the MQOPEN command. Alternatively, you may set the persistence of the message using the *Persistence* field of the MQMD structure to define the message as persistent or not persistent.

The performance of your application is affected when you use persistent messages; the extent of the effect depends on the performance characteristics of the machine’s I/O subsystem and how you use the syncpoint options on each platform:

- A persistent message, outside the current unit of work, is written to disk on every put and get operation. See “Chapter 13. Committing and backing out units of work” on page 177.
- In MQSeries on UNIX systems, MQSeries for Compaq (DIGITAL) OpenVMS, MQSeries for OS/390, MQSeries for OS/2 Warp, MQSeries for VSE/ESA, and MQSeries for Windows NT, a persistent message within the current unit of work is logged only when the unit of work is committed (and the unit of work could contain many queue operations).

On MQSeries for OS/390, only nonpersistent messages can be used when using shared queues accessed by queue managers in a queue-sharing group.

Nonpersistent messages can be used for fast messaging if retrieved outside syncpoint. See the *MQSeries Application Programming Reference* and the *MQSeries Intercommunication* books for further information about fast messages.

Selecting messages from queues

To get a particular message from a queue, you need to use the *MsgId* and *CorrelId* fields of the message descriptor. If you specify Version 2 of the MQMD, the *GroupId* can also be used. (See “Getting a particular message” on page 133.)

The message identifier is usually generated by the queue manager when the message is put on a queue. The queue manager tries to ensure that message identifiers are unique. However, an MQSeries application can specify a particular value for the message identifier.

You can use the correlation identifier in any way you like. However, an intended use of this field is for applications to copy the message identifier of a request message into the *CorrelId* field of a reply message.

The group identifier is usually generated by the queue manager when the first message of a group is put onto a queue. The *MsgSeqNumber* field identifies the position of the message within the group and the *Offset* field identifies the segments within the message.

Where more than one message meets the combined selection criteria, the *MsgDeliverySequence* attribute of the queue determines whether messages are selected in FIFO (first in, first out) or priority order. When messages have equal priority, they are selected in FIFO order. For more information, see “The order in which messages are retrieved from a queue” on page 126.

For an example of an application that uses correlation identifiers, see “The Credit Check sample” on page 451.

Messages that fail to be delivered

When a queue manager is unable to put a message on a queue, you have various options. You can:

- Attempt to put the message on the queue again.
- Request that the message is returned to the sender.
- Put the message on the dead-letter queue.

See “Chapter 5. Handling program errors” on page 53 for more information.

Messages that are backed out

When processing messages from a queue under the control of a unit of work, the unit of work could consist of one or more messages. If a backout occurs, the messages which were retrieved from the queue are reinstated on the queue, and they can be processed again in another unit of work. If the processing of a particular message is causing the problem, the unit of work is backed out again. This could cause a processing loop. Messages which were put to a queue are removed from the queue.

An application can detect messages that are caught up in such a loop by testing the *BackoutCount* field of MQMD. The application can either correct the situation, or issue a warning to an operator.

In MQSeries for OS/390, to ensure that the back-out count survives restarts of the queue manager, set the *HardenGetBackout* attribute to MQQA_BACKOUT_HARDENED; otherwise, if the queue manager has to restart, it does not maintain an accurate back-out count for each message. Setting the attribute this way adds the penalty of extra processing.

Message backout

In MQSeries for AS/400, MQSeries for OS/2 Warp, MQSeries for Windows NT, MQSeries for Compaq (DIGITAL) OpenVMS, and MQSeries on UNIX systems, the back-out count always survives restarts of the queue manager. Any change to the *HardenGetBackout* attribute is ignored.

Note: In MQSeries for VSE/ESA, the BackoutCount field is reserved and so cannot be used as described here.

For more information on committing and backing out messages, see “Chapter 13. Committing and backing out units of work” on page 177.

Reply-to queue and queue manager

There are occasions when you may receive messages in response to a message you send:

- A reply message in response to a request message
- A report message about an unexpected event or expiry
- A report message about a COA (Confirmation Of Arrival) or a COD (Confirmation Of Delivery) event
- A report message about a PAN (Positive Action Notification) or a NAN (Negative Action Notification) event

Using the MQMD structure, specify the name of the queue to which you want reply and report messages sent, in the *ReplyToQ* field. Specify the name of the queue manager that owns the reply-to queue in the *ReplyToQMgr* field.

If you leave the *ReplyToQMgr* field blank, the queue manager sets the contents of the following fields in the message descriptor on the queue:

ReplyToQ

If *ReplyToQ* is a local definition of a remote queue, the *ReplyToQ* field is set to the name of the remote queue; otherwise this field is not changed.

ReplyToQMgr

If *ReplyToQ* is a local definition of a remote queue, the *ReplyToQMgr* field is set to the name of the queue manager that owns the remote queue; otherwise the *ReplyToQMgr* field is set to the name of the queue manager to which your application is connected.

Note: You can request that a queue manager makes more than one attempt to deliver a message, and you can request that the message is discarded if it fails. If the message, after failing to be delivered, is not to be discarded, the remote queue manager puts the message on its dead-letter (undelivered-message) queue (see “Using the dead-letter (undelivered-message) queue” on page 57).

Message context

Message context information allows the application that retrieves the message to find out about the originator of the message. The retrieving application may want to:

- Check that the sending application has the correct level of authority
- Perform some accounting function so that it can charge the sending application for any work it has to perform
- Keep an audit trail of all the messages it has worked with

When you use the MQPUT or MQPUT1 call to put a message on a queue, you can specify that the queue manager is to add some default context information to the message descriptor. Applications that have the appropriate level of authority can add extra context information. For more information on how to specify context information, see “Controlling context information” on page 112.

All context information is stored in the eight context fields of the message descriptor. The type of information falls into two categories: identity and origin context information.

Identity context

Identity context information identifies the user of the application that *first* put the message on a queue:

- The queue manager fills the *UserIdentifier* field with a name that identifies the user—the way that the queue manager can do this depends on the environment in which the application is running.
- The queue manager fills the *AccountingToken* field with a token or number that it determined from the application that put the message.
- Applications can use the *ApplIdentityData* field for any extra information that they want to include about the user (for example, an encrypted password).

Suitably authorized applications may set the above fields.

A Windows NT security identifier (SID) is stored in the *AccountingToken* field when a message is created under MQSeries for Windows NT. The SID can be used to supplement the *UserIdentifier* field and to establish the credentials of a user.

For information on how the queue manager fills the *UserIdentifier* and *AccountingToken* fields, see the descriptions of these fields in the *MQSeries Application Programming Reference* manual.

Applications that pass messages from one queue manager to another should also pass on the identity context information so that other applications know the identity of the originator of the message.

Origin context

Origin context information describes the application that put the message on the queue on which the message is *currently* stored. The message descriptor contains the following fields for origin context information:

PutApplType

The type of application that put the message (for example, a CICS transaction).

PutApplName

The name of the application that put the message (for example, the name of a job or transaction).

PutDate

The date on which the message was put on the queue.

PutTime

The time at which the message was put on the queue.

ApplOriginData

Any extra information that an application may want to include about the

Message context

origin of the message. For example, it could be set by suitably authorized applications to indicate whether the identity data is trusted.

Origin context information is usually supplied by the queue manager. Greenwich Mean Time (GMT) is used for the *PutDate* and *PutTime* fields. See the descriptions of these fields in the *MQSeries Application Programming Reference* manual.

Within MQSeries for OS/2 Warp only, the TZ environment variable is used to calculate the GMT *PutDate* and *PutTime* of a message.

An application with enough authority can provide its own context. This allows accounting information to be preserved when a single user has a different user ID on each of the systems that process a message they have originated.

Chapter 4. MQSeries objects

The MQSeries objects are:

- Queue managers
- Queue-sharing groups (MQSeries for OS/390 only), although these are not strictly objects.
- Queues
- Namelists (MQSeries Version 5 products only)
- Process definitions
- Channels
- Storage classes (MQSeries for OS/390 only)

Queue managers define the properties (known as attributes) of these objects. The values of these attributes affect the way in which these objects are processed by MQSeries. From your applications, you use the Message Queue Interface (MQI) to control these objects. Each object is identified by an *object descriptor* (MQOD) when addressed from a program.

When you use MQSeries commands to define, alter, or delete objects, for example, the queue manager checks that you have the required level of authority to perform these operations. Similarly, when an application uses the MQOPEN call to open an object, the queue manager checks that the application has the required level of authority before it allows access to that object. The checks are made on the name of the object being opened.

This chapter introduces MQSeries objects, under these headings:

- “Queue managers”
- “Queue-sharing groups” on page 40
- “Queues” on page 41
- “Namelists” on page 48
- “Process definitions” on page 48
- “Channels” on page 49
- “Storage classes” on page 49
- “Rules for naming MQSeries objects” on page 49

Queue managers

A *queue manager* supplies an application with MQSeries services. A program must have a connection to a queue manager before it can use the services of that queue manager. A program can make this connection explicitly (using the MQCONN or MQCONNX call), or the connection might be made implicitly (this depends on the platform and the environment in which the program is running).

Queues belong to queue managers, but programs can send messages to queues that belong to any queue manager.

Attributes of queue managers

Associated with each queue manager is a set of attributes (or properties) that define its characteristics. Some of the attributes of a queue manager are fixed when it is created; you can change others using the MQSeries commands. You can inquire about the values of *all* the attributes using the MQINQ call.

Queue managers

The *fixed* attributes include:

- The name of the queue manager
- The platform on which the queue manager runs (for example, AS/400)
- The level of system control commands that the queue manager supports
- The maximum priority that you can assign to messages processed by the queue manager
- The name of the queue to which programs can send MQSeries commands
- The identifier of the character set the queue manager uses for character strings when it processes MQI calls (this can be changed in OS/390 using the system parameters)
- The maximum length of messages the queue manager can process
- Whether the queue manager supports syncpointing when programs put and get messages

The *changeable* attributes include:

- A text description of the queue manager
- The time interval that the queue manager uses to restrict the number of trigger messages
- The name of the queue manager's dead-letter (undelivered-message)queue
- The name of the queue manager's default transmission queue
- The maximum number of open handles for any one connection
- The enabling and disabling of various categories of event reporting
- The maximum number of uncommitted messages within a unit of work

For a full description of all the attributes, see the *MQSeries Application Programming Reference* manual.

Queue managers and workload management

You can set up a cluster of queue managers that has more than one definition for the same queue (for example, the queue managers in the cluster could be clones of each other). Messages for a particular queue can be handled by any queue manager which hosts an instance of the queue. A workload-management algorithm decides which queue manager handles the message and so spreads the workload between your queue managers. See the *MQSeries Queue Manager Clusters* book for further information.

Queue-sharing groups

Queue-sharing groups are available on MQSeries for OS/390 only. They are not strictly objects, but are mentioned here for convenience.

Queue managers that can access the same set of shared queues form a group called a *queue-sharing group* (QSG), and they communicate with each other by means of a coupling facility (CF) that stores the shared queues. A *shared queue* is a type of local queue whose messages can be accessed by one or more queue managers that are in a queue-sharing group. (This is not the same as a queue being "shared" by more than one application, via the same queue manager.) See the *MQSeries for OS/390 Concepts and Planning Guide* for a full discussion of shared queues and queue-sharing groups.

Queues

An MQSeries *queue* is a named object on which applications can put messages, and from which applications can get messages. Messages are stored on a queue, so if the putting application is expecting a reply to its message, it is free to do other work while waiting for that reply. Applications access a queue by using the Message Queue Interface (MQI), described in “Chapter 6. Introducing the Message Queue Interface” on page 65.

Before a message can be put on a queue, the queue must have already been created. A queue is owned by a queue manager, and that queue manager can own many queues. However, each queue must have a name that is unique within that queue manager.

A queue is maintained through a queue manager. Queues are managed physically by their queue managers but this is transparent to an application program.

To create a queue you can use MQSeries commands (MQSC), PCF commands, or platform-specific interfaces such as the MQSeries for OS/390 operations and control panels.

On all platforms except MQSeries for VSE/ESA, you can create local queues for temporary jobs “dynamically” from your application. For example, you can create *reply-to* queues (which are not needed after an application ends). For more information, see “Dynamic queues” on page 45.

Before using a queue, you must open the queue, specifying what you want to do with it. For example, you can open a queue:

- For browsing messages only (not retrieving them)
- For retrieving messages (and either sharing the access with other programs, or with exclusive access)
- For putting messages on the queue
- For inquiring about the attributes of the queue
- For setting the attributes of the queue

For a complete list of the options you can specify when you open a queue, see the description of the MQOPEN call in the *MQSeries Application Programming Reference* manual.

Types of queue

The types of queue that MQSeries supports for applications to use are:

Local and remote queues

A queue is known to a program as *local* if it is owned by the queue manager to which the program is connected; the queue is known as *remote* if it is owned by a different queue manager. The important difference between these two types of queue is that you can get messages only from local queues. (You can put messages on both types of queue.)

The queue definition object, created when you define a local queue, will hold the definition information of the queue as well as the physical messages put on the queue. The queue definition object, created when you ‘define’ a remote queue, will only hold the information necessary for the local queue manager to be able to locate the queue to which you want your message to go. This object is known as the *local definition of a remote*

Queues

queue. All the attributes of the remote queue are held by the queue manager that owns it, because it is a local queue to that queue manager.

Shared queues

Shared queues are only available on MQSeries for OS/390.

A *shared queue* is a type of local queue whose messages can be accessed by one or more queue managers that are in a queue-sharing group. (This is not the same as a queue being "shared" by more than one application, via the same queue manager.) Shared queues are held by a coupling facility (CF), and are accessible by any queue manager in the queue-sharing group. Each shared queue in a queue-sharing group must have a name that is unique within that group. See the *MQSeries for OS/390 Concepts and Planning Guide* for a full discussion of shared queues and queue-sharing groups.

Alias queues

To your program, an *alias queue* appears to be a queue, but it is really an MQSeries object that you can use to access another queue. This means that more than one program can work with the same queue, accessing it using different names.

Model and dynamic queues

A model queue is a template of a queue definition used only when you want to create a dynamic local queue.

You can create a local queue dynamically from an MQSeries program, naming the model queue you wish to use as the template for the queue attributes. You may now, if you wish, change some attributes of the new queue. However, you cannot change the *DefinitionType*. If, for example, you require a permanent queue, you must select a model queue with the definition type set to permanent. Some conversational applications could make use of dynamic queues to hold replies to their queries because they probably would not need to maintain these queues after they have processed the replies.

Cluster queues

A cluster queue is a queue that is hosted by a cluster queue manager and made available to other queue managers in the cluster.

The cluster queue manager makes a local queue definition for the queue specifying the name of the cluster that the queue is to be available in. This definition has the effect of advertising the queue to the other queue managers in the cluster. The other queue managers in the cluster can put messages to a cluster queue without needing a corresponding remote-queue definition. A cluster queue can be advertised in more than one cluster. See "What is a cluster?" on page 5 and the *MQSeries Queue Manager Clusters* book for further information.

Types of local queue

Each queue manager can have some local queues that it uses for special purposes:

Transmission queues

A *transmission queue* is a local queue which holds messages destined for a remote queue. The messages are forwarded to their destination queue by MQSeries when a communication program and link are available.

Initiation queues

An *initiation queue* is a local queue on which the queue manager puts a

message for the purpose of automatically starting an application when certain conditions (such as more than 10 messages arriving, for example) are met on a local queue.

Dead-letter (undelivered-message) queue

The *dead-letter queue* is a local queue on which the queue manager and applications put messages they cannot deliver. You should plan to process any messages that arrive on this queue.

System command queue

The *system command queue* is a queue to which suitably authorized applications can send MQSeries commands.

System default queues

When you create a queue (other than a dynamic queue), MQSeries uses the queue definitions stored in the *system default queues*.

Channel queues

Channel queues are used for distributed queue management.

Event queues

Event queues hold event messages. These messages are reported by the queue manager or a channel.

These special queues are described in greater detail in the following sections.

Attributes of queues

Some of the attributes of a queue are specified when the queue is defined, and may not be changed afterwards (for example, the type of the queue). Other attributes of queues can be grouped into those that can be changed:

- By the queue manager during the processing of the queue (for example, the current depth of a queue)
- Only by commands (for example, the text description of the queue)
- By applications, using the MQSET call (for example, whether or not put operations are allowed on the queue)

You can find the values of all the attributes using the MQINQ call.

The attributes that are common to more than one type of queue are:

QName Name of the queue

QType Type of the queue

QDesc Text description of the queue

InhibitGet

Whether or not programs are allowed to get messages from the queue (although you can never get messages from remote queues)

InhibitPut

Whether or not programs are allowed to put messages on the queue

DefPriority

Default priority for messages put on the queue

DefPersistence

Default persistence for messages put on the queue

Scope (not supported on OS/390 or VSE/ESA)

Controls whether an entry for this queue also exists in a name service

Queues

For a full description of these attributes, see the *MQSeries Application Programming Reference* manual.

Remote queues

To a program, a queue is *remote* if it is owned by a different queue manager to the one to which the program is connected. Where a communication link has been established, it is possible for a program to send a message to a remote queue. A program can never get a message from a remote queue.

When opening a remote queue, to identify the queue you must specify either:

- The name of the local definition that defines the remote queue.
To create a local definition of a remote queue use the DEFINE QREMOTE command; in MQSeries for AS/400, alternatively use the CRTMQMQ command; in MQSeries for Tandem NonStop Kernel, you can use the MQM screen-based interface; in MQSeries for VSE/ESA, you can use the MQMT transaction.
From the viewpoint of an application, this is the same as opening a local queue. An application does not need to know if a queue is local or remote.
- The name of the remote queue manager and the name of the queue as it is known to that remote queue manager.

Local definitions of remote queues have three attributes in addition to the common attributes described in “Attributes of queues” on page 43. These are *RemoteQName* (the name that the queue’s owning queue manager knows it by), *RemoteQMGrName* (the name of the owning queue manager), and *XmitQName* (the name of the local transmission queue that is used when forwarding messages to other queue managers). For a fuller description of these attributes, see the *MQSeries Application Programming Reference* manual.

If you use the MQINQ call against the local definition of a remote queue, the queue manager returns the attributes of the local definition only, that is the remote queue name, the remote queue manager name and the transmission queue name, not the attributes of the matching local queue in the remote system.

See also “Transmission queues” on page 47.

Alias queues

An *alias queue* is an MQSeries object that you can use to access another queue. The queue resulting from the resolution of an alias name (known as the base queue) can be a local queue, the local definition of a remote queue, or a shared queue (a type of local queue only available on MQSeries for OS/390). It can also be either a predefined queue or a dynamic queue, as supported by the platform.

Note: An alias cannot resolve to another alias.

An example of the use of alias queues is for a system administrator to give different access authorities to the base queue name (that is, the queue to which the alias resolves) and to the alias queue name. This would mean that a program or user could be authorized to use the alias queue, but not the base queue.

Alternatively, authorization can be set to inhibit put operations for the alias name, but allow them for the base queue.

In some applications, the use of alias queues means that system administrators can easily change the definition of an alias queue object without having to get the application changed.

MQSeries makes authorization checks against the alias name when programs try to use that name. It does not check that the program is authorized to access the name to which the alias resolves. A program can therefore be authorized to access an alias queue name, but not the resolved queue name.

In addition to the general queue attributes described in “Attributes of queues” on page 43, alias queues have a *BaseQName* attribute. This is the name of the base queue to which the alias name resolves. For a fuller description of this attribute, see the *MQSeries Application Programming Reference* manual.

The *InhibitGet* and *InhibitPut* attributes (see “Attributes of queues” on page 43) of alias queues belong to the alias name. For example, if the alias-queue name ALIAS1 resolves to the base-queue name BASE, inhibitions on ALIAS1 affect ALIAS1 only and BASE is not inhibited. However, inhibitions on BASE also affect ALIAS1.

The *DefPriority* and *DefPersistence* attributes also belong to the alias name. So, for example, you can assign different default priorities to different aliases of the same base queue. Also, you can change these priorities without having to change the applications that use the aliases.

Model queues

A *model queue* is a template of a queue definition, that you use when creating a dynamic queue. You specify the name of a model queue in the *object descriptor* (MQOD) of your MQOPEN call. Using the attributes of the model queue, the queue manager dynamically creates a local queue for you.

You can specify a name (in full) for the dynamic queue, or the stem of a name (for example, ABC) and let the queue manager add a unique part to this, or you can let the queue manager assign a complete unique name for you. If the queue manager assigns the name, it puts it in the MQOD structure.

You can not issue an MQPUT1 call directly to a model queue, however, once a model queue has been opened, you can issue an MQPUT1 to the dynamic queue.

The attributes of a model queue are a subset of those of a local queue. For a fuller description, see the *MQSeries Application Programming Reference* manual.

Dynamic queues

On all platforms except for MQSeries for VSE/ESA, when an application program issues an MQOPEN call to open a model queue, the queue manager dynamically creates an instance of a local queue with the same attributes as the model queue. Depending on the value of the *DefinitionType* field of the model queue, the queue manager creates either a temporary or permanent dynamic queue (See “Creating dynamic queues” on page 104).

Properties of temporary dynamic queues

Temporary dynamic queues have the following properties:

- They cannot be shared queues, accessible from queue managers in a queue-sharing group (only available on MQSeries for OS/390).
- They hold nonpersistent messages only.

Queues

- They are non-recoverable.
- They are deleted when the queue manager is started.
- They are deleted when the application that issued the MQOPEN call which resulted in the creation of the queue closes the queue or terminates.
 - If there are any committed messages on the queue, they will be deleted.
 - If there are any uncommitted MQGET, MQPUT, or MQPUT1 calls outstanding against the queue at this time, the queue is marked as being logically deleted, and is only physically deleted (after these calls have been committed) as part of close processing, or when the application terminates.
 - If the queue happens to be in use at this time (by the creating, or another application), the queue is marked as being logically deleted, and is only physically deleted when closed by the last application using the queue.
 - Attempts to access a logically deleted queue (other than to close it) fail with reason code MQRC_Q_DELETED.
 - MQCO_NONE, MQCO_DELETE and MQCO_DELETE_PURGE are all treated as MQCO_NONE when specified on an MQCLOSE call for the corresponding MQOPEN call that created the queue.

Properties of permanent dynamic queues

Permanent dynamic queues have the following properties:

- They hold persistent or nonpersistent messages.
- They are recoverable in the event of system failures.
- They are deleted when an application (not necessarily the one that issued the MQOPEN call which resulted in the creation of the queue) successfully closes the queue using the MQCO_DELETE, or the MQCO_DELETE_PURGE option.
 - A close request with the MQCO_DELETE option fails if there are any messages (committed or uncommitted) still on the queue. A close request with the MQCO_DELETE_PURGE option succeeds even if there are committed messages on the queue (the messages being deleted as part of the close), but fails if there are any uncommitted MQGET, MQPUT, or MQPUT1 calls outstanding against the queue.
 - If the delete request is successful, but the queue happens to be in use (by the creating, or another application), the queue is marked as being logically deleted and is only physically deleted when closed by the last application using the queue.
- They can not be deleted by an application closing the queue, unless it was the application that issued the MQOPEN which created the queue. Authorization checks are performed against the user identifier (or alternate user identifier if MQOO_ALTERNATE_USER_AUTHORITY was specified) which was used to validate the corresponding MQOPEN call.
- They can be deleted in the same way as a normal queue.

Uses of dynamic queues

You can use dynamic queues for:

- Applications that do not require queues to be retained after the application has terminated.
- Applications that require replies to messages to be processed by another application can dynamically create a reply-to queue by opening a model queue. For example, a client application could:
 1. Create a dynamic queue.
 2. Supply its name in the *ReplyToQ* field of the message descriptor structure of the request message.

3. Place the request on a queue being processed by a server.

The server could then place the reply message on the reply-to queue. Finally, the client could process the reply, and close the reply-to queue with the delete option.

Recommendations for uses of dynamic queues

You should consider the following points when using dynamic queues:

- In a client-server model, each client should create and use its own dynamic reply-to queue. If a dynamic reply-to queue is shared between more than one client, the deletion of the reply-to queue may be delayed because there is uncommitted activity outstanding against the queue, or because the queue is in use by another client. Additionally, the queue might be marked as being logically deleted, and hence inaccessible for subsequent API requests (other than MQCLOSE).
- If your application environment requires that dynamic queues must be shared between applications, you should ensure that the queue is only closed (with the delete option) when all activity against the queue has been committed. This should be by the last user preferably. This ensures that deletion of the queue is not delayed, and should minimize the period that the queue is inaccessible because it has been marked as being logically deleted.

Transmission queues

When an application sends a message to a remote queue, the local queue manager stores the message in a special local queue, called a *transmission queue*.

A *message channel agent* (channel program), or *intra-group queuing agent* when using intra-group queuing on MQSeries for OS/390, will be associated with the transmission queue and the remote queue manager, and it is this that deals with the transmitting of the message. When the message has been transmitted, it is deleted from the transmission queue.

The message may have to pass through many queue managers (or *nodes*) on its journey to its final destination. There must be a transmission queue defined at each queue manager along the route, each holding messages waiting to be transmitted to the next node. (A shared transmission queue is used when using intra-group queuing on MQSeries for OS/390.) There can be several transmission queues defined at a particular queue manager. A given transmission queue holds messages whose *next* destination is the same queue manager, although the messages may have different eventual destinations. There may also be several transmission queues for the same remote queue manager, with each one being used for a different type of service, for example.

Transmission queues can be used to trigger a message channel agent to send messages onward. For information about this, see “Chapter 14. Starting MQSeries applications using triggers” on page 191. These attributes are defined in the transmission queue definition (for triggered channels) or the process definition object (see “Process definitions” on page 48).

Initiation queues

An *initiation queue* is a local queue on which the queue manager puts a trigger message when a trigger event occurs on an application queue. A trigger event is an event (for example, more than 10 messages arriving) that an application designer intends the queue manager to use as a cue, or trigger, to start a program that will process the queue. For more information on how triggering works, see “Chapter 14. Starting MQSeries applications using triggers” on page 191.

Queues

Dead-letter (undelivered-message) queues

A *dead-letter (undelivered-message) queue* is a local queue on which the queue manager puts messages it cannot deliver.

When the queue manager puts a message on the dead-letter queue, it adds a header to the message. This includes such information as the intended destination of the original message, the reason the queue manager put the message on the dead-letter queue, and the date and time it did this.

Applications can also use the queue for messages they cannot deliver. For more information, see “Using the dead-letter (undelivered-message) queue” on page 57.

System command queues

System command queues are not supported on MQSeries for VSE/ESA.

These queues receive the PCF, MQSC, and CL commands, as supported on your platform, in readiness for the queue manager to action them. In MQSeries for OS/390 the queue is known as the **SYSTEM.COMMAND.INPUT.QUEUE** and accepts MQSC commands. On other platforms it is known as the **SYSTEM.ADMIN.COMMAND.QUEUE** and the commands accepted vary by platform. See the *MQSeries Programmable System Management* book for details.

System default queues

The *system default queues* contain the initial definitions of the queues for your system. When you create a new queue, the queue manager copies the definition from the appropriate system default queue.

Namelist

Namelist are supported on MQSeries Version 5 products only.

A *namelist* is an MQSeries object that contains a list of cluster names or queue names. In a cluster, it can be used to identify a list of clusters for which the queue manager holds the repositories.

You can define and modify namelists using only the operations and control panels of MQSeries for OS/390 or the MQSC commands of MQSeries Version 5 products.

Programs can use the MQI to find out which queues are included in these namelists. The organization of the namelists is the responsibility of the application designer and system administrator.

For a full description of the attributes of namelists, see the *MQSeries Application Programming Reference* manual.

Process definitions

Note: Process definition objects are not supported on VSE/ESA.

To allow an application to be started without the need for operator intervention (described in “Chapter 14. Starting MQSeries applications using triggers” on page 191), the attributes of the application must be known to the queue manager. These attributes are defined in a *process definition object*.

The *ProcessName* attribute is fixed when the object is created; you can change the others using the MQSeries commands or the MQSeries for OS/390 operations and control panels. You can inquire about the values of *all* the attributes using the MQINQ call.

For a full description of the attributes of process definitions, see the *MQSeries Application Programming Reference* manual.

Channels

A *channel* is a communication link used by distributed queue managers. There are two categories of channel in MQSeries:

- *Message channels*, which are unidirectional, and transfer messages from one queue manager to another.
- *MQI channels*, which are bidirectional, and transfer MQI calls from an MQSeries client to a queue manager, and responses from a queue manager to an MQSeries client.

These need to be considered when designing your application, but a program will be unaware of MQSeries channel objects. For more information, see the *MQSeries Intercommunication* and *MQSeries Clients* books.

Storage classes

Storage classes are supported on MQSeries for OS/390 only.

A *storage class* maps one or more queues to a page set. This means that messages for that queue are stored (subject to buffering) on that page set.

For further information about storage classes, see the *MQSeries for OS/390 Concepts and Planning Guide*.

Rules for naming MQSeries objects

An MQSeries queue, process definition, namelist, and channel can all have the same name. However, an MQSeries object cannot have the same name as any other object of the same type. Names in MQSeries are case sensitive.

The character set that can be used for naming all MQSeries objects is as follows:

- Uppercase A–Z
- Lowercase a–z (but there are restrictions on the use of lowercase letters for OS/390 console support)
On systems using EBCDIC Katakana you cannot use lowercase characters.
- Numerics 0–9
- Period (.)
- Forward slash (/)
- Underscore (_)
- Percent sign (%)

Notes:

1. Leading or embedded blanks are not allowed.

Naming objects

2. You should also avoid using names with leading or trailing underscores, because they cannot be handled by the MQSeries for OS/390 operations and control panels.
3. Any name that is less than the full field length can be padded to the right with blanks. All short names that are returned by the queue manager are always padded to the right with blanks.
4. Any structure to the names (for example, the use of the period or underscore) is not significant to the queue manager.
5. On AS/400 systems lowercase a-z, forward slash (/), and percent (%) are special characters. If you use any of these characters in a name, the name must be enclosed in quotation marks. Lowercase a-z characters are changed to uppercase if the name is not enclosed in quotation marks.
6. The qshell environment is case sensitive.

Queue names

The name of a queue has two parts:

- The name of a queue manager
- The local name of the queue as it is known to that queue manager

Each part of the queue name is 48 characters long.

To refer to a local queue, you can omit the name of the queue manager (by replacing it with blank characters or using a leading null character). However, all queue names returned to a program by MQSeries contain the name of the queue manager.

Note that a shared queue, accessible to any queue manager in its queue-sharing group, cannot have the same name as any non-shared local queue in the same queue-sharing group. This restriction avoids the possibility of an application mistakenly opening a shared queue when it intended to open a local queue, or *vice versa*. Shared queues and queue-sharing groups are only available on MQSeries for OS/390.

To refer to a remote queue, a program must include the name of the queue manager in the full queue name, or there must be a local definition of the remote queue.

Note that when an application uses a queue name, that name can be either the name of a local queue (or an alias to one) or the name of a local definition of a remote queue, but the application does not need to know which, unless it needs to get a message from the queue (when the queue must be local). When the application opens the queue object, the MQOPEN call performs a name resolution function to determine on which queue to perform subsequent operations. The significance of this is that the application has no built-in dependency on particular queues being defined at particular locations in a network of queue managers. Therefore, if a system administrator relocates queues in the network, and changes their definitions, the applications that use those queues do not need to be changed.

Process definition and namelist names

Process definitions and namelists can have names up to 48 characters long.

Channel names

Channels can have names up to 20 characters long. See the *MQSeries Intercommunication* book for further information on channels.

Reserved object names

Names that start with SYSTEM. are reserved for objects defined by the queue manager.

Chapter 5. Handling program errors

Your application may encounter errors associated with its MQI calls either when it makes a call or when its message is delivered to its final destination:

- Whenever possible, the queue manager returns any errors as soon as an MQI call is made. These are *locally determined errors*.
- When sending messages to a remote queue, errors may not be apparent when the MQI call is made. In this case, the queue manager that identifies the errors reports them by sending another message to the originating program. These are *remotely determined errors*.

This chapter gives advice on how to handle both types of error, under these headings:

- “Locally determined errors”
- “Using report messages for problem determination” on page 55
- “Remotely determined errors” on page 56

Locally determined errors

The three most common causes of errors that the queue manager can report immediately are:

- Failure of an MQI call; for example, because a queue is full
- An interruption to the running of some part of the system on which your application is dependent; for example, the queue manager
- Messages containing data that cannot be processed successfully

Failure of an MQI call

The queue manager can report immediately any errors in the coding of an MQI call. It does this using a set of predefined return codes. These are divided into completion codes and reason codes.

To show whether or not a call is successful, the queue manager returns a *completion code* when the call completes. There are three completion codes, indicating success, partial completion, and failure of the call. The queue manager also returns a *reason code* which indicates the reason for the partial completion or the failure of the call.

The completion and reason codes for each call are listed with the description of that call in the *MQSeries Application Programming Reference* manual. You will also find further information (including some ideas for corrective action) for each completion and reason code, in the *MQSeries Application Programming Reference* manual. You should design your programs to handle all the return codes that could arise from each call.

System interruptions

Your application may be unaware of any interruption if the queue manager to which it is connected has to recover from a system failure. However, you must design your application to ensure that your data is not lost if such an interruption occurs.

Locally determined errors

The methods you can use to make sure that your data remains consistent depends on the platform on which your queue manager is running:

OS/390

In the CICS and IMS environments, you can make MQPUT and MQGET calls within units of work that are managed by CICS or IMS. In the batch environment, you can make MQPUT and MQGET calls in the same way, but you must declare syncpoints by using the MQSeries for OS/390 MQCMIT and MQBACK calls (see “Chapter 13. Committing and backing out units of work” on page 177), or you can use the OS/390 Transaction Management and Recoverable Resource Manager Services (RRS) to provide two-phase syncpoint support. RRS allows you to update both MQSeries and other RRS-enabled product resources, such as DB2 stored procedure resources, within a single logical unit of work. For information on RRS syncpoint support see “Transaction management and recoverable resource manager services” on page 181.

AS/400

You can make your MQPUT and MQGET calls within global units of work that are managed by OS/400 commitment control. You can declare syncpoints by using the native OS/400 COMMIT and ROLLBACK commands or the language-specific commands. Local units of work are managed by MQSeries via the MQCMIT and MQBACK calls.

Digital OpenVMS, DOS, OS/2, UNIX systems, Windows NT, and Windows 3.1

In these environments, you can make your MQPUT and MQGET calls in the normal way, but you must declare syncpoints by using the MQCMIT and MQBACK calls (see “Chapter 13. Committing and backing out units of work” on page 177). In the CICS environment, MQCMIT and MQBACK commands are disabled as you can make your MQPUT and MQGET calls within units of work that are managed by CICS.

Tandem NSK

You can make your MQPUT and MQGET calls within units of work that are managed by Tandem’s TM/MP product.

VSE/ESA

CICS controls the unit of work in the VSE/ESA environment. If the system fails and is restarted, the logical unit of work rollback occurs automatically.

You should use persistent messages for carrying all data you cannot afford to lose. Persistent messages are reinstated on queues if the queue manager has to recover from a failure. With MQSeries on UNIX systems, MQSeries for OS/2 Warp, and MQSeries for Windows NT, note that an MQGET or MQPUT call within your application will fail at the point of filling up all the log files, with the message MQRC_RESOURCE_PROBLEM. For more information on log files, see the *MQSeries System Administration Guide* for MQSeries for AIX, HP-UX, OS/2, Sun Solaris, and Windows NT; for OS/390 see the *MQSeries for OS/390 Concepts and Planning Guide*; for other platforms, see the appropriate *System Management Guide*.

If the queue manager is stopped by an operator while an application is running, the quiesce option is normally used. The queue manager enters a quiescing state in which applications can continue to do work, but they should terminate as soon as it is convenient. Small, quick applications can probably ignore the quiescing state and continue until they terminate as normal. Longer running applications, or ones that wait for messages to arrive, should use the *fail if quiescing* option when they use the MQCONN, MQCONNX, MQPUT, MQPUT1, and MQGET calls. These options mean that the calls fail when the queue manager quiesces, but the

application may still have time to terminate cleanly by issuing calls that ignore the quiescing state. Such applications could also commit, or back out, changes they have made, and then terminate.

If the queue manager is forced to stop (that is, stop without quiescing), applications will receive the `MQRC_CONNECTION_BROKEN` reason code when they make MQI calls. At this point you must exit the application or, alternatively, on MQSeries for AS/400, MQSeries on UNIX systems, MQSeries for OS/2 Warp, and MQSeries for Windows NT, you can issue an MQDISC call.

Messages containing incorrect data

When you use units of work in your application, if a program cannot successfully process a message that it retrieves from a queue, the MQGET call is backed out. The queue manager maintains a count (in the *BackoutCount* field of the message descriptor) of the number of times this happens. It maintains this count in the descriptor of each message that is affected. This count can provide valuable information about the efficiency of an application. Messages whose backout counts are increasing over time are being repeatedly rejected—you should design your application so that it analyzes the reasons for this and handles such messages accordingly.

In MQSeries for OS/390, to make the backout count survive restarts of the queue manager, set the *HardenGetBackout* attribute to `MQQA_BACKOUT_HARDENED`; otherwise, if the queue manager has to restart, it does not maintain an accurate backout count for each message. Setting the attribute this way adds the penalty of extra processing.

In MQSeries for AS/400, MQSeries for OS/2 Warp, MQSeries for Windows NT, and MQSeries on UNIX systems, the backout count always survives restarts of the queue manager.

Also, in MQSeries for OS/390, when you remove messages from a queue within a unit of work, you can mark one message so that it is *not* made available again if the unit of work is backed out *by the application*. The marked message is treated as if it has been retrieved under a new unit of work. You mark the message that is to skip backout using the `MQGMO_MARK_SKIP_BACKOUT` option (in the `MQGMO` structure) when you use the MQGET call. See “Skipping backout” on page 144 for more information about this technique.

Note: In MQSeries for VSE/ESA, *BackoutCount* is a reserved field. It cannot be used as described in this section.

Using report messages for problem determination

The remote queue manager cannot report errors such as failing to put a message on a queue when you make your MQI call, but it can send you a report message to say how it has processed your message.

Within your application you can create (MQPUT) report messages as well as select the option to receive them (in which case they will be sent by either another application or by a queue manager).

Creating report messages

Report messages provide a mechanism for an application to inform another application that it is unable to deal with the message that was sent. However, the

Report messages for error handling

Report field must initially be analyzed to determine whether or not the application that sent the message is interested in being informed of any problems. Having determined that a report message is required, you have to decide:

- Whether you want to include all the original message (not an option on OS/390), just the first 100 bytes of data, or none of the original message.
- What to do with the original message. You can discard it or let it go to the dead-letter queue.
- Whether the content of the *MsgId* and *CorrelId* fields are needed as well.

Use the *Feedback* field to indicate the reason for the report message being generated. Put your report messages on an application's reply-to queue. Refer to the *MQSeries Application Programming Reference* manual for further information.

Requesting and receiving (MQGET) report messages

When you send a message to another application, you will not be informed of any problems unless you complete the *Report* field to indicate the feedback you require. The options available to you are in the *MQSeries Application Programming Reference* manual.

Queue managers always put report messages on an application's reply-to queue and it is recommended that your own applications do the same. When you use the report message facility you must specify the name of your reply-to queue in the message descriptor of your message; otherwise, the MQPUT call fails.

Your application should contain procedures that monitor your reply-to queue and process any messages that arrive on it. Remember that a report message can contain all the original message, the first 100 bytes of the original message, or none of the original message.

The queue manager sets the *Feedback* field of the report message to indicate the reason for the error; for example, the target queue does not exist. Your programs should do the same.

For more information on report messages, see "Report messages" on page 25.

Remotely determined errors

When you send messages to a remote queue, even when the local queue manager has processed your MQI call without finding an error, other factors can influence how your message is handled by a remote queue manager. For example, the queue you are targeting may be full, or may not even exist. If your message has to be handled by other intermediate queue managers on the route to the target queue, any of these could find an error.

Problems delivering a message

When an MQPUT call fails, you have the choice of attempting to put the message on the queue again, returning it to the sender, or putting it on the dead-letter queue.

Each option has its own merits, but you may not want to retry putting a message if the reason that the MQPUT failed was because the destination queue was full. In this instance, putting it on the dead-letter queue allows you to deliver it to the correct destination queue later on.

Retry message delivery

Before the message is put on a dead-letter queue, a remote queue manager attempts to put the message on the queue again if the attributes *MsgRetryCount* and *MsgRetryInterval* have been set for the channel, or if there is a retry exit program for it to use (the name of which is held in the channel attribute *MsgRetryExitId* field).

If the *MsgRetryExitId* field is blank, the values in the attributes *MsgRetryCount* and *MsgRetryInterval* are used.

If the *MsgRetryExitId* field is not blank, the exit program of this name runs. For more information on using your own exit programs, see the *MQSeries Intercommunication* book.

Return message to sender

You return a message to the sender by requesting a report message to be generated to include all of the original message. See “Report messages” on page 25 for details on report message options.

Using the dead-letter (undelivered-message) queue

When a queue manager cannot deliver a message, it attempts to put the message on its dead-letter queue. This queue should be defined when the queue manager is installed.

Your programs can use the dead-letter queue in the same way that the queue manager uses it. You can find the name of the dead-letter queue by opening the queue manager object (using the MQCONN or MQCONNX call) and inquiring about the *DeadLetterQName* attribute (using the MQINQ call).

When the queue manager puts a message on this queue, it adds a header to the message, the format of which is described by the dead-letter header (MQDLH) structure, in the *MQSeries Application Programming Reference* manual. This header includes the name of the target queue and the reason the message was put on the dead-letter queue. It must be removed and the problem must be resolved before the message is put on the intended queue. Also, the queue manager changes the *Format* field of the message descriptor (MQMD) to indicate that the message contains an MQDLH structure.

MQDLH structure

You are recommended to add an MQDLH structure to all messages that you put on the dead-letter queue; however, if you intend to use the dead-letter handler provided by certain MQSeries products, you *must* add an MQDLH structure to your messages.

The addition of the header to a message may make the message too long for the dead-letter queue, so you should always make sure that your messages are shorter than the maximum size allowed for the dead-letter queue, by at least the value of the MQ_MSG_HEADER_LENGTH constant. The maximum size of messages allowed on a queue is determined by the value of the *MaxMsgLength* attribute of the queue. For the dead-letter queue, you should make sure that this attribute is set to the maximum allowed by the queue manager. If your application cannot deliver a message, and the message is too long to be put on the dead-letter queue, follow the advice given in the description of the MQDLH structure.

Remotely determined errors

You need to ensure that the dead-letter queue is monitored, and that any messages arriving on it get processed. A dead-letter queue handler is provided by MQSeries on all platforms except VSE/ESA. It runs as a batch utility and can be used to perform various actions on selected messages on the dead-letter queue. If you have a queue manager on one of the platforms that does not provide a dead-letter queue handler, you will need to provide your own. The program could be triggered, or run at regular intervals. For further details, see the *MQSeries System Administration Guide* for MQSeries for AIX, HP-UX, OS/2, Sun Solaris, and Windows NT; for MQSeries for OS/390 see the *MQSeries for OS/390 System Administration Guide*; for other platforms, see the appropriate *System Management Guide*.

If data conversion is necessary, the queue manager converts the header information when you use the MQGMO_CONVERT option on the MQGET call. If the process putting the message is an MCA, the header is followed by all the text of the original message.

You should be aware that messages put on the dead-letter queue may be truncated if they are too long for this queue. A possible indication of this situation is the messages on the dead-letter queue being the same length as the value of the *MaxMsgLength* attribute of the queue.

Dead-letter queue processing

The rest of this chapter contains general-use programming interface information.

Dead-letter queue processing is dependent on local system requirements, but you should consider the following when you draw up the specification:

- The message can be identified as having a dead-letter queue header because the value of the format field in the MQMD, is MQFMT_DEAD_LETTER_HEADER.
- In MQSeries for OS/390 using CICS, if an MCA puts this message to the dead-letter queue, the *PutApplType* field is MQAT_CICS, and the *PutApplName* field is the *ApplId* of the CICS system followed by the transaction name of the MCA.
- The reason for the message to be routed to the dead-letter queue is contained in the *Reason* field of the dead-letter queue header.
- The dead-letter queue header contains details of the destination queue name and queue manager name.
- The dead-letter queue header contains fields that have to be reinstated in the message descriptor before the message is put to the destination queue. These are:
 1. *Encoding*
 2. *CodedCharSetId*
 3. *Format*
- The message descriptor is the same as PUT by the original application, except for the three fields shown above.

Your dead-letter queue application should do one or more of the following:

- Examine the *Reason* field. A message may have been put by an MCA for the following reasons:
 - The message was longer than the maximum message size for the channel
The reason will be MQRC_MSG_TOO_BIG_FOR_CHANNEL (or MQRC_MSG_TOO_BIG_FOR_Q_MGR if you are using CICS for distributed queuing on MQSeries for OS/390)
 - The message could not be put to its destination queue

Remotely determined errors

The reason will be any MQRC_* reason code that can be returned by an MQPUT operation

- A user exit has requested this action

The reason code will be that supplied by the user exit, or the default MQRC_SUPPRESSED_BY_EXIT

- Try to forward the message to its intended destination, where this is possible.
- Retain the message for a certain length of time before discarding when the reason for the diversion is determined, but not immediately correctable.
- Give instructions to administrators for the correction of problems where these have been determined.
- Discard messages that are corrupted or otherwise not processible.

There are two ways that you deal with the messages you have recovered from the dead-letter queue:

1. If the message is for a local queue, you should:
 - Carry out any code translations required to extract the application data
 - Carry out code conversions on that data if this is a local function
 - Put the resulting message on the local queue with all the detail of the message descriptor restored
2. If the message is for a remote queue, put the message on the queue.

For information on how undelivered messages are handled in a distributed queuing environment, see the *MQSeries Intercommunication* book.

Part 2. Writing an MQSeries application

Chapter 6. Introducing the Message Queue

Interface	65	Coding in RPG	82
What is in the MQI?	65	Coding in PL/I	83
Calls.	66	Structures	83
Syncpoint calls	67	Named constants	83
MQSeries for OS/390 calls	67	Coding in TAL	83
OS/400 calls	67	OS/390 batch considerations	84
MQSeries for Tandem NonStop Kernel calls	68	UNIX signal handling on MQSeries Version 5 products	84
MQSeries for VSE/ESA calls.	68	Unthreaded applications	86
MQSeries calls on other platforms.	68	Threaded applications	86
Data conversion	68	Synchronous signals	86
Structures	68	Asynchronous signals	86
Elementary data types.	68	MQSeries use of SIGALRM	87
MQSeries data definitions	69	Threaded client applications - additional considerations	87
MQSeries stub programs and library files	69	Fastpath (trusted) applications	87
MQSeries for OS/390	70	MQI function calls within signal handlers	87
MQSeries for AS/400	70	Signals during MQI calls	87
MQSeries for OS/2 Warp.	71	User exits and installable services	88
MQSeries for Windows	71		
MQSeries for Windows NT	71	Chapter 7. Connecting and disconnecting a queue manager	89
MQSeries for AIX	72	Connecting to a queue manager using the MQCONN call	90
MQSeries for AT&T GIS UNIX	72	Scope of MQCONN	91
MQSeries for Compaq (DIGITAL) OpenVMS	72	Connecting to a queue manager using the MQCONN call.	92
MQSeries for Compaq Tru64 UNIX	72	Restrictions	93
MQSeries for HP-UX	73	Environment variable	94
MQSeries for SINIX and DC/OSx	73	Disconnecting programs from a queue manager using MQDISC	95
DOS and Windows 3.1 clients	74	Authority checking.	96
MQSeries for Sun Solaris	74		
MQSeries for VSE/ESA	74	Chapter 8. Opening and closing objects	97
MQSeries for Tandem NonStop Kernel	74	Opening objects using the MQOPEN call	98
Parameters common to all the calls	75	Scope of an object handle.	98
Using connection and object handles	75	Identifying objects (the MQOD structure)	99
Understanding return codes	75	Name resolution.	99
Specifying buffers	76	Using the options of the MQOPEN call.	102
Programming language considerations	76	MQOPEN option for cluster queue	102
Coding in C	77	MQOPEN option for putting messages	102
Parameters of the MQI calls	77	MQOPEN option for browsing messages	102
Parameters with undefined data type.	77	MQOPEN options for removing messages	102
Data types.	77	MQOPEN options for setting and inquiring about attributes.	103
Manipulating binary strings	77	MQOPEN options relating to message context	103
Manipulating character strings	78	MQOPEN option for alternate user authority	104
Initial values for structures	78	MQOPEN option for queue manager quiescing.	104
Initial values for dynamic structures	78	Creating dynamic queues	104
Use from C++	79	Opening remote queues	105
Coding in COBOL	79	Closing objects using the MQCLOSE call	105
Named constants	79		
Coding in System/390 [®] assembler language	80	Chapter 9. Putting messages on a queue	107
Names	80		
Using the MQI calls	80		
Declaring constants.	80		
Specifying the name of a structure.	81		
Specifying the form of a structure	81		
Controlling the listing	81		
Specifying initial values for fields	81		
Writing reenterable programs	82		
Using CEDF	82		

Putting messages on a local queue using the MQPUT call	107
Specifying handles	108
Defining messages using the MQMD structure	108
Specifying options using the MQPMO structure	108
The data in your message	111
Putting messages on a remote queue	112
Controlling context information	112
Passing identity context	112
Passing all context	113
Setting identity context	113
Setting all context	113
Putting one message on a queue using the MQPUT1 call	113
Distribution lists	115
Opening distribution lists	116
Using the MQOD structure	116
Using the MQOR structure	116
Using the MQRR structure	118
Using the MQOPEN options	118
Putting messages to a distribution list	118
Using the MQPMR structure	119
Using MQPUT1	120
Some cases where the put calls fail	120
Chapter 10. Getting messages from a queue	121
Getting messages from a queue using the MQGET call	121
Specifying connection handles	122
Describing messages using the MQMD structure and the MQGET call	122
Specifying MQGET options using the MQGMO structure	122
Specifying the size of the buffer area	125
The order in which messages are retrieved from a queue	126
Priority	126
Logical and physical ordering	126
Grouping logical messages	129
Putting and getting a group that spans units of work	130
Getting a particular message	133
Type of index	134
Handling messages greater than 4 MB long	135
Increasing the maximum message length	135
Message segmentation	136
Segmentation and reassembly by queue manager	137
Application segmentation	137
Application segmentation of logical messages	138
Putting and getting a segmented message that spans units of work	139
Reference messages	139
Using the MQRMH and MQMD structures	140
Waiting for messages	141
Signaling	142
To set a signal	142
When the message arrives	143
Skipping backout	144
Application data conversion	147
Conversion of EBCDIC newline characters	148

Browsing messages on a queue	149
The browse cursor	149
Queues in FIFO (first in, first out) sequence	149
Queues in priority sequence	150
Uncommitted messages	150
Change to queue sequence	150
Browsing messages when message length unknown	150
Removing a message you have browsed	151
Browsing messages in logical order	151
Browsing messages in groups	152
Browsing and retrieving destructively	153
Some cases where the MQGET call fails	154
Chapter 11. Writing data-conversion exits	155
Invoking the data-conversion exit	156
Data conversion on OS/390	156
Writing a data-conversion exit program	157
Skeleton source file	157
Convert characters call	158
Utility for creating conversion-exit code	158
Invoking the CSQUCVX utility on OS/390	159
Data definition statements	159
Error messages in OS/2, Windows NT, and UNIX systems	159
Valid syntax	159
Example of valid syntax for the input data set	160
Writing a data-conversion exit program for MQSeries for AS/400	161
Writing a data-conversion exit for MQSeries for OS/2 Warp	162
Writing a data-conversion exit program for MQSeries for OS/390	164
Writing a data-conversion exit for MQSeries for Tandem NonStop Kernel	165
Reusing data-conversion exit programs	165
Writing a data-conversion exit for MQSeries on UNIX systems and Compaq (DIGITAL) OpenVMS	166
UNIX environment	167
Non-threaded environment	167
Threaded environment	167
Compiling data-conversion exits on Digital OpenVMS	168
Compiling data-conversion exits on UNIX	168
On AIX 4.2	168
On AIX 4.3	169
On AT&T GIS UNIX	169
On Compaq Tru64 UNIX Version 4.0	169
On Compaq Tru64 UNIX Version 5.0	169
On HP-UX Version 10.20	169
On HP-UX Version 11.00	170
On SINIX	170
On DC/OSx	170
On Sun Solaris	170
Writing a data-conversion exit for MQSeries for Windows NT	171
Chapter 12. Inquiring about and setting object attributes	173
Inquiring about the attributes of an object	174

Some cases where the MQINQ call fails	175
Setting queue attributes	175

Chapter 13. Committing and backing out units of work 177

Syncpoint considerations in MQSeries applications	178
Syncpoints in MQSeries for OS/390 applications	179
Syncpoints in CICS Transaction Server for OS/390 and CICS for MVS/ESA applications.	179
Syncpoints in IMS applications	180
Syncpoints in OS/390 batch applications	180
Committing changes using the MQCMIT call	180
Backing out changes using the MQBACK call	181
Transaction management and recoverable resource manager services	181
Syncpoints in CICS for AS/400 applications	182
Syncpoints in MQSeries for OS/2 Warp, MQSeries for Windows NT, MQSeries for AS/400, and MQSeries on UNIX systems	182
Local units of work	183
Global units of work	183
Internal syncpoint coordination	183
External syncpoint coordination	184
Interfaces to external syncpoint managers	185
Interfaces to the AS/400 external syncpoint manager	187
Syncpoints in MQSeries for Tandem NonStop Kernel applications	188
General XA support	190

Chapter 14. Starting MQSeries applications using triggers. 191

What is triggering?	191
Prerequisites for triggering	196
Conditions for a trigger event	198
Controlling trigger events	202
Example of the use of trigger type EVERY.	203
Example of the use of trigger type FIRST	203
Example of the use of trigger type DEPTH	203
Special case of trigger type FIRST	203
Designing an application that uses triggered queues	204
Trigger messages and units of work	204
Getting messages from a triggered queue	205
Trigger monitors	206
MQSeries for OS/390 trigger monitors	206
MQSeries for OS/2 Warp, Digital OpenVMS, Tandem NSK, UNIX systems, AS/400, and Windows NT trigger monitors.	206
For CICS.	207
MQSeries for AS/400 trigger monitors	208
Properties of trigger messages	209
Persistence and priority of trigger messages	209
Queue manager restart and trigger messages	209
Trigger messages and changes to object attributes	209
Format of trigger messages	209
When triggering does not work	210
How CKTI detects errors	211
How CSQQTRMN detects errors	211
How RUNMQTRM detects errors	211

Chapter 15. Using and writing applications on MQSeries for OS/390 213

Environment-dependent MQSeries for OS/390 functions	213
Program debugging facilities	214
Syncpoint support.	214
Recovery support	215
The MQSeries for OS/390 interface with the application environment.	215
The batch adapter	215
RRS batch adapter.	215
Migration	216
The CICS adapter	216
CICS adapter performance considerations	216
Adapter trace points	218
Abends	218
Using the CICS Execution Diagnostic Facility	219
The IMS adapter	219
Writing OS/390 OpenEdition applications.	220
The API-crossing exit for OS/390.	221
Using the API-crossing exit.	221
Defining the exit program	221
How the exit is invoked.	221
Communicating with the exit program	222
Writing your own exit program	222
Usage notes	223
The sample API-crossing exit program, CSQCAPX	224
Design of the sample exit program	224
Preparing and using the API-crossing exit	224
MQSeries Workflow	225
Application programming with shared queues	226
Serializing your applications	226
Applications that are not suitable for use with shared queues	227
Deciding whether to share non-application queues	228
Migrating your existing applications to use shared queues	228

Chapter 16. Using and writing MQSeries-CICS bridge applications for OS/390 231

Using CICS DPL programs with the bridge	232
CICS DPL bridge message structure	232
Application programming for the CICS DPL bridge.	233
Managing units of work for the CICS DPL bridge.	234
Using CICS transactions with the bridge	234
Using CICS bridge vectors	235
CICS 3270 bridge message structure	235
Inbound messages.	235
Outbound messages	236
Application programming for the CICS 3270 bridge.	237
Example: Invoking CEMT I TASK from an application	238
Managing units of work for the CICS 3270 bridge.	239
Writing applications using CICS Basic Mapping Support	239

	Interpreting outbound SEND MAP and	
	RECEIVE MAP vectors	241
	Transactions with start data	249
	Programming for the distributed environment	249
	CICS transactions in the distributed	
	environment.	249
	CICS DPL programs in the distributed	
	environment.	250
	From 3270 legacy to 3270 bridge - an example	250
	Exact emulation - no optimization	251
	Improved emulation, with optimization	252
	Setting fields in the MQMD and MQCIH structures	253
	Setting the MQMD fields	253
	Setting the MQCIH fields	253
	Setting the open options and put message options	
	for the bridge request queue	256
	Managing MsgId and CorrelId in a unit of work	256
	Error handling by the CICS bridge	258
	Debugging CICS bridge applications	260
	Application data structure terminology.	262

Coding in LotusScript	281
Coding in ActiveX.	281

Chapter 17. IMS and IMS Bridge applications on

MQSeries for OS/390	265	
Writing IMS applications using MQSeries	265	
Syncpoints in IMS applications	265	
MQI calls in IMS applications	265	
Server applications	266	
Enquiry applications	268	
Writing MQSeries-IMS bridge applications	269	
How the MQSeries-IMS bridge deals with		
messages	269	
Mapping MQSeries messages to IMS		
transaction types	270	
If the message cannot be put to the IMS		
queue	270	
IMS bridge feedback codes	271	
	The MQMD fields in messages from the IMS	
	bridge	271
	The MQCIH fields in messages from the IMS	
	bridge	272
	Reply messages from IMS	273
	Using alternate response PCBs in IMS	
	transactions	273
	Sending unsolicited messages from IMS	273
	Message segmentation	274
	Data conversion	274
	Writing your program	275
	Writing MQSeries applications to invoke IMS	
	conversational transactions	276
	Triggering	276
	Writing programs containing IMS commands	276

Chapter 18. Object-oriented programming with

MQSeries	279
What is in the MQSeries Object Model?	279
Classes	279
Object references	280
Return codes	280
Programming language considerations	280
Coding in C++	280
Coding in Java	281

Chapter 6. Introducing the Message Queue Interface

This chapter introduces the features of the Message Queue Interface (MQI).

The remaining chapters in this part of the book describe how to use these features. Detailed descriptions of the calls, structures, data types, return codes, and constants are given in the *MQSeries Application Programming Reference* manual.

The MQI is introduced under these headings:

- “What is in the MQI?”
- “Parameters common to all the calls” on page 75
- “Specifying buffers” on page 76
- “Programming language considerations” on page 76
- “OS/390 batch considerations” on page 84
- “UNIX signal handling on MQSeries Version 5 products” on page 84

Note

The Application Messaging Interface (AMI) provides a simpler and higher-level programming interface than the MQI. Although it has some limitations compared with the MQI, its function should be sufficient for the majority of users. The AMI supports both point-to-point and publish/subscribe messaging models, and includes support for both MQSeries Integrator Version 1 and MQSeries Integrator Version 2. The AMI eliminates the need for application programmers to understand all of the options and functions available in the MQI. The Application Messaging Interface for C, C++, and Java™ runs on the following operating systems or environments: AIX, HP-UX, Sun Solaris, Microsoft® Windows 98, Microsoft Windows NT, and OS/390. The Application Messaging Interface for COBOL runs on the OS/390 operating system. Further information about the AMI is outside the scope of this book. For full details, refer to the *MQSeries Application Messaging Interface* book.

What is in the MQI?

The Message Queue Interface comprises the following:

- *Calls* through which programs can access the queue manager and its facilities
- *Structures* that programs use to pass data to, and get data from, the queue manager
- *Elementary data types* for passing data to, and getting data from, the queue manager

MQSeries for OS/390 also supplies:

- Two extra calls through which OS/390 batch programs can commit and back out changes.
- *Data definition files* (sometimes known as copy files, macros, include files, and header files) that define the values of constants supplied with MQSeries for OS/390.
- *Stub programs* to link-edit to your applications.

MQI contents

- A suite of sample programs that demonstrate how to use the MQI on the OS/390 platform. For further information about these samples, see “Chapter 35. Sample programs for MQSeries for OS/390” on page 421.

MQSeries for AS/400 also supplies:

- *Data definition files* (sometimes known as copy files, macros, include files, and header files) that define the values of constants supplied with MQSeries for AS/400.
- Three stub programs to link-edit to your ILE C, ILE COBOL, and ILE RPG applications.
- A suite of sample programs that demonstrate how to use the MQI on the AS/400 platform. For further information about these samples, see “Chapter 34. Sample programs (all platforms except OS/390)” on page 359.

MQSeries for OS/2 Warp, MQSeries for Windows NT, MQSeries for Compaq (DIGITAL) OpenVMS, and MQSeries on UNIX systems also supply:

- Calls through which MQSeries for OS/2 Warp, MQSeries for Windows NT, MQSeries for AS/400, and MQSeries on UNIX systems programs can commit and back out changes.
- *Include files* that define the values of constants supplied on these platforms.
- *Library files* to link your applications.
- A suite of sample programs that demonstrate how to use the MQI on these platforms.
- Sample source and executable code for bindings to external transaction managers.

MQSeries for Tandem NonStop Kernel also supplies:

- *Include files* that define the values of constants supplied with MQSeries for Tandem NonStop Kernel.
- *Library files* to link your applications.
- A suite of sample programs that demonstrate how to use the MQI on the Tandem NSK platform.

MQSeries for VSE/ESA also supplies:

- *Include files* that define the values of constants supplied with MQSeries for VSE/ESA.
- A suite of sample programs that demonstrate how to use the MQI on the VSE/ESA platform.

MQSeries for Windows provides a subset of the MQI. For more information, see the following:

- *MQSeries for Windows V2.0 User's Guide*.
- *MQSeries for Windows V2.1 User's Guide*.

Calls

The calls in the MQI can be grouped as follows:

MQCONN, MQCONNX, and MQDISC

Use these calls to connect a program to (with or without options), and disconnect a program from, a queue manager. If you write CICS programs for OS/390, or VSE/ESA, you do not need to use these calls. However, you are recommended to use them if you want your application to be portable to other platforms.

MQOPEN and MQCLOSE

Use these calls to open and close an object, such as a queue.

MQPUT and MQPUT1

Use these calls to put a message on a queue.

MQGET

Use this call to browse messages on a queue, or to remove messages from a queue.

MQINQ

Use this call to inquire about the attributes of an object.

MQSET

Use this call to set some of the attributes of a queue. You cannot set the attributes of other types of object.

MQBEGIN, MQCMIT, and MQBACK

Use these calls when MQSeries is the coordinator of a unit of work. MQBEGIN starts the unit of work. MQCMIT and MQBACK end the unit of work, either committing or rolling back the updates made during the unit of work. OS/400 commitment controller is used to coordinate global units of work on AS/400. Native start commitment control, commit, and rollback commands are used.

The MQI calls are described fully in the *MQSeries Application Programming Reference* manual.

Syncpoint calls

Syncpoint calls are available as follows:

MQSeries for OS/390 calls

MQSeries for OS/390 provides the MQCMIT and MQBACK calls. Use these calls in OS/390 batch programs to tell the queue manager that all the MQGET and MQPUT operations since the last syncpoint are to be made permanent (committed) or are to be backed out. To commit and back out changes in other environments:

CICS Use commands such as EXEC CICS SYNCPOINT and EXEC CICS SYNCPOINT ROLLBACK.

IMS Use the IMS syncpoint facilities, such as the GU (get unique) to the IOPCB, CHKP (checkpoint), and ROLB (rollback) calls.

RRS Use MQCMIT and MQBACK or SRRCMIT and SRRBACK as appropriate. (See "Transaction management and recoverable resource manager services" on page 181.)

Note: SRRCMIT and SRRBACK are 'native' RRS commands, they are not MQI calls.

For backward compatibility, the CSQBCMT and CSQBBAK calls are available as synonyms for MQCMIT and MQBACK. These are described fully in the *MQSeries Application Programming Reference* manual.

OS/400 calls

MQSeries for AS/400 provides the MQCMIT and MQBACK commands. You can also use the OS/400 COMMIT and ROLLBACK commands, or any other commands or calls that initiate the OS/400 commitment control facilities (for example, EXEC CICS SYNCPOINT).

MQI contents

MQSeries for Tandem NonStop Kernel calls

The default SYNCPOINT option for the MQPUT and MQGET calls is SYNCPOINT, rather than NO_SYNCPOINT. To use the default (SYNCPOINT) option for MQPUT, MQGET and MQPUT1 operations, the application must have an active TM/MP Transaction that defines the unit of work to be committed.

MQSeries for VSE/ESA calls

Use CICS commands such as EXEC CICS SYNCPOINT and EXEC CICS SYNCPOINT ROLLBACK. The batch interface and server support the MQCMIT and MQBACK calls which are translated into the CICS commands EXEC CICS SYNCPOINT and EXEC CICS SYNCPOINT ROLLBACK respectively. Use these calls in programs to tell the queue manager that all the MQGET and MQPUT operations since the last syncpoint are to be made permanent (committed) or are to be backed out.

MQSeries calls on other platforms

The following products provide the MQCMIT and MQBACK calls:

- MQSeries for OS/2 Warp
- MQSeries for Windows
- MQSeries for Windows NT
- MQSeries for Compaq (DIGITAL) OpenVMS
- MQSeries on UNIX systems

Use syncpoint calls in programs to tell the queue manager that all the MQGET and MQPUT operations since the last syncpoint are to be made permanent (committed) or are to be backed out. To commit and back out changes in the CICS environment, use commands such as EXEC CICS SYNCPOINT and EXEC CICS SYNCPOINT ROLLBACK.

Data conversion

The MQXCNVC - convert characters call is used only from a data-conversion exit. This call converts message character data from one character set to another.

See the *MQSeries Application Programming Reference* manual for the syntax used with the MQXCNVC call, and “Chapter 11. Writing data-conversion exits” on page 155 for guidance on writing and invoking data conversion exits.

Structures

Structures, used with the MQI calls listed in “Calls” on page 66, are supplied in data definition files for each of the supported programming languages. MQSeries for OS/390 and MQSeries for AS/400 supply files that contain constants for you to use when filling in some of the fields of these structures. For more information on these, see “MQSeries data definitions” on page 69.

All the structures are described fully in the *MQSeries Application Programming Reference* manual.

Elementary data types

For the C language, the MQI provides the following elementary data types or unstructured fields:

MQBYTE	A single byte of data
MQBYTEn	A string of 16, 24, 32, 40, or 64 bytes
MQCHAR	One single-byte character

MQCHARn	A string of 4, 8, 12, 16, 20, 28, 32, 48, 64, 128, or 256 single-byte characters
MQHCONN	A connection handle (this data is 32 bits long)
MQHOBJ	An object handle (this data is 32 bits long)
MQLONG	A 32-bit signed binary integer
PMQLONG	A pointer to data of type MQLONG

These data types are described fully in the *MQSeries Application Programming Reference* manual.

Table 1 shows the Visual Basic[®] equivalents of the C elementary data types.

Table 1. Visual Basic equivalents of the C elementary data types

C data type	Visual Basic data type
MQBYTE	String * 1
MQBYTEn	String * n
MQCHAR	String * 1
MQCHARn	String * n
MQHCONN	Long
MQHOBJ	Long
MQLONG	Long
PMQLONG	No equivalent

For COBOL, assembler, PL/I, or RPG, use the equivalent declarations shown in the relevant native language manual.

MQSeries data definitions

MQSeries for OS/390 supplies data definitions in the form of COBOL copy files, assembler-language macros, a single PL/I include file, a single C language include file, and C++ language include files.

MQSeries for AS/400 supplies data definitions in the form of COBOL copy files, RPG copy files, C language include files, and C++ language include files.

MQSeries for VSE/ESA supplies data definitions in the form of a C language include file, COBOL copy files, and PL/I include files.

The data definition files supplied with MQSeries contain:

- Definitions of all the MQSeries constants and return codes
- Definitions of the MQSeries structures and data types
- Constant definitions for initializing the structures
- Function prototypes for each of the calls (for PL/I and the C language only)

For a full description of MQSeries data definition files, see “Appendix F. MQSeries data definition files” on page 541.

MQSeries stub programs and library files

The stub programs and library files provided are listed here, for each platform.

MQI contents

For more information about how to use stub programs and library files when you build an executable application, see “Part 3. Building an MQSeries application” on page 283. For information about linking to C++ library files, see the *MQSeries Using C++* book.

MQSeries for OS/390

Before you can run an MQSeries for OS/390 program, you must link-edit it to the stub program supplied with MQSeries for OS/390 for the environment in which you are running the application. The stub program provides the first stage of the processing of your calls into requests that MQSeries for OS/390 can process.

MQSeries for OS/390 supplies the following stub programs:

CSQBSTUB	Stub program for OS/390 batch programs
CSQBRRSI	Stub program for OS/390 batch programs using RRS via the MQI
CSQBRSTB	Stub program for OS/390 batch programs using RRS directly
CSQCSTUB	Stub program for CICS programs
CSQQSTUB	Stub program for IMS programs
CSQXSTUB	Stub program for distributed queuing non-CICS exits
CSQASTUB	Stub program for data-conversion exits

Note: If you use the CSQBRSTB stub program you must link-edit with ATRSCSS from SYS1.CSSLIB. (SYS1.CSSLIB is also known as the “Callable Services Library”). For more information about RRS see “Transaction management and recoverable resource manager services” on page 181.

Alternatively, you can dynamically call the stub from within your program. This technique is described in “Dynamically calling the MQSeries stub” on page 315.

In IMS, you may also need to use a special language interface module that is supplied by MQSeries.

MQSeries for AS/400

In MQSeries for AS/400, you must link your program to the MQI library files supplied for the environment in which you are running your application, in addition to those provided by the operating system.

For non-threaded 4.2.1 applications:

AMQZSTUB	Server service program provided for compatibility with previous releases
AMQVSTUB	Data conversion service program provided for compatibility with previous releases. For a non-threaded application:
LIBMQM	Server service program
LIBMQIC	Client service program
IMQB23I4	C++ base service program
IMQS23I4	C++ server service program
LIBMQMZF	Installable exits for C

In a threaded application:

LIBMQM_R	Server service program
IMQB23I4_R	C++ base service program
IMQS23I4_R	C++ server service program

LIBMQMZF_R Installable exits for C

If you are using MQSeries for AS/400 you can write your applications in C++. To see how to link your C++ applications, and for full details of all aspects of using C++, see the book *MQSeries Using C++*.

MQSeries for OS/2 Warp

In MQSeries for OS/2 Warp, you must link your program to the MQI library files supplied for the environment in which you are running your application, in addition to those provided by the operating system:

MQM.LIB	Server for 32-bit C
MQIC.LIB	Client for C
MQMXA.LIB	Static XA interface for C
MQMCICS.LIB	CICS for OS/2 V2 exits for C
MQMCICS3.LIB	CICS Transaction Server for OS/2, V4 exits
MQMZFLIB	Installable services exits for C
MQICCB16.LIB	Client for 16-bit Micro Focus COBOL
MQMCB16.LIB	Server for 16-bit Micro Focus COBOL
MQMCBB.LIB	Server for 32-bit IBM VisualAge [®] COBOL
MQMCB32.LIB	Server for 32-bit Micro Focus COBOL
MQICCB.LIB	Client for 32-bit IBM VisualAge COBOL
MQICCB32.LIB	Client for 32-bit Micro Focus COBOL
IMQ*.LIB	Server for C++

MQSeries for Windows

In MQSeries for Windows, you must link your program to the MQI library files supplied for the environment in which you are running your application, in addition to those provided by the operating system:

MQM16.LIB	Server for 16-bit C
MQM.LIB	Server for 32-bit C

MQSeries for Windows NT

In MQSeries for Windows NT, you must link your program to the MQI library files supplied for the environment in which you are running your application, in addition to those provided by the operating system:

MQM.LIB	Server for 32-bit C
MQIC.LIB	Client for 16-bit C
MQIC32.LIB	Client for 32-bit C
MQMXA.LIB	Static XA interface for C
MQMCICS.LIB	CICS for Windows NT V2 exits for C
MQMCICS4.LIB	TXSeries for Windows NT, V4 exits for C
MQMZFLIB	Installable services exits for C
MQMCBB.LIB	Server for 32-bit IBM COBOL
MQMCB32	Server for 32-bit Micro Focus COBOL
MQICCB.LIB	Client for 32-bit IBM COBOL
MQICCB32	Client for 32-bit Micro Focus COBOL
IMQ*.LIB	Server for C++
MQMENC.LIB	Dynamic XA interface in C for Encina
MQMTUX.LIB	Dynamic XA interface in C for Tuxedo

MQI contents

MQSeries for AIX

In MQSeries for AIX, you must link your program to the MQI library files supplied for the environment in which you are running your application, in addition to those provided by the operating system.

In a non-threaded application:

libmqm.a	Server for C
libmqic.a	Client for C
libmqmzf.a	Installable service exits for C
libmqmxa.a	XA interface for C
libmqmcbprt.o	MQSeries run-time library for Micro Focus COBOL support
libmqmcb.a	Server for COBOL
libmqicb.a	Client for COBOL
libimq*.a	Client for C++

In a threaded application:

libc_r.a	Server for C
libmqm_r.a	Server for C
libmqmzf_r.a	Installable service exits for C
libmqmxa_r.a	XA interface for C
libimq*_r.a	Client for C++
libmqmxa_r.a	For Encina

MQSeries for AT&T GIS UNIX

In MQSeries for AT&T GIS UNIX, you must link your program to the MQI library files supplied for the environment in which you are running your application, in addition to those provided by the operating system.

libmqm.so	Server for C
libmqmzse.so	For C
libmqic.so	Client for C
libmqmcs.so	Client for C
libmqmzf.so	Installable service exits for C
libmqmxa.a	XA interface for C

MQSeries for Compaq (DIGITAL) OpenVMS

In MQSeries for Compaq (DIGITAL) OpenVMS, you must link your program to the MQI library files supplied for the environment in which you are running your application, in addition to those provided by the operating system:

mqm.exe	Server for C
mqic.exe	Client for C
mqmzf.exe	Installable service exits for C
mqmxa.exe	XA interface for C
mqcbrt.exe	MQSeries COBOL run-time
mqmcb.exe	Server for COBOL
mqicb.exe	Client for COBOL

MQSeries for Compaq Tru64 UNIX

In MQSeries for Compaq Tru64 UNIX, you must link your program to the MQI library files supplied for the environment in which you are running your application, in addition to those provided by the operating system. The files listed

are for DIGITAL UNIX Version 4.0D or Version 4.0E, and Compaq Tru64 UNIX Version 4.0F or Version 5.0, except where stated.

In a non-threaded application on V4.0 or *any* application on V5.0:

libmqm.so	Server for C
libmqic.so	Client for C
libimqs23ff.so	Server for C++
libimqc23ff.so	Client for C++
libmqmcb.so	Server for COBOL
libmqmicb.so	Client for COBOL
libmqmzf.so	Installable service exits (not Compaq Tru64 UNIX Version 5.0)

In a threaded application on V4.0:

libmqm_r.so	Server for C
libmqic_r.so	Client for C
libimqs23ff_r.so	Server for C++
libimqc23ff_r.so	Client for C++
libmqmcb_r.so	COBOL
libmqmzf.so	Installable service exits (not Compaq Tru64 UNIX Version 5.0)

MQSeries for HP-UX

In MQSeries for HP-UX, you must link your program to the MQI library files supplied for the environment in which you are running your application, in addition to those provided by the operating system.

In a non-threaded application:

libmqm.sl	Server for C
libmqic.sl	Client for C
libmqmzf.sl	Installable service exits for C
libmqmxa.sl	XA interface for C
libmqmcbprt.o	MQSeries run-time library for Micro Focus COBOL support
libmqmcb.sl	Server for COBOL
libmqicb.sl	Client for COBOL

In a threaded application:

libmqm_r.sl	Server for C
libmqmzf_r.sl	Installable service exits for C
libmqmxa_r.sl	XA interface for C

MQSeries for SINIX and DC/OSx

In MQSeries for SINIX and DC/OSx, you must link your program to the MQI library files supplied for the environment in which you are running your application, in addition to those provided by the operating system.

In a non-threaded application:

libmqm.so	Server for C
libmqmzse.so	For C
libmqic.so	Client for C
libmqmcs.so	Client for C

MQI contents

libmqmzf.so	Installable service exits for C
libmqmxa.a	XA interface for C
libmqmcbprt.o	MQSeries COBOL run-time
libmqmcb.so	Server for COBOL
libmqicb.so	Client for COBOL

In a threaded application:

libmqm_r.so	For C
libmqmcs_r.so	For C
libmqmcics_r.so	For CICS
libmqmxa_r.a	For XA interface in C

DOS and Windows 3.1 clients

In DOS and Windows 3.1, you must link your program to the MQIC.LIB library file (or imq*vw.lib for C++), followed by the protocol libraries, indicating the protocol you do and do not want.

mqicn.lib	NetBIOS required
mqicdn.lib	NetBIOS not required
mqict.lib	TCP/IP required
mqicdt.lib	TCP/IP not required

libmqm.so	Server for C
libmqmzse.so	For C
libmqic.so	Client for C
libmqmcs.so	Client for C
libmqmzf.so	Installable service exits for C
libmqmxa.a	XA interface for C
imq*.so	C++

MQSeries for Sun Solaris

In MQSeries for Sun Solaris, you must link your program to the MQI library files supplied for the environment in which you are running your application in addition to those provided by the operating system.

libmqm.so	Server for C
libmqmzse.so	For C
libmqic.so	Client for C
libmqmcs.so	Client for C
libmqmzf.so	Installable service exits for C
libmqmxa.a	XA interface for C
imq*.so	C++

MQSeries for VSE/ESA

In MQSeries for VSE/ESA you must link your program to the install sublibrary PRD2.MQSERIES (this is its default name). This sublibrary contains all the required object decks.

MQSeries for Tandem NonStop Kernel

In MQSeries for Tandem NonStop Kernel, you must link your program to the MQI library files supplied for the environment in which you are running your application in addition to those provided by the operating system.

<code>mqmlibc</code>	For C, non-native
<code>mqmlibt</code>	For TAL or COBOL, non-native
<code>mqmlibnc</code>	For native C
<code>mqmlibnt</code>	For native TAL or COBOL

Parameters common to all the calls

There are two types of parameter common to all the calls: handles and return codes.

Using connection and object handles

For a program to communicate with a queue manager, the program must have a unique identifier by which it knows that queue manager. This identifier is called a *connection handle*. For CICS programs, the connection handle is always zero. For all other platforms or styles of programs, the connection handle is returned by the MQCONN or MQCONNX call when the program connects to the queue manager. Programs pass the connection handle as an input parameter when they use the other calls.

For a program to work with an MQSeries object, the program must have a unique identifier by which it knows that object. This identifier is called an *object handle*. The handle is returned by the MQOPEN call when the program opens the object to work with it. Programs pass the object handle as an input parameter when they use subsequent MQPUT, MQGET, MQINQ, MQSET, or MQCLOSE calls.

Understanding return codes

A completion code and a reason code are returned as output parameters by each call. These are known collectively as *return codes*.

To show whether or not a call is successful, each call returns a *completion code* when the call is complete. The completion code is usually either MQCC_OK or MQCC_FAILED, showing success and failure, respectively. Some calls can return an intermediate state, MQCC_WARNING, indicating partial success.

Each call also returns a *reason code* that shows the reason for the failure, or partial success, of the call. There are many reason codes, covering such circumstances as a queue being full, get operations not being allowed for a queue, and a particular queue not being defined for the queue manager. Programs can use the reason code to decide how to proceed. For example, they could prompt the user of the program to make changes to his input data, then make the call again, or they could return an error message to the user.

When the completion code is MQCC_OK, the reason code is always MQRC_NONE.

The completion and reason codes for each call are listed with the description of that call in the *MQSeries Application Programming Reference*

You will also find further information (including some ideas for corrective action) for each completion and reason code, in the *MQSeries Application Programming Reference* manual.

Specifying buffers

The queue manager refers to buffers only if they are required. If you do not require a buffer on a call or the buffer is zero in length, you can use a null pointer to a buffer.

Always use `datalength` when specifying the size of the buffer you require.

When you use a buffer to hold the output from a call (for example, to hold the message data for an `MQGET` call, or the values of attributes queried by the `MQINQ` call), the queue manager attempts to return a reason code if the buffer you specify is not valid or is in read-only storage. However, it may not be able to return a reason code in some situations.

Programming language considerations

MQSeries provides support for the following programming languages:

- C.
- C++ (MQSeries for AIX, AS/400, HP-UX, OS/2, OS/390, Sun Solaris, Compaq Tru64 UNIX, and Windows NT only). See the *MQSeries Using C++* book for information about coding MQSeries programs in C++.
- Visual Basic (MQSeries for Windows and Windows NT only). See the *MQSeries for Windows Version 2.0 User's Guide* and the *MQSeries for Windows Version 2.1 User's Guide* for information about coding MQSeries programs in Visual Basic.
- COBOL.
- Assembler language (MQSeries for OS/390 only).
- RPG (MQSeries for AS/400 only).
- PL/I (MQSeries for OS/390, AIX, OS/2 Warp, VSE/ESA, and Windows NT only).
- TAL (MQSeries for Tandem NonStop Kernel only).

The call interface, and how you can code the calls in each of these languages, is described in the *MQSeries Application Programming Reference* manual.

MQSeries provides data definition files to assist you with the writing of your applications. For a full description, see "Appendix F. MQSeries data definition files" on page 541.

If you can choose which language to code your programs in, you should consider the maximum length of the messages that your programs will process. If your programs will process only messages of a known maximum length, you can code them in any of the supported programming languages. But if you do not know the maximum length of the messages the programs will have to process, the language you choose will depend on whether you are writing a CICS, IMS, or batch application:

IMS and batch

Code the programs in C, PL/I, or assembler language to use the facilities these languages offer for obtaining and releasing arbitrary amounts of memory. Alternatively, you could code your programs in COBOL, but use assembler language, PL/I, or C subroutines to get and release storage.

CICS Code the programs in any language supported by CICS. The EXEC CICS interface provides the calls for managing memory, if necessary.

Coding in C

See “Appendix A. Language compilers and assemblers” on page 471 for the compilers that you can use to process your C programs.

Note the information in the following sections when coding MQSeries programs in C.

Parameters of the MQI calls

Parameters that are *input-only* and of type MQHCONN, MQHOBJ, or MQLONG are passed by value; for all other parameters, the *address* of the parameter is passed by value.

Not all parameters that are passed by address need to be specified every time a function is invoked. Where a particular parameter is not required, a null pointer can be specified as the parameter on the function invocation, in place of the address of the parameter data. Parameters for which this is possible are identified in the call descriptions.

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

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

Parameters with undefined data type

The MQGET, MQPUT, and MQPUT1 functions each have one parameter that has an undefined data type, namely the *Buffer* parameter. This parameter is used to send and receive the application’s message data.

Parameters of this sort are shown in the C examples as arrays of MQBYTE. It is valid to declare the parameters in this way, but it is usually more convenient to declare them as the particular structure that describes the layout of the data in the message. The function parameter is declared as a pointer-to-void, and so the address of any sort of data can be specified as the parameter on the function invocation.

Data types

All data types are defined by means of the typedef statement. For each data type, the corresponding pointer data type is also defined. The name of the pointer data type is the name of the elementary or structure data type prefixed with the letter “P” to denote a pointer. The attributes of the pointer are defined by the MQPOINTER macro variable; the value of this macro variable depends on the environment. The following illustrates how pointer data types are declared:

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

Manipulating binary strings

Strings of binary data are declared as one of the MQBYTEn data types. Whenever you copy, compare, or set fields of this type, use the C functions memcpy, memcmp, or memset:

```
#include <string.h>
#include "cmqc.h"

MQMD MyMsgDesc;
```

Programming language considerations

```
memcpy(MyMsgDesc.MsgId,          /* set "MsgId" field to nulls */
       MQMI_NONE,                /* ...using named constant */
       sizeof(MyMsgDesc.MsgId));

memset(MyMsgDesc.CorrelId,       /* set "CorrelId" field to nulls */
       0x00,                    /* ...using a different method */
       sizeof(MQBYTE24));
```

Do not use the string functions `strcpy`, `strcmp`, `strncpy`, or `strncmp` because these do not work correctly with data declared as `MQBYTE24`.

Manipulating character strings

When the queue manager returns character data to the application, the queue manager always pads the character data with blanks to the defined length of the field. The queue manager *does not* return null-terminated strings, but you can use them in your input. Therefore, when copying, comparing, or concatenating such strings, use the string functions `strncpy`, `strncmp`, or `strncat`.

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

Initial values for structures

The include file `<cmqc.h>` defines various macro variables that may be used to provide initial values for the structures when instances of those structures are declared. These macro variables have names of the form `MQxxx_DEFAULT`, where `MQxxx` represents the name of the structure. Use them like this:

```
MQMD   MyMsgDesc = {MQMD_DEFAULT};
MQPMO  MyPutOpts = {MQPMO_DEFAULT};
```

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

- One macro variable defines the value as a string whose length, excluding the implied null, matches exactly the defined length of the field. For example, (the symbol `b` represents a blank character):

```
#define MQMD_STRUC_ID "MDbb"
#define MQFMT_STRING "MQSTRbbb"
```

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

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

```
#define MQMD_STRUC_ID_ARRAY 'M','D','b','b'
#define MQFMT_STRING_ARRAY 'M','Q','S','T','R','b','b','b'
```

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

Initial values for dynamic structures

When a variable number of instances of a structure are required, the instances are usually created in main storage obtained dynamically using the `calloc` or `malloc` functions.

Programming language considerations

To initialize the fields in such structures, the following technique is recommended:

1. Declare an instance of the structure using the appropriate MQ_{xxx}_DEFAULT macro variable to initialize the structure. This instance becomes the “model” for other instances:

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

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

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

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

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

```
memcpy(InstancePtr,&ModelMsgDesc,sizeof(MQMD));  
/* initialize dynamic instance */
```

Use from C++

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

```
#ifdef __cplusplus  
extern "C" {  
#endif  
  
/* rest of header file */  
  
#ifdef __cplusplus  
}  
#endif
```

Coding in COBOL

See “Appendix A. Language compilers and assemblers” on page 471 for the compilers that you can use to process your COBOL programs.

Note the information in the following sections when coding MQSeries programs in COBOL.

Named constants

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

Constants that have character-string values use the single quotation mark character (') as the string delimiter. To make the compiler accept this character, use the compiler option APOST.

The copy file CMQV contains declarations of the named constants as level-10 items. To use the constants, declare the level-01 item explicitly, then use the COPY statement to copy in the declarations of the constants:

```
WORKING-STORAGE SECTION.  
01 MQM-CONSTANTS.  
COPY CMQV.
```

However, this method causes the constants to occupy storage in the program even if they are not referred to. If the constants are included in many separate programs within the same run unit, multiple copies of the constants will exist—this may

Programming language considerations

result in a significant amount of main storage being used. You can avoid this situation by adding the GLOBAL clause to the level-01 declaration:

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

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

Coding in System/390[®] assembler language

System/390 assembler is supported on OS/390 only.

See “Appendix A. Language compilers and assemblers” on page 471 for the assemblers that you can use to process your assembler-language programs.

Note the information in the following sections when coding MQSeries for OS/390 programs in assembler language.

Names

In this book, the names of parameters in the descriptions of calls, and the names of fields in the descriptions of structures are shown in mixed case. In the assembler-language macros supplied with MQSeries, all names are in uppercase.

Using the MQI calls

The MQI is a call interface, so assembler-language programs must observe the OS linkage convention. In particular, before they issue an MQI call, assembler-language programs must point register R13 at a save area of at least 18 full words. This save area is to provide storage for the called program. It stores the registers of the caller before their contents are destroyed, and restores the contents of the caller’s registers on return.

Note: This is of particular importance for CICS assembler-language programs that use the DFHEIENT macro to set up their dynamic storage, but that choose to override the default DATAREG from R13 to other registers. When the CICS Resource Manager Interface receives control from the stub, it saves the current contents of the registers at the address to which R13 is pointing. Failing to reserve a proper save area for this purpose gives unpredictable results, and will probably cause an abend in CICS.

Declaring constants

Most constants are declared as equates in macro CMQA. However, the following constants cannot be defined as equates, and these are not included when you call the macro using default options:

```
MQACT_NONE
MQCL_NONE
MQFMT_NONE
MQFMT_ADMIN
MQFMT_COMMAND_1
MQFMT_COMMAND_2
MQFMT_DEAD_LETTER_HEADER
MQFMT_EVENT
MQFMT_IMS
MQFMT_IMS_VAR_STRING
MQFMT_PCF
MQFMT_STRING
```

```
MQFMT_TRIGGER
MQFMT_XMIT_Q_HEADER
MQMI_NONE
```

To include them, add the keyword EQUONLY=NO when you call the macro.

CMQA is protected against multiple declaration, so you can include it many times. However, the keyword EQUONLY takes effect only the first time the macro is included.

Specifying the name of a structure

To allow more than one instance of a structure to be declared, the macro that generates the structure prefixes the name of each field with a user-specifiable string and an underscore character (_). Specify the string when you invoke the macro. If you do not specify a string, the macro uses the name of the structure to construct the prefix:

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

The structure declarations in the *MQSeries Application Programming Reference* manual show the default prefix.

Specifying the form of a structure

The macros can generate structure declarations in one of two forms, controlled by the DSECT parameter:

DSECT=YES	An assembler-language DSECT instruction is used to start a new data section; the structure definition immediately follows the DSECT statement. No storage is allocated, so no initialization is possible. The label on the macro invocation is used as the name of the data section; if no label is specified, the name of the structure is used.
DSECT=NO	Assembler-language DC instructions are used to define the structure at the current position in the routine. The fields are initialized with values, which you can specify by coding the relevant parameters on the macro invocation. Fields for which no values are specified on the macro invocation are initialized with default values.

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

Controlling the listing

You can control the appearance of the structure declaration in the assembler-language listing by means of the LIST parameter:

LIST=YES	The structure declaration appears in the assembler-language listing.
LIST=NO	The structure declaration does not appear in the assembler-language listing. This is assumed if the LIST parameter is not specified.

Specifying initial values for fields

You can specify the value to be used to initialize a field in a structure by coding the name of that field (without the prefix) as a parameter on the macro invocation, accompanied by the value required.

Programming language considerations

For example, to declare a message descriptor structure with the *MsgType* field initialized with MQMT_REQUEST, and the *ReplyToQ* field initialized with the string MY_REPLY_TO_QUEUE, you could use the following code:

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

If you specify a named constant (or equate) as a value on the macro invocation, you must use the CMQA macro to define the named constant. You must not enclose in single quotation marks (' ') values that are character strings.

Writing reenterable programs

MQSeries uses its structures for both input and output. If you want your program to remain reenterable, you should:

1. Define working storage versions of the structures as DSECTs, or define the structures inline within an already-defined DSECT. Then copy the DSECT to storage that is obtained using:
 - For batch and TSO programs, the STORAGE or GETMAIN OS/390 assembler macros
 - For CICS, the working storage DSECT (DFHEISTG) or the EXEC CICS GETMAIN command

To correctly initialize these working storage structures, copy a constant version of the corresponding structure to the working storage version.

Note: The MQMD and MQXQH structures are each more than 256 bytes long. To copy these structures to storage, you will have to use the MVCL assembler instruction.

2. Reserve space in storage by using the LIST form (MF=L) of the CALL macro. When you use the CALL macro to make an MQI call, use the EXECUTE form (MF=E) of the macro, using the storage reserved earlier, as shown in the example under "Using CEDF". For more examples of how to do this, see the assembler language sample programs as shipped with MQSeries.

Use the assembler language RENT option to help you determine if your program is reenterable.

For information on writing reenterable programs, see the *MVS/ESA Application Development Guide: Assembler Language Programs*, GC28-1644.

Using CEDF

If you want to use the CICS-supplied transaction, CEDF (CICS Execution Diagnostic Facility) to help you to debug your program, you must add the ,VL keyword to each CALL statement, for example:

```
CALL MQCONN,(NAME,HCONN,COMPCODE,REASON),MF=(E,PARMAREA),VL
```

The above example is reenterable assembler-language code where PARMAREA is an area in the working storage you specified.

Coding in RPG

RPG is supported on OS/400 only.

See "Appendix A. Language compilers and assemblers" on page 471 for the compilers that you can use to process your RPG programs.

Programming language considerations

In this book, the parameters of calls, the names of data types, the fields of structures, and the names of constants are described using their long names. In RPG, these names are abbreviated to six or fewer uppercase characters. For example, the field *MsgType* becomes *MDMT* in RPG. For more information, see the *MQSeries for AS/400 Application Programming Reference (ILE RPG)* manual.

Coding in PL/I

PL/I is supported on AIX, OS/390, OS/2 Warp, VSE/ESA, and Windows NT only.

See “Appendix A. Language compilers and assemblers” on page 471 for the compilers that you can use to process your PL/I programs.

Note the information in the following sections when coding MQSeries for OS/390 programs in PL/I.

Structures

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

An instance of a structure can be declared by using the `like` attribute, for example:

```
dc1 my_mqmd          like MQMD; /* one instance */
dc1 my_other_mqmd   like MQMD; /* another one */
```

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

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

Named constants

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

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

```
%dc1 MQMD_STRUC_ID char;
%MQMD_STRUC_ID = 'MQMD';

%dc1 MQMD_VERSION_1 char;
%MQMD_VERSION_1 = '1';
```

Coding in TAL

TAL is supported on Tandem NonStop Kernel only.

See “Appendix A. Language compilers and assemblers” on page 471 for the compilers that you can use to process your TAL programs.

Programming language considerations

Note the following when coding MQSeries for Tandem NonStop Kernel programs in TAL:

- The MQI library (bound into the application process) does not open \$RECEIVE and does not open \$TMP (TM/MP transaction pseudo-file) itself, so you may code your application to use these features.
- The MQI library uses a SERVERCLASS_SEND_() call in initial communication with the Queue Manager. While connected, it maintains two process file opens (with the LINKMON process and a Local Queue Manager Agent) and a small number of disk file opens (fewer than 10).

OS/390 batch considerations

OS/390 batch programs that call the MQI can be in either supervisor or problem state. However, they must meet the following conditions:

- They must be in task mode, not service request block (SRB) mode.
- They must be in Primary address space control (ASC) mode (not Access Register ASC mode).
- They must not be in cross-memory mode. The primary address space number (ASN) must be equal to the secondary ASN and the home ASN.
- No OS/390 locks can be held.
- There can be no function recovery routines (FRRs) on the FRR stack.
- Any program status word (PSW) key can be in force for the MQCONN or MQCONNX call (provided the key is compatible with using storage that is in the TCB key), but subsequent calls that use the connection handle returned by MQCONN or MQCONNX:
 - Must have the same PSW key that was used on the MQCONN or MQCONNX call
 - Must have parameters accessible (for write, where appropriate) under the same PSW key
 - Must be issued under the same task (TCB), but not in any subtask of the task
- They can be in either 24-bit or 31-bit addressing mode. However, if 24-bit addressing mode is in force, parameter addresses must be interpreted as valid 31-bit addresses.

If any of these conditions is not met, a program check may occur. In some cases the call will fail and a reason code will be returned.

UNIX signal handling on MQSeries Version 5 products

This does not apply to MQSeries for OS/390.

In general, UNIX and AS/400 systems have moved from a nonthreaded (process) environment to a multithreaded environment. In the nonthreaded environment, some functions could be implemented only by using signals, though most applications did not need to be aware of signals and signal handling. In the multithreaded environment, thread-based primitives support some of the functions that used to be implemented in the nonthreaded environments using signals. In many instances, signals and signal handling, although supported, do not fit well into the multithreaded environment and various restrictions exist. This can be particularly problematic when you are integrating application code with different middleware libraries (running as part of the application) in a multithreaded environment where each is trying to handle signals. The traditional approach of saving and restoring signal handlers (defined per process), which worked when

there was only one thread of execution within a process, does not work in a multithreaded environment: many threads of execution could be trying to save and restore a process-wide resource, with unpredictable results.

For a standard application, MQSeries supports both nonthreaded and threaded application environments on AIX, AS/400, and HP-UX.

MQSeries for AS/400 uses ILE/C condition and cancel handlers as its exception processing mechanisms. Because of this, applications must not use the ILE/C `signal()` API when connected to MQSeries. The `signal()` API is implemented by ILE to handle ILE/C conditions as if they were signals, and can interfere with the ILE/C condition handlers used by MQSeries.

`Sigaction()` and `sigwait()` are safe to use with MQSeries, because they do not interact with ILE conditions at all. The ILE condition and cancel handler APIs are also safe to use in all circumstances. These APIs, when used together, will handle the same combination of exception conditions as `signal()`.

All MQSeries applications in the Sun Solaris environment are threaded. MQSeries for Sun Solaris V2.2 supported only single-threaded applications (though there was no way to enforce this) and, because there was only one thread of execution, was able to make use of the traditional signal handling functions. In MQSeries for Sun Solaris, V5.0, and subsequent releases, true multithreaded applications are supported and so the signal behavior has changed.

The library `libmqm` is provided for migration of nonthreaded applications from Version 2 of MQSeries for AIX or MQSeries for HP-UX to Version 5. The goal of this library is to maintain the Version 2 behavior (including signals) for nonthreaded applications. Within an application in this environment there is only one thread of execution, which means that signal handlers can be saved and restored safely across MQSeries API calls (as can any middleware library that is part of the application). Therefore, if you have an application suite on V2 of MQSeries for AIX or MQSeries for HP-UX that uses signals, and you do not want to move to the threaded environment, the suite should run unchanged on V5 using the nonthreaded library, `libmqm`.

The library `libmqm_r` is provided for threaded applications on MQSeries for AIX or MQSeries for HP-UX. On AS/400 `libmqm_r` is provided as a service program. However, the behavior, particularly for signals, is different:

- As in the nonthreaded environment, MQSeries still establishes signal handlers for synchronous terminating signals (SIGBUS, SIGFPE, SIGSEGV).
- MQSeries must run some clean-up code during abnormal termination. This is achieved by setting up a `sigwait` thread to handle terminating, asynchronous signals. While this approach is suitable for an application that does not handle signals, it can cause problems when the signals being trapped on the MQSeries `sigwait` thread overlap with signals that an application wishes to intercept.
- Even in the threaded environment MQSeries needs a signal for its internal processing. As was stated earlier, use of signals in a threaded environment may cause problems when you are integrating a middleware stack. (With many threads all independently trying to handle signals, saving and restoring signal handlers, results are unpredictable.) MQSeries must use one signal: SIGALRM.

Note: Some system functions may use signals internally (for example, SIGALRM in a nonthreaded environment). For a particular operating system, some of these functions may have thread-safe equivalents or it may be stated

UNIX signal handling

that they are not multithread safe. Any non-thread-safe operating system call should be replaced if moving to a multithreaded environment.

Unthreaded applications

Each MQI function sets up its own signal handler for the signals:

- SIGALRM
- SIGBUS
- SIGFPE
- SIGSEGV

Users' handlers for these are replaced for the duration of the MQI function call. Other signals can be caught in the normal way by user-written handlers. If you do not install a handler, the default actions (for example, ignore, core dump, or exit) are left in place.

Note: On Sun Solaris all applications are threaded even if they use a single thread.

Threaded applications

A thread is considered to be connected to MQSeries from MQCONN (or MQCONNX) until MQDISC.

Synchronous signals

Synchronous signals arise in a specific thread. UNIX safely allows the setting up of a signal handler for such signals for the whole process. However, MQSeries sets up its own handler for the following signals, in the application process, while any thread is connected to MQSeries:

- SIGBUS
- SIGFPE
- SIGSEGV

If you are writing multithreaded applications, you should note that there is only one process-wide signal handler for each signal. MQSeries alters this signal handler when the application is connected to MQSeries. If one of these signals occurs while not on a thread connected to MQSeries, MQSeries attempts to call the signal handler that was in effect at the time of the first MQSeries connection within the process. Application threads must not establish signal handlers for these signals while there is any possibility that another thread of the same process is also connected to MQSeries.

Asynchronous signals

Asynchronous signals arise outside the whole process. UNIX does not guarantee predictable behavior for handling asynchronous signals, in certain situations, when running multithreaded. MQSeries must perform clean-up of thread and process resources as part of the termination from these asynchronous signals:

- SIGCHLD
- SIGHUP
- SIGINT
- SIGQUIT
- SIGTERM

MQSeries establishes a sigwait thread in the application process to intercept these signals.

These signals must not be used by the application when running multithreaded and when any thread is within an MQSeries connection. These signals should not

be unmasked within any application thread; be aware of the default status of the signal mask for threads that do not make MQSeries calls.

MQSeries use of SIGALRM

For communication purposes MQSeries needs a signal for its internal use. This signal should not be used by the application while any thread is within an MQSeries connection.

Threaded client applications - additional considerations

MQSeries handles the following signals during I/O to a server. These signals are defined by the communications stack. The application should not establish a signal handler for these signals while a thread of the process is making an MQSeries call:

SIGPIPE

(for TCP/IP)

SIGUSR1

(for LU 6.2)

Fastpath (trusted) applications

Fastpath applications run in the same process as MQSeries and so are running in the multithreaded environment. In this environment the application should not use any signals or timer interrupts. If a Fastpath application intercepts such an event, the queue manager must be stopped and restarted, or it may be left in an undefined state. For a full list of the restrictions for Fastpath applications under MQCONN see "Connecting to a queue manager using the MQCONN call" on page 92 .

MQI function calls within signal handlers

While you are in a signal handler, you cannot call an MQI function. If you call an MQI function, while another MQI function is active, MQRC_CALL_IN_PROGRESS is returned. If you call an MQI function, while no other MQI function is active, it is likely to fail because of the operating system restrictions on which calls can be issued from within a handler.

In the case of C++ destructor methods, which may be called automatically during program exit, you may not be able to stop the MQI functions from being called. Therefore, ignore any errors about MQRC_CALL_IN_PROGRESS. If a signal handler calls exit(), MQSeries backs out uncommitted messages in syncpoint as normal and closes any open queues.

Signals during MQI calls

MQI functions do not return the code EINTR or any equivalent to application programs. If a signal occurs during an MQI call, and the handler calls 'return', the call continues to run as if the signal had not happened. In particular, MQGET cannot be interrupted by a signal to return control immediately to the application. If you want to break out of an MQGET, set the queue to GET_DISABLED; alternatively, use a loop around a call to MQGET with a finite time expiry (MQGMO_WAIT with gmo.WaitInterval set), and use your signal handler (in a nonthreaded environment) or equivalent function in a threaded environment to set a flag which breaks the loop.

In the AIX environment, MQSeries requires that system calls interrupted by signals are restarted. You must establish the signal handler with sigaction(2) and set the SA_RESTART flag in the sa_flags field of the new action structure. The default behavior is that calls are not restarted (the SA_RESTART flag is not set).

UNIX signal handling

User exits and installable services

User exits and installable services that run as part of an MQSeries process in a multithreaded environment have the same restrictions as for Fastpath applications. They should be considered as permanently connected to MQSeries and so not use signals or non-threadsafe operating system calls.

Chapter 7. Connecting and disconnecting a queue manager

To use MQSeries programming services, a program must have a connection to a queue manager. The way this connection is made depends on the platform and the environment in which the program is operating:

OS/390 batch, MQSeries for AS/400, MQSeries for Compaq (DIGITAL) OpenVMS, MQSeries for OS/2 Warp, MQSeries for Tandem NonStop Kernel, MQSeries on UNIX systems, MQSeries for Windows, and MQSeries for Windows NT

Programs that run in these environments can use the MQCONN MQI call to connect to, and the MQDISC call to disconnect from, a queue manager. Alternatively, OS/390 batch, MQSeries on UNIX systems, MQSeries for AS/400, MQSeries for OS/2 Warp, and MQSeries for Windows NT can use the MQCONNX call. This chapter describes how writers of such programs should use these calls.

OS/390 batch programs can connect, consecutively or concurrently, to multiple queue managers on the same TCB.

IMS The IMS control region is connected to one or more queue managers when it starts. This connection is controlled by IMS commands. (For information on how to control the IMS adapter of MQSeries for OS/390, see the *MQSeries for OS/390 System Administration Guide*.) However, writers of message queuing IMS programs must use the MQCONN MQI call to specify the queue manager to which they want to connect. They can use the MQDISC call to disconnect from that queue manager. This chapter describes how writers of such programs should use these calls. Before the IMS adapter processes a message for another user following a Get Unique call from the IOPCB, or one implied by a checkpoint call, the adapter ensures that the application closes handles and disconnects from the queue manager.

IMS programs can connect, consecutively or concurrently, to multiple queue managers on the same TCB.

CICS Transaction Server for OS/390 and CICS for MVS/ESA

CICS programs do not need to do any work to connect to a queue manager because the CICS system itself is connected. This connection is usually made automatically at initialization, but you can also use the CQKC transaction, which is supplied with MQSeries for OS/390. CQKC is discussed in the *MQSeries for OS/390 System Administration Guide*.

CICS tasks can connect only to the queue manager to which the CICS region, itself, is connected.

Note: CICS programs can also use the MQI connect and disconnect calls (MQCONN and MQDISC). You may want to do this so that you can port these applications to non-CICS environments with a minimum of recoding. Be warned, though, that these calls *always* complete successfully in a CICS environment. This means that the return code may not reflect the true state of the connection to the queue manager.

TXSeries for Windows NT and Open Systems

These programs do not need to do any work to connect to a queue

Connecting and disconnecting

manager because the CICS system itself is connected. Therefore, only one connection at a time is supported. CICS applications must issue an MQCONN call to obtain a connection handle, and should issue an MQDISC call before they exit.

MQSeries for VSE/ESA

In your VSE/ESA application, make an explicit call to MQCONN to establish a connection to the VSE/ESA queue manager. Ensure that your application issues an MQDISC call to disconnect. The performance of your application is better if you connect and disconnect as infrequently as possible.

This chapter introduces connecting to and disconnecting from a queue manager, under these headings:

- “Connecting to a queue manager using the MQCONN call”
- “Connecting to a queue manager using the MQCONNX call” on page 92
- “Disconnecting programs from a queue manager using MQDISC” on page 95

Connecting to a queue manager using the MQCONN call

In general, you can connect either to a specific queue manager, or to the default queue manager:

- For MQSeries for OS/390, in the batch environment, the default queue manager is specified in the CSQBDEFV module.
- For MQSeries for AS/400, MQSeries for Compaq (DIGITAL) OpenVMS, MQSeries for OS/2 Warp, MQSeries for Tandem NonStop Kernel, and MQSeries on UNIX systems, the default queue manager is specified in the mqs.ini file.
- For MQSeries for Tandem NonStop Kernel, the default queue manager is specified in the MQSINI file, resident in the ZMQSSYS subvolume.
- For MQSeries for Windows NT, the default queue manager is specified in the registry.
- MQSeries for Windows allows only one queue manager to run at a time; it uses the running queue manager as its default.
- MQSeries for VSE/ESA allows only one queue manager to run at a time; its name is specified in the Global System Definition of the System Management Facility (SMF). Your application can specify the name or use the default value.

Alternatively, in the OS/390 MVS batch, TSO, and RRS environments you may connect to any one queue manager within a queue-sharing group. The MQCONN or MQCONNX request will select any one of the active members of the group.

The queue manager you connect to must be *local* to the task. This means that it must belong to the same system as the MQSeries application.

In the IMS environment, the queue manager must be connected to the IMS control region and to the dependent region that the program uses. The default queue manager is specified in the CSQQDEFV module when MQSeries for OS/390 is installed.

With the CICS on Open Systems environment, and TXSeries for Windows NT and AIX, the queue manager must be defined as an XA resource to CICS.

To connect to the default queue manager, call MQCONN, specifying a name consisting entirely of blanks or starting with a null (X'00') character.

Within MQSeries on UNIX systems, an application must be authorized for it to successfully connect to a queue manager. For more information, see the *MQSeries System Administration Guide* for MQSeries for AIX, HP-UX, and Sun Solaris; for other platforms, see the appropriate *System Management Guide*.

The output from MQCONN is:

- A connection handle
- A completion code
- A reason code

You will need to use the connection handle on subsequent MQI calls.

If the reason code indicates that the application is already connected to that queue manager, the connection handle that is returned is the same as the one that was returned when the application first connected. So the application probably should not issue the MQDISC call in this situation because the calling application will expect to remain connected.

The scope of the connection handle is the same as that for the object handle (see "Opening objects using the MQOPEN call" on page 98).

Descriptions of the parameters are given in the description of the MQCONN call in the *MQSeries Application Programming Reference* manual.

The MQCONN call fails if the queue manager is in a quiescing state when you issue the call, or if the queue manager is shutting down.

Scope of MQCONN

Within MQSeries for AS/400, MQSeries for Compaq (DIGITAL) OpenVMS, MQSeries for OS/2 Warp, MQSeries on UNIX systems, MQSeries for Windows, and MQSeries for Windows NT, the scope of an MQCONN call is the thread that issued it. That is, the connection handle returned from an MQCONN call is valid only within the thread that issued the call. Only one call may be made at any one time using the handle. If it is used from a different thread, it will be rejected as invalid. If you have multiple threads in your application and each wishes to use MQSeries calls, each one must individually issue MQCONN.

Each thread can connect to a different queue manager on OS/2 and Windows NT, but not on OS/400 or UNIX.

If your application is running as a client, it may connect to more than one queue manager within a thread. This does not apply if your application is not running as a client.

OS/2 has a limit of 4095 active threads in a system. However, the default is 64. This value may be controlled by the THREADS=xxxx parameter in CONFIG.SYS. Limitations on the number of concurrent MQCONN calls that can be made within a system are dependent on this value, although other factors to consider are disk space availability for the swapper.dat file and shared memory availability.

On MQSeries for Windows, the scope of an MQCONN call is the application process.

Using MQCONN

On MQSeries for VSE/ESA, there is a maximum of 1000 concurrently-connected tasks. The connection handle is unique to the ID of the transaction that is executing and only valid for the duration of that transaction.

Connecting to a queue manager using the MQCONNX call

MQCONNX is not supported on Compaq (DIGITAL) OpenVMS, Tandem NonStop Kernel, and VSE/ESA.

The MQCONNX call is similar to the MQCONN call, but includes options to control the way that the call actually works.

As input to MQCONNX, you must supply a queue manager name, or a queue-sharing group name on OS/390 shared queue systems. The output from MQCONNX is:

- A connection handle
- A completion code
- A reason code

You will need to use the connection handle on subsequent MQI calls.

A description of all of the parameters of MQCONNX is given in the *MQSeries Application Programming Reference* manual. The *Options* field allows you to set STANDARD_BINDING or FASTPATH_BINDING for Version 1 and Version 2 MQCNO:

MQCNO_STANDARD_BINDING

By default, MQCONNX (like MQCONN) implies two logical threads where the MQSeries application and the local queue manager agent run in separate processes. The MQSeries application performs the MQSeries operation and the local queue manager agent performs the application operation. This is defined by the MQCNO_STANDARD_BINDING option on the MQCONNX call.

Note: This default maintains the integrity of the queue manager (that is, it makes the queue manager immune to errant programs), but impairs the performance of the MQI calls.

MQCNO_FASTPATH_BINDING

Trusted applications imply that the MQSeries application and the local queue manager agent become the same process. Since the agent process no longer needs to use an interface to access the queue manager, these applications become an extension of the queue manager. This is defined by the MQCNO_FASTPATH_BINDING option on the MQCONNX call.

You need to link trusted applications to the threaded MQSeries libraries. For instructions on how to set up an MQSeries application to run as trusted, see the *MQSeries Application Programming Reference* manual.

Note: This option compromises the integrity of the queue manager as there is no protection from overwriting its storage. This also applies if the application contains errors which can be exposed to messages and other data in the queue manager too. These issues must be considered before using this option.

For OS/390 these two options are tolerated, but only MQCONN is performed. MQCNO Version 3, for OS/390, allows four new options:

MQCNO_SERIALIZE_CONN_TAG_QSG

This allows an application to request that only one instance of an application may be running at any one time in a queue-sharing group. This is achieved by registering the use of a connection tag, whose value is specified or derived by the application. The tag is a 128 byte character string specified in the Version 3 MQCNO.

MQCNO_RESTRICT_CONN_TAG_QSG

This is used where an application consists of more than one process (or a TCB on OS/390), each of which can connect to a queue manager. Connection is permitted only if there is no current use of the tag, or the requesting application is within the same processing scope. For OS/390 this is MVS address space within the same queue-sharing group as the tag owner.

MQCNO_SERIALIZE_CONN_TAG_Q_MGR

This is similar to MQCNO_SERIALIZE_CONN_TAG_QSG, but only the local queue manager is interrogated to see if the requested tag is already in use.

MQCNO_RESTRICT_CONN_TAG_Q_MGR

This is similar to MQCNO_RESTRICT_CONN_TAG_QSG, but only the local queue manager is interrogated to see if the requested tag is already in use.

Restrictions

The following restrictions apply to trusted applications:

- On MQSeries on UNIX systems, it is necessary to use mqm as the effective userID and groupID for all MQI calls. You may change these IDs before making a non-MQI call requiring authentication (for example, opening a file), but you *must* change it back to mqm before making the next MQI call.
- On MQSeries on UNIX systems, trusted applications must run in threaded processes but only one thread can be connected at a time.
- On MQSeries for AS/400 trusted applications must be run under the QMQM user profile. It is not sufficient that the user profile be member of the QMQM group or that the program adopt QMQM authority. It may not be possible, or desirable, for the QMQM user profile to be used to sign on to interactive jobs, or be specified in the job description for jobs running trusted applications. In this case one approach is to use the OS/400[®] profile swapping API functions, QSYGETPH, QWTSETP and QSYRLSPH to temporarily change the current user of the job to QMQM while the MQ programs run. Details of these functions together with an example of their use is provided in the Security APIs section of the AS/400 System API Reference.
- On MQSeries for OS/2 Warp and MQSeries for Windows NT, a thread within a trusted application cannot connect to a queue manager while another thread in the same process is connected to a different queue manager.
- You must explicitly disconnect trusted applications from the queue manager.
- You must stop trusted applications before ending the queue manager with the endmqm command.
- You must not use asynchronous signals and timer interrupts (such as sigkill) with MQCNO_FASTPATH_BINDING.

Using MQCONN

- On MQSeries for AS/400 trusted applications must not be cancelled through the use of System-Request Option 2, or by the jobs in which they are running being ended using ENDJOB.
- On MQSeries for AIX, trusted applications cannot be compiled using the PL/I programming language.
- On MQSeries for AIX, there are restrictions on the use of shared memory segments:

MQSeries uses a single “shmat()” command to connect to shared memory resources. However, on AIX, one process cannot attach to more than 10 memory segments.

MQSeries uses two additional shared memory segments for trusted applications, reducing the amount of shared storage available. Therefore, it is important that your applications do not connect to too many shared segments, causing a failure in the application code.

Here is a breakdown of the memory segments:

Segment	Use	
0	Reserved for AIX	
1	Reserved for AIX	
2	Stack and heap	
3	CICS	MQSeries (trusted applications only)
4	DB2 [®] and DT/6000	
5		
6		
7		MQSeries (trusted applications only)
8		MQSeries
9	CICS	
A	CICS	
B	CICS	
C	DB2	
D	Reserved for AIX	
E	Reserved for AIX	
F	Reserved for AIX	

This also implies that trusted applications cannot use the maxdata binder option to specify a greater user data area: this conflicts with the queue manager use of shared memory within the application process as it causes the program data to be placed in shared memory segment 3.

Environment variable

On MQSeries for AS/400, MQSeries for OS/2 Warp, MQSeries for Windows NT, and MQSeries on UNIX systems, the environment variable, `MQ_CONNECT_TYPE`, can be used in combination with the type of binding specified in the *Options* field. This environment variable allows you to execute the application with the `STANDARD_BINDING` if any problems occur with the `FASTPATH_BINDING`. If the environment variable is specified, it should have the value `FASTPATH` or `STANDARD` to select the type of binding required. However, the `FASTPATH` binding is used only if the connect option is appropriately specified as shown in Table 2 on page 95:

Table 2. Environment variable

MQCONN	Environment variable	Result
STANDARD	UNDEFINED	STANDARD
FASTPATH	UNDEFINED	FASTPATH
STANDARD	STANDARD	STANDARD
FASTPATH	STANDARD	STANDARD
STANDARD	FASTPATH	STANDARD
FASTPATH	FASTPATH	FASTPATH

So, to run a trusted application, either:

1. Specify the MQCNO_FASTPATH_BINDING option on the MQCONN call and the FASTPATH environment variable,

or

2. Specify the MQCNO_FASTPATH_BINDING option on the MQCONN call and leave the environment variable undefined.

If neither MQCNO_STANDARD_BINDING nor MQCNO_FASTPATH_BINDING is specified, you can use MQCNO_NONE, which defaults to MQCNO_STANDARD_BINDING.

Disconnecting programs from a queue manager using MQDISC

When a program that has connected to a queue manager using the MQCONN or MQCONN call has finished all interaction with the queue manager, it should break the connection using the MQDISC call. However, note that

- On CICS Transaction Server for OS/390 applications, the call is optional unless MQCONN was used and you want to drop the connection tag before the application ends.
- On MQSeries for AS/400, when you sign off from the operating system, an implicit MQDISC call is made.
- On MQSeries for VSE/ESA, if your application does not issue the MQDISC call explicitly, the MQSeries for VSE/ESA housekeeping routine issues the MQDISC call on its behalf and unwanted messages appear in the SYSTEM.LOG queue.

As input to the MQDISC call, you must supply the connection handle (Hconn) that was returned by MQCONN or MQCONN when you connected to the queue manager.

Except on CICS, after MQDISC is called the connection handle (Hconn) is no longer valid, and you cannot issue any further MQI calls until you call MQCONN or MQCONN again. MQDISC does an implicit MQCLOSE for any objects that are still open using this handle.

If MQCONN was used to connect on MQSeries for OS/390, MQDISC also ends the scope of the connection tag established by the MQCONN. However, in a CICS, IMS, or RRS application, if there is an active unit of recovery associated with a connection tag, then the MQDISC is rejected with a reason code of MQRC_CONN_TAG_NOT_RELEASED.

Using MQDISC

| When a program that has connected to a queue manager using the MQCONN or
| MQCONNX call has finished all interaction with the queue manager, it should
| break the connection using the MQDISC call.

Descriptions of the parameters are given in the description of the MQDISC call in the *MQSeries Application Programming Reference* manual.

Authority checking

The MQCLOSE and MQDISC calls usually perform no authority checking. In the normal course of events a job which has the authority to open or connect to an MQSeries object will close or disconnect from that object. Even if the authority of a job that has connected to, or opened an MQSeries object is revoked, the MQCLOSE and MQDISC calls are accepted.

Chapter 8. Opening and closing objects

To perform any of the following operations, you must first *open* the relevant MQSeries object:

- Put messages on a queue
- Get (browse or retrieve) messages from a queue
- Set the attributes of an object
- Inquire about the attributes of any object

Use the MQOPEN call to open the object, using the options of the call to specify what you want to do with the object. The only exception is if you want to put a single message on a queue, then close the queue immediately. In this case, you can bypass the “opening” stage by using the MQPUT1 call (see “Putting one message on a queue using the MQPUT1 call” on page 113).

Before you open an object using the MQOPEN call, you must connect your program to a queue manager. This is explained in detail, for all environments, in “Chapter 7. Connecting and disconnecting a queue manager” on page 89.

There are four types of MQSeries object that can be opened:

- Queue
- Namelist (MQSeries Version 5 products only)
- Process definition
- Queue manager

You open all of these objects in a similar way using the MQOPEN call. For more information about MQSeries objects, see “Chapter 4. MQSeries objects” on page 39.

You can open the same object more than once, and each time you get a new object handle. You might want to browse messages on a queue using one handle, and remove messages from the same queue using another handle. This saves using up resources to close and reopen the same object. You can also open a queue for browsing *and* removing messages at the same time.

Moreover, you can open multiple objects with a single MQOPEN and close them using MQCLOSE. See “Distribution lists” on page 115 for information about how to do this.

When you attempt to open an object, the queue manager checks that you are authorized to open that object for the options you specify in the MQOPEN call.

Objects are closed automatically when a program disconnects from the queue manager. In the IMS environment, disconnection is forced when a program starts processing for a new user following a GU (get unique) IMS call. On the AS/400 platform, objects are closed automatically when a job ends.

It is good programming practice to close objects you have opened. Use the MQCLOSE call to do this.

This chapter introduces opening and closing MQSeries objects, under these headings:

- “Opening objects using the MQOPEN call” on page 98
- “Creating dynamic queues” on page 104

Opening and closing

- “Opening remote queues” on page 105
- “Closing objects using the MQCLOSE call” on page 105

Opening objects using the MQOPEN call

As input to the MQOPEN call, you must supply:

- A connection handle. For CICS applications, you can specify the constant MQHC_DEF_HCONN (which has the value zero), or use the connection handle returned by the MQCONN or MQCONNEX call. For other programs, always use the connection handle returned by the MQCONN or MQCONNEX call.
- A description of the object you want to open, using the object descriptor structure (MQOD).
- One or more options that control the action of the call.

The output from MQOPEN is:

- An object handle that represents your access to the object. Use this on input to any subsequent MQI calls.
- A modified object-descriptor structure, if you are creating a dynamic queue (and it is supported on your platform).
- A completion code.
- A reason code.

Namelists can be opened only on AIX, HP-UX, OS/2 Warp, OS/390, OS/400, Sun Solaris, Compaq Tru64 UNIX, and Windows NT.

Scope of an object handle

The scope of an object handle is the same as the scope of a connection handle, however there are variations between platforms:

CICS In a CICS program, you can use the handle only within the same CICS task from which you made the MQOPEN call.

IMS and OS/390 batch

In the IMS and batch environments, you can use the handle within the same task, but not within any subtasks.

MQSeries for AS/400

In an MQSeries for AS/400 program, you can use the handle only within the same job from which you made the MQOPEN call.

MQSeries for OS/2 Warp

In the MQSeries for OS/2 Warp environment, you can use the same handle within the same thread.

MQSeries for Windows NT

In the MQSeries for Windows NT environment, you can use the same handle within the same thread.

MQSeries on Tandem NonStop Kernel

In this environment, you can use the same handle within the same process.

MQSeries on UNIX systems

In these environments, you can use the same handle within the same thread.

DOS In the DOS environment, there are no restrictions on where you can use the handle.

MQSeries for VSE/ESA

In the VSE/ESA environment, you can use the handle only within the same application transaction from which you made the MQOPEN call.

Windows 3.1

In the Windows 3.1 environment, you can use the handle in the same Windows 3.1 instance.

Descriptions of the parameters of the MQOPEN call are given in the *MQSeries Application Programming Reference* manual.

The following sections describe the information you must supply as input to MQOPEN.

Identifying objects (the MQOD structure)

Use the MQOD structure to identify the object you want to open. This structure is an input parameter for the MQOPEN call. (The structure is modified by the queue manager when you use the MQOPEN call to create a dynamic queue.)

For full details of the MQOD structure see the *MQSeries Application Programming Reference* manual.

For information about using the MQOD structure for distribution lists, see Using the MQOD structure under “Distribution lists” on page 115.

Name resolution

Note: A Queue manager alias is a remote queue definition without an RNAME field.

When you open an MQSeries queue, the MQOPEN call performs a name resolution function on the queue name you specify. This determines on which queue the queue manager performs subsequent operations. This means that when you specify the name of an alias queue or a remote queue in your object descriptor (MQOD), the call resolves the name either to a local queue or to a transmission queue. If a queue is opened for any type of input, browse, or set, it resolves to a local queue if there is one, and fails otherwise. It resolves to a nonlocal queue only if it is opened for output only, inquire only, or output and inquire only. See Table 3 on page 100 for an overview of the name resolution process. Note that the name you supply in *ObjectQMgrName* is resolved *before* that in *ObjectName*.

Table 3 on page 100 also shows how you can use a local definition of a remote queue to define an alias for the name of a queue manager. This allows you to select which transmission queue is used when you put messages on a remote queue, so you could, for example, use a single transmission queue for messages destined for many remote queue managers.

To use the following table, first read down the two left-hand columns, under the heading ‘Input to MQOD’, and select the appropriate case. Then read across the corresponding row, following any instructions. Following the instructions in the ‘Resolved names’ columns, you can either return to the ‘Input to MQOD’ columns and insert values as directed, or you can exit the table with the results supplied. For example, you may be required to input *ObjectName*.

Using MQOPEN

Table 3. Resolving queue names when using MQOPEN

Input to MQOD		Resolved names		
<i>ObjectQMgrName</i>	<i>ObjectName</i>	<i>ObjectQMgrName</i>	<i>ObjectName</i>	Transmission queue
Blank or local queue manager	Local queue with no CLUSTER attribute	Local queue manager	Input <i>ObjectName</i>	Not applicable (local queue used)
Blank queue manager	Local queue with CLUSTER attribute	Workload management selected cluster queue manager or specific cluster queue manager selected on PUT	Input <i>ObjectName</i>	SYSTEM.CLUSTER. TRANSMIT.QUEUE and local queue used SYSTEM.QSG. TRANSMIT.QUEUE (see note)
Local queue manager	Local queue with CLUSTER attribute	Local queue manager	Input <i>ObjectName</i>	Not applicable (local queue used)
Blank or local queue manager	Model queue	Local queue manager	Generated name	Not applicable (local queue used)
Blank or local queue manager	Alias queue with or without CLUSTER attribute	Perform name resolution again with <i>ObjectQMgrName</i> unchanged, and input <i>ObjectName</i> set to the <i>BaseQName</i> in the alias queue definition object. Must not resolve to an alias queue		
Blank or local queue manager	Local definition of a remote queue with or without CLUSTER attribute	Perform name resolution again with <i>ObjectQMgrName</i> set to <i>RemoteQMgrName</i> , and <i>ObjectName</i> set to <i>RemoteQName</i> . Must not resolve remote queues		Name of <i>XmitQName</i> attribute, if non-blank; otherwise <i>RemoteQMgrName</i> in the remote queue definition object. SYSTEM.QSG. TRANSMIT.QUEUE (see note)
Blank queue manager	No matching local object; cluster queue found	Workload management selected cluster queue manager or specific cluster queue manager selected on PUT	Input <i>ObjectName</i>	SYSTEM.CLUSTER. TRANSMIT.QUEUE SYSTEM.QSG. TRANSMIT.QUEUE (see note)
Blank or local queue manager	No matching local object; cluster queue not found		Error, queue not found	Not applicable
Name of a local transmission queue	(Not resolved)	Input <i>ObjectQMgrName</i>	Input <i>ObjectName</i>	Input <i>ObjectQMgrName</i> SYSTEM.QSG. TRANSMIT.QUEUE (see note)

Table 3. Resolving queue names when using MQOPEN (continued)

Queue manager alias definition (<i>RemoteQMgrName</i> may be the local queue manager)	(Not resolved, remote queue)	Perform name resolution again with <i>ObjectQMgrName</i> set to <i>RemoteQMgrName</i> . Must not resolve to remote queues	Input <i>ObjectName</i>	Name of <i>XmitQName</i> attribute, if non-blank; otherwise <i>RemoteQMgrName</i> in the remote queue definition object. SYSTEM.QSG. TRANSMIT.QUEUE (see note)
Queue manager is not the name of any local object; cluster queue managers or queue manager alias found	(Not resolved)	<i>ObjectQMgrName</i> or specific cluster queue manager selected on PUT	Input <i>ObjectName</i>	SYSTEM.CLUSTER. TRANSMIT.QUEUE SYSTEM.QSG. TRANSMIT.QUEUE (see note)
Queue manager is not the name of any local object; no cluster objects found	(Not resolved)	Input <i>ObjectQMgrName</i>	Input <i>ObjectName</i>	<i>DefXmitQName</i> attribute of the queue manager where <i>DefXmitQName</i> is supported. SYSTEM.QSG. TRANSMIT.QUEUE (see note)
Note: The SYSTEM.QSG.TRANSMIT.QUEUE is used if local and remote queue managers are in the same queue-sharing group, intra-group queuing is enabled, and small, non-persistent messages are being put.				

Notes:

1. *BaseQName* is the name of the base queue from the definition of the alias queue.
2. *RemoteQName* is the name of the remote queue from the local definition of the remote queue.
3. *RemoteQMgrName* is the name of the remote queue manager from the local definition of the remote queue.
4. *XmitQName* is the name of the transmission queue from the local definition of the remote queue.
5. When using MQSeries for OS/390 queue managers that are part of a queue-sharing group (QSG), the name of the QSG can be used instead of the local queue manager name in Table 3 on page 100.

Opening an alias queue also opens the base queue to which the alias resolves, and opening a remote queue also opens the transmission queue. Therefore you cannot delete either the queue you specify or the queue to which it resolves while the other one is open.

The resolved queue name and the resolved queue manager name are stored in the *ResolvedQName* and *ResolvedQMgrName* fields in the MQOD.

For more information about name resolution in a distributed queuing environment see the *MQSeries Intercommunication* book.

Using the options of the MQOPEN call

In the *Options* parameter of the MQOPEN call, you must choose one or more options to control the access you are given to the object you are opening. With these options you can:

- Open a queue and specify that all messages put to that queue must be directed to the same instance of it
- Open a queue to allow you to put messages on it
- Open a queue to allow you to browse messages on it
- Open a queue to allow you to remove messages from it
- Open an object to allow you to inquire about and set its attributes (but you can set the attributes of queues only)
- Associate context information with a message
- Nominate an alternate user identifier to be used for security checks
- Control the call if the queue manager is in a quiescing state

MQOPEN option for cluster queue

To specify that all messages MQPUT to a queue are to be routed to the same queue manager by the same route use the MQOO_BIND_ON_OPEN option on the MQOPEN call. To specify that a destination is to be selected at MQPUT time, that is, on a message-by-message basis, use the MQOO_BIND_NOT_FIXED option on the MQOPEN call. If you specify neither of these options the default, MQOO_BIND_AS_Q_DEF, is used. In this case the binding used for the queue handle is taken from the *DefBind* queue attribute, which can take the value MQBND_BIND_ON_OPEN or MQBND_BIND_NOT_FIXED. If the queue you open is not a cluster queue the MQOO_BIND_* options are ignored. If you specify the name of the local queue manager in the MQOD the local instance of the cluster queue is selected. If the queue manager name is blank, any instance can be selected. See the *MQSeries Queue Manager Clusters* book for more information.

MQOPEN option for putting messages

To open a queue in order to put messages on it, use the MQOO_OUTPUT option.

MQOPEN option for browsing messages

To open a queue so that you can *browse* the messages on it, use the MQOPEN call with the MQOO_BROWSE option. This creates a *browse cursor* that the queue manager uses to identify the next message on the queue. For more information, see “Browsing messages on a queue” on page 149.

Notes:

1. You cannot browse messages on a remote queue. Therefore you cannot open a remote queue using the MQOO_BROWSE option.
2. You cannot specify this option when opening a distribution list. For further information about distribution lists, see “Distribution lists” on page 115.

MQOPEN options for removing messages

There are three options that control the opening of a queue in order to remove messages from it. You can use only one of them in any MQOPEN call. These options define whether your program has exclusive or shared access to the queue. *Exclusive access* means that, until you close the queue, only you can remove messages from it. If another program attempts to open the queue to remove messages, its MQOPEN call fails. *Shared access* means that more than one program can remove messages from the queue.

The most advisable approach is to accept the type of access that was intended for the queue when the queue was defined. The queue definition involved the setting of the *Shareability* and the *DefInputOpenOption* attributes. To accept this access, use the MQOO_INPUT_AS_Q_DEF option. Refer to Table 4 to see how the setting of these attributes affects the type of access you will be given when you use this option.

Table 4. How queue attributes and options of the MQOPEN call affect access to queues

Queue attributes		Type of access with MQOPEN options		
<i>Shareability</i>	<i>DefInputOpenOption</i>	AS_Q_DEF	SHARED	EXCLUSIVE
SHAREABLE	SHARED	shared	shared	exclusive
SHAREABLE	EXCLUSIVE	exclusive	shared	exclusive
NOT_SHAREABLE*	SHARED*	exclusive	exclusive	exclusive
NOT_SHAREABLE	EXCLUSIVE	exclusive	exclusive	exclusive

Note: * Although you can define a queue to have this combination of attributes, the default input open option is overridden by the shareability attribute.

Alternatively:

- If you know that your application can work successfully even if other programs can remove messages from the queue at the same time, use the MQOO_INPUT_SHARED option. Table 4 shows how, in some cases you will be given exclusive access to the queue, even with this option.
- If you know that your application can work successfully only if other programs are prevented from removing messages from the queue at the same time, use the MQOO_INPUT_EXCLUSIVE option.

Notes:

1. You cannot remove messages from a remote queue. Therefore you cannot open a remote queue using any of the MQOO_INPUT_* options.
2. You cannot specify this option when opening a distribution list. For further information, see “Distribution lists” on page 115.

MQOPEN options for setting and inquiring about attributes

To open a queue so that you can set its attributes, use the MQOO_SET option. You cannot set the attributes of any other type of object (see “Chapter 12. Inquiring about and setting object attributes” on page 173).

To open an object so that you can inquire about its attributes, use the MQOO_INQUIRE option.

Note: You cannot specify this option when opening a distribution list.

MQOPEN options relating to message context

If you want to be able to associate context information with a message when you put it on a queue, you must use one of the message context options when you open the queue.

The options allow you to differentiate between context information that relates to the *user* who originated the message, and that which relates to the *application* that originated the message. Also, you can opt to set the context information when you put the message on the queue, or you can opt to have the context taken automatically from another queue handle.

Using MQOPEN

For more information about the subject of message context, see “Message context” on page 36.

MQOPEN option for alternate user authority

The MQOPEN option for alternate user authority is not supported on MQSeries for Windows.

When you attempt to open an object using the MQOPEN call, the queue manager checks that you have the authority to open that object. If you are not authorized, the call fails.

However, server programs may want the queue manager to check the authorization of the user on whose behalf they are working, rather than the server’s own authorization. To do this, they must use the MQOO_ALTERNATE_USER_AUTHORITY option of the MQOPEN call, and specify the alternate user ID in the *AlternateUserId* field of the MQOD structure. Typically, the server would get the user ID from the context information in the message it is processing.

MQOPEN option for queue manager quiescing

This is not supported on MQSeries for Windows.

In the CICS environment, if you use the MQOPEN call when the queue manager is in a quiescing state, the call always fails. In other OS/390 environments, AS/400, OS/2, Windows NT, and in UNIX systems environments, the call fails when the queue manager is quiescing only if you use the MQOO_FAIL_IF QUIESCING option of the MQOPEN call.

Creating dynamic queues

Dynamic queues are supported on MQSeries for AS/400, MQSeries for OS/2 Warp, MQSeries for OS/390, MQSeries for Tandem NonStop Kernel, MQSeries on UNIX systems, and MQSeries for Windows NT only.

You should use a dynamic queue for those cases where you do not need the queue after your application ends. For example, you may want to use a dynamic queue for your “reply-to” queue. You specify the name of the reply-to queue in the *ReplyToQ* field of the MQMD structure when you put a message on a queue (see “Defining messages using the MQMD structure” on page 108).

To create a dynamic queue, you use a template known as a model queue, together with the MQOPEN call. You create a model queue using the MQSeries commands or the operations and control panels. The dynamic queue you create takes the attributes of the model queue.

When you call MQOPEN, specify the name of the model queue in the *ObjectName* field of the MQOD structure. When the call completes, the *ObjectName* field is set to the name of the dynamic queue that is created. Also, the *ObjectQMGrName* field is set to the name of the local queue manager.

There are three ways to specify the name of the dynamic queue you create:

- Give the full name you want in the *DynamicQName* field of the MQOD structure.
- Specify a prefix (fewer than 33 characters) for the name, and allow the queue manager to generate the rest of the name. This means that the queue manager generates a unique name, but you still have some control (for example, you may want each user to use a certain prefix, or you may want to give a special

Creating dynamic queues

security classification to queues with a certain prefix in their name). To use this method, specify an asterisk (*) for the last non-blank character of the *DynamicQName* field. Do not specify a single asterisk (*) for the dynamic queue name.

- Allow the queue manager to generate the full name. To use this method, specify an asterisk (*) in the first character position of the *DynamicQName* field.

For more information about these methods, see the description of the *DynamicQName* field in the *MQSeries Application Programming Reference* manual.

There is more information on dynamic queues in “Dynamic queues” on page 45.

Opening remote queues

A remote queue is a queue owned by a queue manager other than the one to which the application is connected.

To open a remote queue, use the MQOPEN call as for a local queue, but there are two ways you can specify the name of the queue:

1. In the *ObjectName* field of the MQOD structure, specify the name of the remote queue as known to the *local* queue manager.
2. In the *ObjectName* field of the MQOD structure, specify the name of the remote queue, as known to the *remote* queue manager. In the *ObjectQMGrName* field, specify either:
 - The name of the transmission queue that has the same name as the remote queue manager.
 - The name of an alias queue object that resolves to the transmission queue that has the same name as the remote queue manager.

This tells the queue manager the destination of the message as well as the transmission queue it needs to be put on to get there.

3. If *DefXmitQname* is supported, in the *ObjectName* field of the MQOD structure, specify the name of the remote queue as known by the *remote* queue manager.

Only local names are validated when you call MQOPEN; the last check is for the existence of the transmission queue to be used.

These three methods are summarized in Table 3 on page 100.

Closing objects using the MQCLOSE call

To close an object, you use the MQCLOSE call. If the object is a queue, you should note the following:

- There is no need to empty a temporary dynamic queue before you close it.
When you close a temporary dynamic queue, the queue is deleted, along with any messages that may still be on it. This is true even if there are uncommitted MQGET, MQPUT, or MQPUT1 calls outstanding against the queue.
- In MQSeries for OS/390, if you have any MQGET requests with an MQGMO_SET_SIGNAL option outstanding for that queue, they are canceled.
- If you opened the queue using the MQOO_BROWSE option, your browse cursor is destroyed.

Namelist can be closed only on AIX, HP-UX, OS/2 Warp, OS/390, OS/400, Sun Solaris, Compaq Tru64 UNIX, and Windows NT.

Using MQCLOSE

In MQSeries for VSE/ESA, ensure that your application issues a matching MQCLOSE call for each MQOPEN call. If your application does not issue the MQCLOSE call, the MQSeries for VSE/ESA housekeeping routine issues the MQCLOSE call on its behalf and unwanted messages appear in the SYSTEM.LOG queue.

Closure is unrelated to syncpoint, so you can close queues before or after syncpoint.

As input to the MQCLOSE call, you must supply:

- A connection handle. Use the same connection handle used to open it, or alternatively, for CICS applications, you can specify the constant MQHC_DEF_HCONN (which has the value zero).
- The handle of the object you want to close. Get this from the output of the MQOPEN call.
- MQCO_NONE in the *Options* field (unless you are closing a permanent dynamic queue).
- The control option to determine whether the queue manager should delete the queue even if there are still messages on it (when closing a permanent dynamic queue).

The output from MQCLOSE is:

- A completion code
- A reason code
- The object handle, reset to the value MQHO_UNUSABLE_HOBJ

Descriptions of the parameters of the MQCLOSE call are given in the *MQSeries Application Programming Reference* manual.

Chapter 9. Putting messages on a queue

Use the MQPUT call to put messages on the queue. You can use MQPUT repeatedly to put many messages on the same queue, following the initial MQOPEN call. Call MQCLOSE when you have finished putting all your messages on the queue.

If you want to put a single message on a queue and close the queue immediately afterwards, you can use the MQPUT1 call. MQPUT1 performs the same functions as the following sequence of calls:

- MQOPEN
- MQPUT
- MQCLOSE

Generally however, if you have more than one message to put on the queue, it is more efficient to use the MQPUT call. This depends on the size of the message and the platform you are working on.

This chapter introduces putting messages to a queue, under these headings:

- “Putting messages on a local queue using the MQPUT call”
- “Putting messages on a remote queue” on page 112
- “Controlling context information” on page 112
- “Putting one message on a queue using the MQPUT1 call” on page 113
- “Distribution lists” on page 115
- “Some cases where the put calls fail” on page 120

Putting messages on a local queue using the MQPUT call

As input to the MQPUT call, you must supply:

- A connection handle (HCONN).
- A queue handle (HObj).
- A description of the message you want to put on the queue. This is in the form of a message descriptor structure (MQMD).
- Control information, in the form of a put-message options structure (MQPMO).
- The length of the data contained within the message (MQLONG).
- The message data itself.

The output from the MQPUT call is

- A reason code (MQLONG)
- A completion code (MQLONG)

If the call completes successfully, it also returns your options structure and your message descriptor structure. The call modifies your options structure to show the name of the queue and the queue manager to which the message was sent. If you request that the queue manager generates a unique value for the identifier of the message you are putting (by specifying binary zero in the *MsgId* field of the MQMD structure), the call inserts the value in the *MsgId* field before returning this structure to you. This value must be reset before you issue another MQPUT.

There is a description of the MQPUT call in the *MQSeries Application Programming Reference* manual.

MQPUT to local queue

The following sections describe the information you must supply as input to the MQPUT call.

Specifying handles

For the connection handle (*Hconn*) in CICS on OS/390 applications, you can specify the constant MQHC_DEF_HCONN (which has the value zero), or you can use the connection handle returned by the MQCONN or MQCONNX call. For other applications, always use the connection handle returned by the MQCONN or MQCONNX call.

Whatever environment you are working in, use the same queue handle (*Hobj*) that is returned by the MQOPEN call.

Defining messages using the MQMD structure

The message descriptor structure (MQMD) is an input/output parameter for the MQPUT and MQPUT1 calls. You use it to define the message you are putting on a queue.

If MQPRI_PRIORITY_AS_Q_DEF or MQPER_PERSISTENCE_AS_Q_DEF is specified for the message and the queue is a cluster queue the values used will be those of the queue the MQPUT resolves to. If that queue is disabled for MQPUT the call will fail. See the *MQSeries Queue Manager Clusters* book for more information.

Note: You must reset the *MsgId* and *CorrelId* to null prior to putting a new message in order to ensure they are unique. The values in these fields are returned on a successful MQPUT.

There is an introduction to the message properties that MQMD describes in “Chapter 3. MQSeries messages” on page 23, and there is a description of the structure itself in the *MQSeries Application Programming Reference* manual.

Specifying options using the MQPMO structure

You use the MQPMO (Put Message Option) structure to pass options to the MQPUT and MQPUT1 calls.

The following sections give you help on filling in the fields of this structure. There is a description of the structure in the *MQSeries Application Programming Reference* manual.

The fields of the structure include:

- *StrucId*
- *Version*
- *Options*
- *Context*
- *ResolvedQName*
- *ResolvedQMgrName*

These fields are described below.

StrucId

This identifies the structure as a put-message options structure. This is a 4-character field. Always specify MQPMO_STRUC_ID.

Version

This describes the version number of the structure. The default is

MQPMO_VERSION_1. If you enter MQPMO_VERSION_2, you can use distribution lists (see “Distribution lists” on page 115). If you enter MQPMO_CURRENT_VERSION, your application is set always to use the most recent level.

Options

This controls the following:

- Whether the put operation is included in a unit of work
- How much context information is associated with a message
- Where the context information is taken from
- Whether the call fails if the queue manager is in a quiescing state
- Whether grouping and, or segmentation is allowed
- Generation of a new message identifier and correlation identifier
- The order in which messages and segments are put on a queue

If you leave the *Options* field set to the default value (MQPMO_NONE), the message you put has default context information associated with it.

Also, the way that the call operates with syncpoints is determined by the platform. The syncpoint control default is ‘yes’ in OS/390; for other platforms, it is ‘no’.

Context

This states the name of the queue handle that you want context information to be copied from (if requested in the *Options* field).

For an introduction to message context, see “Message context” on page 36. For information about using the MQPMO structure to control the context information in a message, see “Controlling context information” on page 112.

ResolvedQName

This contains the name (after resolution of any alias name) of the queue that was opened to receive the message. This is an output field.

ResolvedQMgrName

This contains the name (after resolution of any alias name) of the queue manager that owns the queue in *ResolvedQName*. This is an output field.

The MQPMO can also accommodate fields required for distribution lists (see “Distribution lists” on page 115). If you wish to use this facility, Version 2 of the MQPMO structure is used. This includes the following fields:

Version

This field describes the version number of the structure. For distribution lists, you are required to specify MQPMO_VERSION_2.

RecsPresent

This field contains the number queues in the distribution list. That is the number of Put Message Records (MQPMR) and corresponding Response Records (MQRR) present.

The value you enter can be the same as the number of Object Records provided at MQOPEN. However, if the value is less than the number of Object Records provided on the MQOPEN call (or if no Put Message Records are provided), the values of the queues that are not defined are taken from the default values provided by the message descriptor. Also, if the value is greater than the number of Object Records provided, the excess Put Message Records are ignored.

MQPUT to local queue

You are recommended to do one of the following:

- If you want to receive a report or reply from each destination, enter the same value as appears in the MQOR structure and use MQPMRs containing *MsgId* fields. Either initialize these *MsgId* fields to zeros or specify MQPMO_NEW_MSG_ID.

When you have put the message to the queue, *MsgId* values that the queue manager has created become available in the MQPMRs; you can use these to identify which destination is associated with each report or reply.

- If you do not want to receive reports or replies, choose one of the following:
 1. If you want to identify destinations that fail immediately, you may still want to enter the same value in the *RecsPresent* field as appears in the MQOR structure and provide MQRRs to identify these destinations. Do not specify any MQPMRs.
 2. If you do not want to identify failed destinations, enter zero in the *RecsPresent* field and do not provide MQPMRs nor MQRRs.

Note: If you are using MQPUT1, the number of Response Record Pointers and Response Record Offsets must be zero.

For a full description of Put Message Records (MQPMR) and Response Records (MQRR), see the *MQSeries Application Programming Reference* manual.

PutMsgRecFields

This indicates which fields are present in each Put Message Record (MQPMR). For a list of these fields, see “Using the MQPMR structure” on page 119.

PutMsgRecOffset and *PutMsgRecPtr*

Pointers (typically in C) and offsets (typically in COBOL) are used to address the Put Message Records (see “Using the MQPMR structure” on page 119 for an overview of the MQPMR structure).

Use the *PutMsgRecPtr* field to specify a pointer to the first Put Message Record, or the *PutMsgRecOffset* field to specify the offset of the first Put Message Record. This is the offset from the start of the MQPMO. Depending on the *PutMsgRecFields* field, enter a nonnull value for either *PutMsgRecOffset* or *PutMsgRecPtr*.

ResponseRecOffset and *ResponseRecPtr*

You also use pointers and offsets to address the Response Records (see “Using the MQRR structure” on page 118 for further information about Response Records).

Use the *ResponseRecPtr* field to specify a pointer to the first Response Record, or the *ResponseRecOffset* field to specify the offset of the first Response Record. This is the offset from the start of the MQPMO structure. Enter a nonnull value for either *ResponseRecOffset* or *ResponseRecPtr*.

Note: If you are using MQPUT1 to put messages to a distribution list, *ResponseRecPtr* must be null or zero and *ResponseRecOffset* must be zero.

Additional information for putting to a distribution list (see “Distribution lists” on page 115) is provided in Version 2 of the Put Message Option structure (MQPMR). This is described in the *MQSeries Application Programming Reference* manual.

The data in your message

Give the address of the buffer that contains your data in the *Buffer* parameter of the MQPUT call. You can include anything in the data in your messages. The amount of data in the messages, however, affects the performance of the application that is processing them.

The maximum size of the data is determined by:

- The *MaxMsgLength* attribute of the queue manager
- The *MaxMsgLength* attribute of the queue on which you are putting the message
- The size of any message header added by MQSeries (including the Dead-letter header, MQDLH and the Distribution list header, MQDH)

The *MaxMsgLength* attribute of the queue manager holds the size of message that the queue manager can process. This has a default of 4 MB (1 MB=1048576 bytes), or 100 MB for all MQSeries Version 5 products.

To determine the value of this attribute, use the MQINQ call on the queue manager object. For large messages, you can change this value.

The *MaxMsgLength* attribute of a queue determines the maximum size of message you can put on the queue. If you attempt to put a message with a size larger than the value of this attribute, your MQPUT call fails. If you are putting a message on a remote queue, the maximum size of message that you can successfully put is determined by the *MaxMsgLength* attribute of the remote queue, of any intermediate transmission queues that the message is put on along the route to its destination, and of the channels used.

For an MQPUT operation, the size of the message must be smaller than or equal to the *MaxMsgLength* attribute of both the queue and the queue manager. The values of these attributes are independent, but you are recommended to set the *MaxMsgLength* of the queue to a value less than or equal to that of the queue manager.

MQSeries adds header information to messages in the following circumstances:

- When you put a message on a remote queue, MQSeries adds a transmission header, MQXQH, structure to the message. This structure includes the name of the destination queue and its owning queue manager.
- If MQSeries cannot deliver a message to a remote queue, it attempts to put the message on the dead-letter (undelivered-message) queue. It adds an MQDLH structure to the message. This structure includes the name of the destination queue and the reason the message was put on the dead-letter (undelivered-message) queue.
- If you want to send a message to multiple destination queues, MQSeries adds an MQDH header to the message. This describes the data that is present in a message, belonging to a distribution list, on a transmission queue. This point should be considered when choosing an optimum value for the maximum message length.

These structures are described in the *MQSeries Application Programming Reference* manual.

MQPUT to local queue

If your messages are of the maximum size allowed for these queues, the addition of these headers means that the put operations fail because the messages are now too big. To reduce the possibility of the put operations failing:

- Make the size of your messages smaller than the *MaxMsgLength* attribute of the transmission and dead-letter (undelivered-message) queues. Allow at least the value of the `MQ_MSG_HEADER_LENGTH` constant (more for large distribution lists).
- Make sure that the *MaxMsgLength* attribute of the dead-letter (undelivered-message) queue is set to the same as the *MaxMsgLength* of the queue manager that owns the dead-letter queue.

The attributes for the queue manager and the message queuing constants are described in the *MQSeries Application Programming Reference* manual.

For information on how undelivered messages are handled in a distributed queuing environment, see the *MQSeries Intercommunication* book.

Putting messages on a remote queue

When you want to put a message on a remote queue (that is, a queue owned by a queue manager other than the one to which your application is connected) rather than a local queue, the only extra consideration is how you specify the name of the queue when you open it. This is described in “Opening remote queues” on page 105. There is no change to how you use the MQPUT or MQPUT1 call for a local queue.

For more information on using remote and transmission queues, see the *MQSeries Intercommunication* book.

Controlling context information

To control context information, you use the *Options* field in the MQPMO structure.

If you don't, the queue manager will overwrite context information that may already be in the message descriptor with the identity and context information it has generated for your message. This is the same as specifying the `MQPMO_DEFAULT_CONTEXT` option. You may want this default context information when you create a new message (for example, when processing user input from an inquiry screen).

If you want no context information associated with your message, use the `MQPMO_NO_CONTEXT` option.

Passing identity context

In general, programs should pass identity context information from message to message around an application until the data reaches its final destination. Programs should change the origin context information each time they change the data. However, applications that want to change or set any context information must have the appropriate level of authority. The queue manager checks this authority when the applications open the queues; they must have authority to use the appropriate context options for the MQOPEN call.

If your application gets a message, processes the data from the message, then puts the changed data into another message (possibly for processing by another

application), the application should pass the identity context information from the original message to the new message. You can allow the queue manager to create the origin context information.

To save the context information from the original message, you must use the `MQOO_SAVE_ALL_CONTEXT` option when you open the queue for getting the message. This is in addition to any other options you use with the `MQOPEN` call. Note, however, that you cannot save context information if you only browse the message.

When you create the second message, you must:

- Open the queue using the `MQOO_PASS_IDENTITY_CONTEXT` option (in addition to the `MQOO_OUTPUT` option).
- In the *Context* field of the put-message options structure, give the handle of the queue from which you saved the context information.
- In the *Options* field of the put-message options structure, specify the `MQPMO_PASS_IDENTITY_CONTEXT` option.

Passing all context

If your application gets a message, and puts the message data (unchanged) into another message, the application should pass both the identity and the origin context information from the original message to the new message. An example of an application that might do this is a message mover, which moves messages from one queue to another.

Follow the same procedure as for passing identity context, except you use the `MQOPEN` option `MQOO_PASS_ALL_CONTEXT` and the put-message option `MQPMO_PASS_ALL_CONTEXT`.

Setting identity context

If you want to set the identity context information for a message, leaving the queue manager to set the origin context information:

- Open the queue using the `MQOO_SET_IDENTITY_CONTEXT` option.
- Put the message on the queue, specifying the `MQPMO_SET_IDENTITY_CONTEXT` option. In the message descriptor, specify whatever identity context information you require.

Setting all context

If you want to set both the identity and the origin context information for a message:

- Open the queue using the `MQOO_SET_ALL_CONTEXT` option.
- Put the message on the queue, specifying the `MQPMO_SET_ALL_CONTEXT` option. In the message descriptor, specify whatever identity and origin context information you require.

Appropriate authority is needed for each type of context setting.

Putting one message on a queue using the MQPUT1 call

Use the `MQPUT1` call when you want to close the queue immediately after you have put a single message on it. For example, a server application is likely to use the `MQPUT1` call when it is sending a reply to each of the different queues.

Using MQPUT1

MQPUT1 is functionally equivalent to calling MQOPEN followed by MQPUT, followed by MQCLOSE. The only difference in the syntax for the MQPUT and MQPUT1 calls is that for MQPUT you must specify an object handle, whereas for MQPUT1 you must specify an object descriptor structure (MQOD) as defined in MQOPEN (see “Identifying objects (the MQOD structure)” on page 99). This is because you need to give information to the MQPUT1 call about the queue it has to open, whereas when you call MQPUT, the queue must already be open.

As input to the MQPUT1 call, you must supply:

- A connection handle.
- A description of the object you want to open. This is in the form of an object descriptor structure (MQOD).
- A description of the message you want to put on the queue. This is in the form of a message descriptor structure (MQMD).
- Control information in the form of a put-message options structure (MQPMO).
- The length of the data contained within the message (MQLONG).
- The address of the message data.

The output from MQPUT1 is:

- A completion code
- A reason code

If the call completes successfully, it also returns your options structure and your message descriptor structure. The call modifies your options structure to show the name of the queue and the queue manager to which the message was sent. If you request that the queue manager generate a unique value for the identifier of the message you are putting (by specifying binary zero in the *MsgId* field of the MQMD structure), the call inserts the value in the *MsgId* field before returning this structure to you.

Note: You cannot use MQPUT1 with a model queue name; however, once a model queue has been opened, you can issue an MQPUT1 to the dynamic queue.

The six input parameters for MQPUT1 are:

Hconn This is a connection handle. For CICS applications, you can specify the constant MQHC_DEF_HCONN (which has the value zero), or use the connection handle returned by the MQCONN or MQCONNX call. For other programs, always use the connection handle returned by the MQCONN or MQCONNX call.

ObjDesc

This is an object descriptor structure (MQOD).

In the *ObjectName* and *ObjectQMGrName* fields, give the name of the queue on which you want to put a message, and the name of the queue manager that owns this queue.

The *DynamicQName* field is ignored for the MQPUT1 call because it cannot use model queues.

Use the *AlternateUserId* field if you want to nominate an alternate user identifier that is to be used to test authority to open the queue.

MsgDesc

This is a message descriptor structure (MQMD). As with the MQPUT call, use this structure to define the message you are putting on the queue.

PutMsgOpts

This is a put-message options structure (MQPMO). Use it as you would for the MQPUT call (see “Specifying options using the MQPMO structure” on page 108).

When the *Options* field is set to zero, the queue manager uses your own user ID when it performs tests for authority to access the queue. Also, the queue manager ignores any alternate user identifier given in the *AlternateUserId* field of the MQOD structure.

BufferLength

This is the length of your message.

Buffer This is the buffer area that contains the text of your message.

When you use clusters, MQPUT1 operates as though MQOO_BIND_NOT_FIXED is in effect. Applications must use the resolved fields in the MQPMO structure rather than the MQOD structure to determine where the message was sent. See the *MQSeries Queue Manager Clusters* book for more information.

There is a description of the MQPUT1 call in the *MQSeries Application Programming Reference* manual.

Distribution lists

These are supported on MQSeries Version 5 products except MQSeries for OS/390.

Distribution lists allow you to put a message to multiple destinations in a single MQPUT or MQPUT1 call. Multiple queues can be opened using a single MQOPEN and a message can then be put to each of those queues using a single MQPUT. Some generic information from the MQI structures used for this process can be superseded by specific information relating to the individual destinations included in the distribution list.

When an MQOPEN call is issued, generic information is taken from the Object Descriptor (MQOD). If you specify MQOD_VERSION_2 in the *Version* field and a value greater than zero in the *RecsPresent* field, the *Hobj* can be defined as a handle of a list (of one or more queues) rather than of a queue. In this case, specific information is given through the object records (MQORs), which give details of destination (that is, *ObjectName* and *ObjectQMgrName*).

The object handle (*Hobj*) is passed to the MQPUT call, allowing you to put to a list rather than to a single queue.

When a message is put on the queues (MQPUT), generic information is taken from the Put Message Option structure (MQPMO) and the Message Descriptor (MQMD). Specific information is given in the form of Put Message Records (MQPMRs).

Response Records (MQRR) can receive a completion code and reason code specific to each destination queue.

Figure 5 on page 116 shows how distribution lists work.

Distribution lists

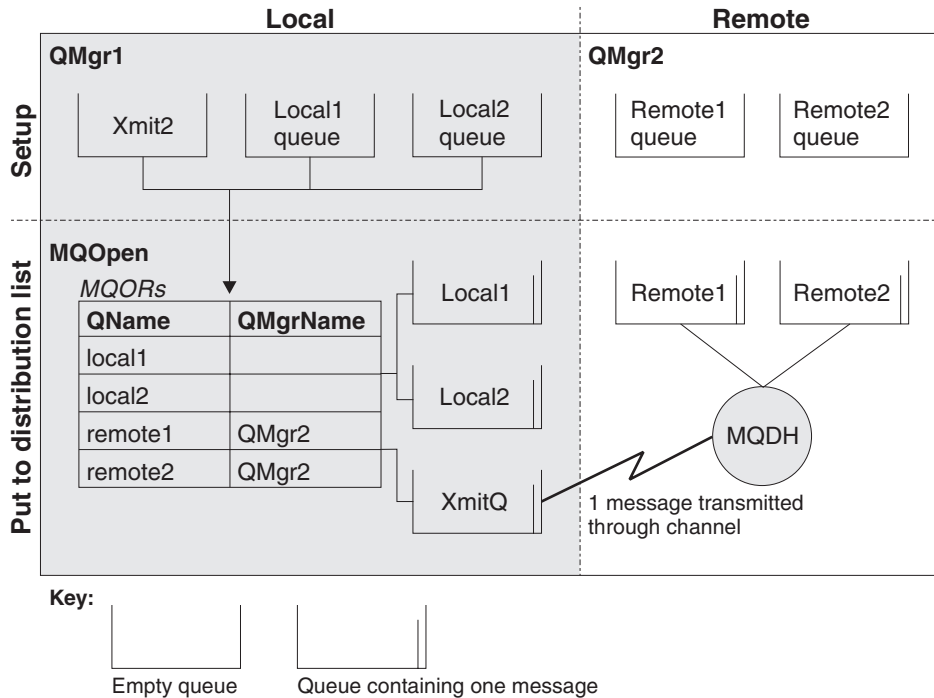


Figure 5. How distribution lists work. This diagram shows that **one** message is transmitted through the channel and can be put on more than one remote queue.

Opening distribution lists

Use the MQOPEN call to open a distribution list, and use the options of the call to specify what you want to do with the list.

As input to MQOPEN, you must supply:

- A connection handle (see "Chapter 9. Putting messages on a queue" on page 107 for a description)
- Generic information in the Object Descriptor structure (MQOD)
- The name of each queue you want to open, using the Object Record structure (MQOR)

The output from MQOPEN is:

- An object handle that represents your access to the distribution list
- A generic completion code
- A generic reason code
- Response Records (optional), containing a completion code and reason for each destination

Using the MQOD structure

Use the MQOD structure to identify the queues you want to open. To define a distribution list, you must specify MQOD_VERSION_2 in the *Version* field, a value greater than zero in the *RecsPresent* field, and MQOT_Q in the *ObjectType* field. See the *MQSeries Application Programming Reference* manual for a description of all the fields of the MQOD structure.

Using the MQOR structure

An MQOR structure must be provided for each destination. The structure contains the destination queue and queue manager names. The *ObjectName* and *ObjectQMGrName* fields in the MQOD are not used for distribution lists. There must

be one or more object records. If the *ObjectQMgrName* is left blank, the local queue manager is used. See the *MQSeries Application Programming Reference* manual for further information about these fields.

You can specify the destination queues in two ways:

- By using the offset field *ObjectRecOffset*.

In this case, the application should declare its own structure containing an MQOD structure, followed by the array of MQOR records (with as many array elements as are needed), and set *ObjectRecOffset* to the offset of the first element in the array from the start of the MQOD. Care must be taken to ensure that this offset is correct.

Use of built-in facilities provided by the programming language is recommended, if these are available in all of the environments in which the application must run. The following illustrates this technique for the COBOL programming language:

```
01 MY-OPEN-DATA.
   02 MY-MQOD.
      COPY CMQODV.
   02 MY-MQOR-TABLE OCCURS 100 TIMES.
      COPY CMQORV.
MOVE LENGTH OF MY-MQOD TO MQOD-OBJECTRECOFFSET.
```

Alternatively, the constant MQOD_CURRENT_LENGTH can be used if the programming language does not support the necessary built-in facilities in all of the environments concerned. The following illustrates this technique:

```
01 MY-MQ-CONSTANTS.
   COPY CMQV.
01 MY-OPEN-DATA.
   02 MY-MQOD.
      COPY CMQODV.
   02 MY-MQOR-TABLE OCCURS 100 TIMES.
      COPY CMQORV.
MOVE MQOD-CURRENT-LENGTH TO MQOD-OBJECTRECOFFSET.
```

However, this will work correctly only if the MQOD structure and the array of MQOR records are contiguous; if the compiler inserts skip bytes between the MQOD and the MQOR array, these must be added to the value stored in *ObjectRecOffset*.

Using *ObjectRecOffset* is recommended for programming languages that do not support the pointer data type, or that implement the pointer data type in a way that is not portable to different environments (for example, the COBOL programming language).

- By using the pointer field *ObjectRecPtr*.

In this case, the application can declare the array of MQOR structures separately from the MQOD structure, and set *ObjectRecPtr* to the address of the array. The following illustrates this technique for the C programming language:

```
MQOD MyMqod;
MQOR MyMqor[100];
MyMqod.ObjectRecPtr = MyMqor;
```

Using *ObjectRecPtr* is recommended for programming languages that support the pointer data type in a way that is portable to different environments (for example, the C programming language).

Distribution lists

Whichever technique is chosen, one of *ObjectRecOffset* and *ObjectRecPtr* must be used; the call fails with reason code MQRC_OBJECT_RECORDS_ERROR if both are zero, or both are nonzero.

Using the MQRR structure

These structures are destination specific as each Response Record contains a *CompCode* and *Reason* field for each queue of a distribution list. You must use this structure to enable you to distinguish where any problems lie.

For example, if you receive a reason code of MQRC_MULTIPLE_REASONS and your distribution list contains five destination queues, you will not know which queues the problems apply to if you do not use this structure. However, if you have a completion code and reason code for each destination, you can locate the errors more easily.

See the *MQSeries Application Programming Reference* manual for further information about the MQRR structure.

Figure 6 shows how you can open a distribution list in C.

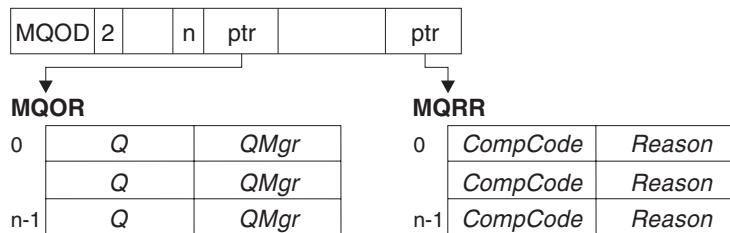


Figure 6. Opening a distribution list in C. The MQOD uses pointers to the MQOR and MQRR structures.

Figure 7 shows how you can open a distribution list in COBOL.

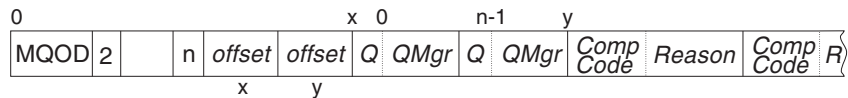


Figure 7. Opening a distribution list in COBOL. The MQOD uses offsets in COBOL.

Using the MQOPEN options

The following options can be specified when opening a distribution list:

- MQOO_OUTPUT
- MQOO_FAIL_IF QUIESCING (optional)
- MQOO_ALTERNATE_USER_AUTHORITY (optional)
- MQOO_*_CONTEXT (optional)

See “Chapter 8. Opening and closing objects” on page 97 for a description of these options.

Putting messages to a distribution list

To put messages to a distribution list, you can use MQPUT or MQPUT1. As input, you must supply:

- A connection handle (see “Chapter 9. Putting messages on a queue” on page 107 for a description).

Putting messages to a distribution list

- An object handle. If a distribution list is opened using MQOPEN, the *Hobj* allows you only to put to the list.
- A message descriptor structure (MQMD). See the *MQSeries Application Programming Reference* manual for a description of this structure.
- Control information in the form of a put-message option structure (MQPMO). See “Specifying options using the MQPMO structure” on page 108 for information about filling in the fields of the MQPMO structure.
- Control information in the form of Put Message Records (MQPMR).
- The length of the data contained within the message (MQLONG).
- The message data itself.

The output is:

- A completion code
- A reason code
- Response Records (optional)

Using the MQPMR structure

This structure is optional and gives destination-specific information for some fields that you may want to identify differently from those already identified in the MQMD. For a description of these fields, see the *MQSeries Application Programming Reference* manual.

The content of each record depends on the information given in the *PutMsgRecFields* field of the MQPMO. For example, in the sample program AMQSPTL0.C (see “The Distribution List sample program” on page 375 for a description) showing the use of distribution lists, the sample chooses to provide values for *MsgId* and *CorrelId* in the MQPMR. This section of the sample program looks like this:

```
typedef struct
{
  MQBYTE24 MsgId;
  MQBYTE24 CorrelId;
} PutMsgRec;...
/*****
MQLONG PutMsgRecFields=MQPMRF_MSG_ID | MQPMRF_CORREL_ID;
```

This implies that *MsgId* and *CorrelId* are provided for each destination of a distribution list. The Put Message Records are provided as an array.

Figure 8 shows how you can put a message to a distribution list in C.

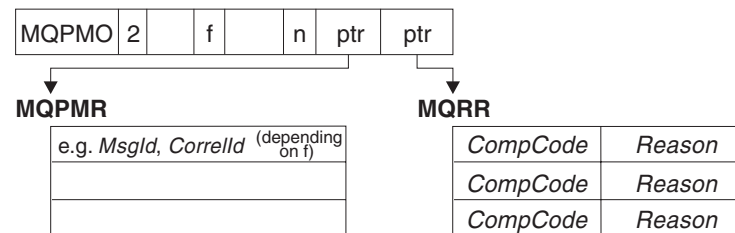


Figure 8. Putting a message to a distribution list in C. The MQPMO uses pointers to the MQPMR and MQRR structures.

Figure 9 on page 120 shows how you can put a message to a distribution list in COBOL.

Putting messages to a distribution list

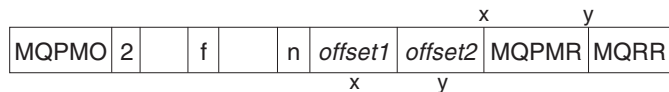


Figure 9. Putting a message to a distribution list in COBOL. The MQPMO uses offsets in COBOL.

Using MQPUT1

If you are using MQPUT1, consider the following:

1. The values of the *ResponseRecOffset* and *ResponseRecPtr* fields must be null or zero.
2. The Response Records, if required, must be addressed from the MQOD.

Some cases where the put calls fail

If certain attributes of a queue are changed using the FORCE option on a command during the interval between you issuing an MQOPEN and an MQPUT call, the MQPUT call fails and returns the MQRC_OBJECT_CHANGED reason code. The queue manager marks the object handle as being no longer valid. This also happens if the changes are made while an MQPUT1 call is being processed, or if the changes apply to any queue to which the queue name resolves. The attributes that affect the handle in this way are listed in the description of the MQOPEN call in the *MQSeries Application Programming Reference* manual. If your call returns the MQRC_OBJECT_CHANGED reason code, close the queue, reopen it, then try to put a message again.

If put operations are inhibited for a queue on which you are attempting to put messages (or any queue to which the queue name resolves), the MQPUT or MQPUT1 call fails and returns the MQRC_PUT_INHIBITED reason code. You may be able to put a message successfully if you attempt the call at a later time, if the design of the application is such that other programs change the attributes of queues regularly.

Further, if the queue that you are trying to put your message on is full, the MQPUT or MQPUT1 call fails and returns MQRC_Q_FULL.

If a dynamic queue (either temporary or permanent) has been deleted, MQPUT calls using a previously acquired object handle fail and return the MQRC_Q_DELETED reason code. In this situation, it is good practice to close the object handle as it is no longer of any use to you.

In the case of distribution lists, multiple completion codes and reason codes can occur in a single request. These cannot be handled using only the *CompCode* and *Reason* output fields on MQOPEN and MQPUT.

When distribution lists are used to put messages to multiple destinations, the Response Records contain the specific *CompCode* and *Reason* for each destination. If you receive a completion code of MQCC_FAILED, no message is put on any destination queue successfully. If the completion code is MQCC_WARNING, the message is successfully put on one or more of the destination queues. If you receive a return code of MQRC_MULTIPLE_REASONS, the reason codes are not all the same for every destination. Therefore, it is recommended to use the MQRR structure so that you can determine which queue or queues caused an error and the reasons for each.

Chapter 10. Getting messages from a queue

You can get messages from a queue in two ways:

1. You can *remove* a message from the queue so that other programs can no longer see it.
2. You can *copy* a message, leaving the original message on the queue. This is known as *browsing*. You can easily remove the message once you have browsed it.

In both cases, you use the MQGET call, but first your application must be connected to the queue manager, and you must use the MQOPEN call to open the queue (for input, browse, or both). These operations are described in “Chapter 7. Connecting and disconnecting a queue manager” on page 89 and “Chapter 8. Opening and closing objects” on page 97.

When you have opened the queue, you can use the MQGET call repeatedly to browse or remove messages on the same queue. Call MQCLOSE when you have finished getting all the messages you want from the queue.

This chapter introduces getting messages from a queue, under these headings:

- “Getting messages from a queue using the MQGET call”
- “The order in which messages are retrieved from a queue” on page 126
- “Getting a particular message” on page 133
- “Type of index” on page 134
- “Handling messages greater than 4 MB long” on page 135
- “Waiting for messages” on page 141
- “Signaling” on page 142
- “Skipping backout” on page 144
- “Application data conversion” on page 147
- “Browsing messages on a queue” on page 149
- “Browsing messages in logical order” on page 151
- “Some cases where the MQGET call fails” on page 154

Getting messages from a queue using the MQGET call

The MQGET call gets a message from an open local queue. It cannot get a message from a queue on another system.

As input to the MQGET call, you must supply:

- A connection handle.
- A queue handle.
- A description of the message you want to get from the queue. This is in the form of a message descriptor (MQMD) structure.
- Control information in the form of a Get Message Options (MQGMO) structure.
- The size of the buffer you have assigned to hold the message (MQLONG).
- The address of the storage in which the message must be put.

The output from MQGET is:

- A reason code
- A completion code

Using MQGET

- The message in the buffer area you specified, if the call completes successfully
- Your options structure, modified to show the name of the queue from which the message was retrieved
- Your message descriptor structure, with the contents of the fields modified to describe the message that was retrieved
- The length of the message (MQLONG)

There is a description of the MQGET call in the *MQSeries Application Programming Reference* manual.

The following sections describe the information you must supply as input to the MQGET call.

Specifying connection handles

For CICS on OS/390 and VSE/ESA applications, you can specify the constant MQHC_DEF_HCONN (which has the value zero), or use the connection handle returned by the MQCONN or MQCONNX call. For other applications, always use the connection handle returned by the MQCONN or MQCONNX call.

Use the queue handle (*Hobj*) that is returned when you call MQOPEN.

Describing messages using the MQMD structure and the MQGET call

To identify the message you want to get from a queue, use the message descriptor structure (MQMD). This is an input/output parameter for the MQGET call. There is an introduction to the message properties that MQMD describes in “Chapter 3. MQSeries messages” on page 23, and there is a description of the structure itself in the *MQSeries Application Programming Reference* manual.

If you know which message you want to get from the queue, see “Getting a particular message” on page 133.

If you do not specify a particular message, MQGET retrieves the *first* message in the queue. “The order in which messages are retrieved from a queue” on page 126 describes how the priority of a message, the *MsgDeliverySequence* attribute of the queue, and the MQGMO_LOGICAL_ORDER option determine the order of the messages in the queue.

Note: If you want to use MQGET more than once (for example, to step through the messages in the queue), you must set the *MsgId* and *CorrelId* fields of this structure to null after each call. This clears these fields of the identifiers of the message that was retrieved.

However, if you want to group your messages, the *GroupId* should be the same for messages in the same group, so that the call will look for a message having the same identifiers as the previous message in order to make up the whole group.

Specifying MQGET options using the MQGMO structure

The MQGMO structure is an input/output variable for passing options to the MQGET call.

The following sections give you help on filling in some of the fields of this structure. There is a description of the structure in the *MQSeries Application Programming Reference* manual.

StrucId

StrucId is a 4-character field used to identify the structure as a get-message options structure. Always specify MQGMO_STRUC_ID.

Version

Version describes the version number of the structure.

MQGMO_VERSION_1 is the default. If you wish to use the Version 2 fields or retrieve messages in logical order, specify MQGMO_VERSION_2. If you wish to use the Version 3 fields or retrieve messages in logical order, specify MQGMO_VERSION_3. MQGMO_CURRENT_VERSION sets your application to use the most recent level.

Options

Within your code, you can select the options in any order as each option is represented by a bit in the *Options* field.

The *Options* field controls:

- Whether the MQGET call waits for a message to arrive on the queue before it completes (see “Waiting for messages” on page 141)
- Whether the get operation is included in a unit of work.
- Whether a nonpersistent message is retrieved outside syncpoint, allowing fast messaging
- In MQSeries for OS/390, whether the message retrieved is marked as skipping backout (see “Skipping backout” on page 144)
- Whether the message is removed from the queue, or merely browsed
- Whether to select a message by using a browse cursor or by other selection criteria
- Whether the call succeeds even if the message is longer than your buffer
- In MQSeries for OS/390, whether to allow the call to complete, but set a signal to indicate that you want to be notified when a message arrives
- Whether the call fails if the queue manager is in a quiescing state
- On OS/390, whether the call fails if the connection is in a quiescing state
- Whether application message data conversion is required (see “Application data conversion” on page 147)
- On MQSeries Version 5 products except MQSeries for OS/390, the order in which messages and segments are retrieved from a queue
- On MQSeries Version 5 products except MQSeries for OS/390, whether complete, logical messages only are retrievable
- On MQSeries Version 5 products except MQSeries for OS/390, whether messages in a group can be retrieved only when *all* messages in the group are available
- On MQSeries Version 5 products except MQSeries for OS/390, whether segments in a logical message can be retrieved only when *all* segments in the logical message are available

If you leave the *Options* field set to the default value (MQGMO_NO_WAIT), the MQGET call operates this way:

- If there is no message matching your selection criteria on the queue, the call does not wait for a message to arrive, but completes immediately.

Using MQGET

Also, in MQSeries for OS/390, the call does not set a signal requesting notification when such a message arrives.

- The way that the call operates with syncpoints is determined by the platform:

Platform	Under syncpoint control
AS/400	No
UNIX systems	No
OS/390	Yes
OS/2	No
Tandem NSK	Yes
VSE/ESA	Yes
Windows NT	No
Windows	No

- In MQSeries for OS/390, the message retrieved is not marked as skipping backout.
- The selected message is removed from the queue (not browsed).
- No application message data conversion is required.
- The call fails if the message is longer than your buffer.

WaitInterval

The *WaitInterval* field specifies the maximum time (in milliseconds) that the MQGET call waits for a message to arrive on the queue when you use the MQGMO_WAIT option. If no message arrives within the time specified in *WaitInterval*, the call completes and returns a reason code showing that there was no message that matched your selection criteria on the queue.

In MQSeries for OS/390, if you use the MQGMO_SET_SIGNAL option, the *WaitInterval* field specifies the time for which the signal is set.

For more information on these options, see “Waiting for messages” on page 141 and “Signaling” on page 142.

Signal1

Signal1 is supported on MQSeries for OS/390, MQSeries for Tandem NonStop Kernel, and MQSeries for Windows Version 2.1 only.

If you have chosen to use the MQGMO_SET_SIGNAL option to request that your application is notified when a suitable message arrives, you must specify the type of signal in the *Signal1* field. In MQSeries on all other platforms, the *Signal1* field is reserved and its value is not significant.

For more information, see “Signaling” on page 142.

Signal2

On MQSeries for Windows Version 2.1 this specifies an identifier for the signal message. The *Signal2* field is reserved on all other platforms and its value is not significant.

For more information, see “Signaling” on page 142.

ResolvedQName

ResolvedQName is an output field in which the queue manager returns the name of the queue (after resolution of any alias) from which the message was retrieved.

MatchOptions

MatchOptions controls the selection criteria for MQGET.

GroupStatus

GroupStatus indicates whether the message you have retrieved is in a group.

SegmentStatus

SegmentStatus indicates whether the item you have retrieved is a segment of a logical message.

Segmentation

Segmentation indicates whether segmentation is allowed for the message retrieved.

MsgToken

MsgToken is supported on MQSeries for OS/390 only.

MsgToken uniquely identifies a message.

For more information, see “MQSeries Workflow” on page 225.

ReturnedLength

ReturnedLength is an output field in which the queue manager returns the length of message data returned (in bytes).

Specifying the size of the buffer area

In the *BufferLength* parameter of the MQGET call, specify the size of the buffer area you want to use to hold the message data that you retrieve. There are three ways to decide how big this should be:

1. You may already know what length of messages to expect from this program. If so, specify a buffer of this size.

However, you can use the MQGMO_ACCEPT_TRUNCATED_MSG option in the MQGMO structure if you want the MQGET call to complete even if the message is too big for the buffer. In this case:

- The buffer is filled with as much of the message as it can hold
- The call returns a warning completion code
- The message is removed from the queue (discarding the remainder of the message), or the browse cursor is advanced (if you are browsing the queue)
- The real length of the message is returned in *DataLength*

Without this option, the call still completes with a warning, but it does not remove the message from the queue (or advance the browse cursor).

2. Estimate a size for the buffer (or even specify a size of zero bytes) and **do not** use the MQGMO_ACCEPT_TRUNCATED_MSG option. If the MQGET call fails (for example, because the buffer is too small), the length of the message is returned in the *DataLength* parameter of the call. (The buffer is still filled with as much of the message as it can hold, but the processing of the call is not completed.) Store the *MsgId* of this message, then repeat the MQGET call, specifying a buffer area of the correct size, and the *MsgId* you noted from the first call.

If your program is serving a queue that is also being served by other programs, one of those other programs may remove the message you want before your program can issue another MQGET call. Your program could waste time searching for a message that no longer exists. To avoid this, first browse the queue until you find the message you want, specifying a *BufferLength* of zero and using the MQGMO_ACCEPT_TRUNCATED_MSG option. This positions

Using MQGET

the browse cursor under the message you want. You can then retrieve the message by calling MQGET again, specifying the MQGMO_MSG_UNDER_CURSOR option. If another program removes the message between your browse and removal calls, your second MQGET fails immediately (without searching the whole queue), because there is no message under your browse cursor.

3. The *MaxMsgLength* queue attribute determines the maximum length of messages accepted for that queue and the *MaxMsgLength* queue manager attribute determines the maximum length of messages accepted for that queue manager. If you do not know what length of message to expect, you can inquire about the *MaxMsgLength* attribute (using the MQINQ call), then specify a buffer of this size.

For further information about the *MaxMsgLength* attribute, see “Increasing the maximum message length” on page 135.

The order in which messages are retrieved from a queue

You have control over the order in which you retrieve messages from a queue. This section looks at the options.

Priority

A program can assign a priority to a message when it puts the message on a queue (see “Message priorities” on page 32). Messages of equal priority are stored in a queue in order of arrival, not the order in which they are committed.

The queue manager maintains queues either in strict FIFO (first in, first out) sequence, or in FIFO within priority sequence. This depends on the setting of the *MsgDeliverySequence* attribute of the queue. When a message arrives on a queue, it is inserted immediately following the last message that has the same priority.

Programs can either get the first message from a queue, or they can get a particular message from a queue, ignoring the priority of those messages. For example, a program may want to process the reply to a particular message that it sent earlier. For more information, see “Getting a particular message” on page 133.

If an application puts a sequence of messages on a queue, another application can retrieve those messages in the same order that they were put, provided:

- The messages all have the same priority
- The messages were all put within the same unit of work, or all put outside a unit of work
- The queue is local to the putting application

If these conditions are not met, and the applications depend on the messages being retrieved in a certain order, the applications must either include sequencing information in the message data, or establish a means of acknowledging receipt of a message before the next one is sent.

On MQSeries for OS/390, the queue attribute, *IndexType*, can be used to increase the speed of MQGET operations on the queue. For more information, see “Type of index” on page 134.

Logical and physical ordering

Logical and physical ordering is supported on MQSeries Version 5 products only, except MQSeries for OS/390.

MQGET retrieval sequence

Messages on queues can occur (within each priority level) in *physical* or *logical* order:

Order Meaning

Physical

This is the order in which messages arrive on a queue.

Logical

This is when all of the messages and segments within a group are in their logical sequence, adjacent to each other, in the position determined by the physical position of the first item belonging to the group.

For a description of groups, messages, and segments, see “Message groups” on page 32. These physical and logical orders may differ because:

- Groups can arrive at a destination at similar times from different applications, therefore losing any distinct physical order.
- Even within a single group, messages may get out of order due to rerouting or delay of some of the messages in the group.

For example, the logical order might look like Figure 10:

These messages would appear in the following logical order on a queue:

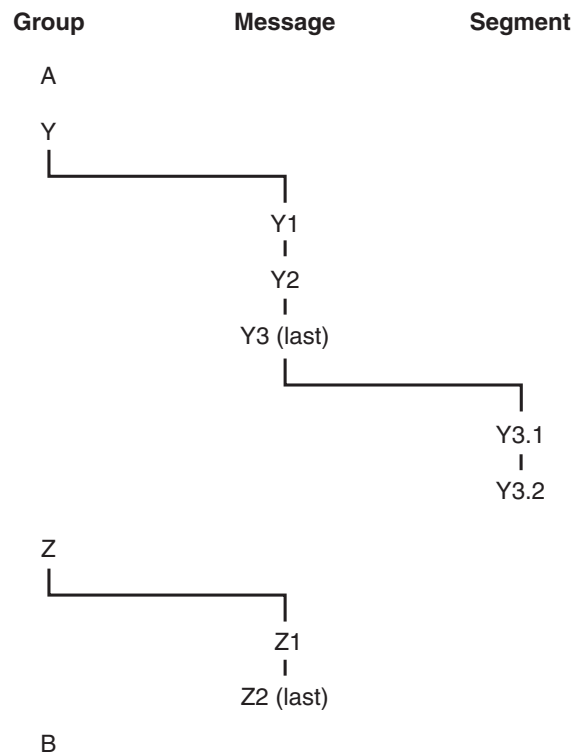


Figure 10. Logical order on a queue

1. Message A (not in a group)
2. Logical message 1 of group Y
3. Logical message 2 of group Y
4. Segment 1 of (last) logical message 3 of group Y
5. (Last) segment 2 of (last) logical message 3 of group Y
6. Logical message 1 of group Z
7. (Last) logical message 2 of group Z
8. Message B (not in a group)

MQGET retrieval sequence

The physical order, however, might be entirely different. As stated on page 126, the physical position of the *first* item within each group determines the logical position of the whole group. For example, if groups Y and Z arrived at similar times, and message 2 of group Z overtook message 1 of the same group, the physical order would look like Figure 11:

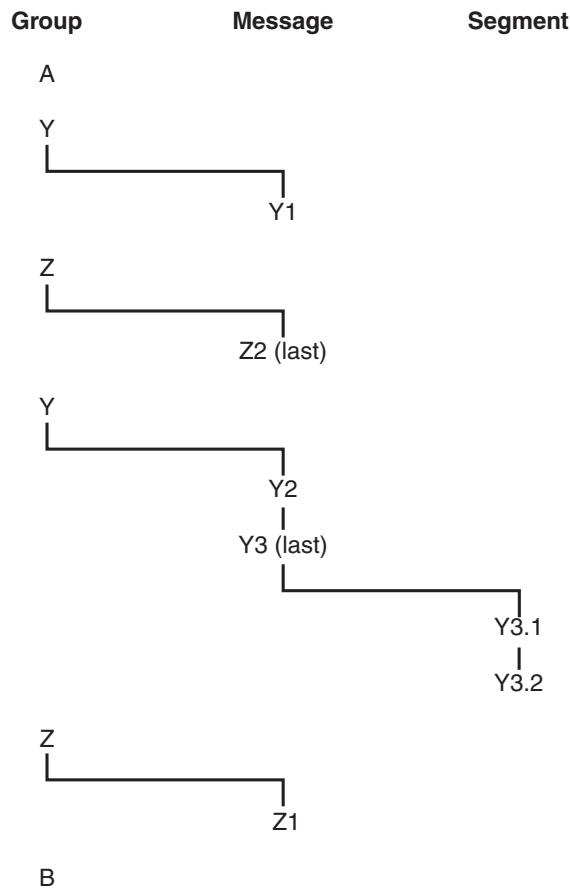


Figure 11. Physical order on a queue

These messages appear in the following logical order on the queue:

1. Message A (not in a group)
2. Logical message 1 of group Y
3. Logical message 2 of group Z
4. Logical message 2 of group Y
5. Segment 1 of (last) logical message 3 of group Y
6. (Last) segment 2 of (last) logical message 3 of group Y
7. Logical message 1 of group Z
8. Message B (not in a group)

When getting messages, you can specify `MQGMO_LOGICAL_ORDER` to retrieve messages in logical rather than physical order.

If you issue an `MQGET` call with `MQGMO_BROWSE_FIRST` and `MQGMO_LOGICAL_ORDER`, subsequent `MQGET` calls with `MQGMO_BROWSE_NEXT` must also specify this option. Conversely, if the `MQGET` with `MQGMO_BROWSE_FIRST` does not specify `MQGMO_LOGICAL_ORDER`, neither must the following `MQGETs` with `MQGMO_BROWSE_NEXT`.

MQGET retrieval sequence

The group and segment information that the queue manager retains for MQGET calls that browse messages on the queue is separate from the group and segment information that the queue manager retains for MQGET calls that remove messages from the queue. When MQGMO_BROWSE_FIRST is specified, the queue manager ignores the group and segment information for browsing, and scans the queue as though there were no current group and no current logical message.

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

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

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

For further information about this, see the *MQSeries Application Programming Reference* manual.

For most applications you will probably choose either logical or physical ordering when browsing. However, if you want to switch between these modes, remember that when you first issue a browse with MQGMO_LOGICAL_ORDER, your position within the logical sequence is established.

If the first item within the group is not present at this time, the group you are in is not considered to be part of the logical sequence.

Once the browse cursor is within a group, it can continue within the same group, even if the first message is removed. Initially though, you can never move into a group using MQGMO_LOGICAL_ORDER where the first item is not present.

Grouping logical messages

There are two main reasons for using logical messages in a group:

- The messages may need to be processed in the correct order
- Each of the messages in a group may need to be processed in a related way.

In either case, retrieval of the entire group must be carried out by the same getting application instance.

For example, assume that the group consists of four logical messages. The putting application looks like this:

```
PMO.Options = MQPMO_LOGICAL_ORDER | MQPMO_SYNCPOINT
```

```
MQPUT MD.MsgFlags = MQMF_MSG_IN_GROUP
```

```
MQPUT MD.MsgFlags = MQMF_MSG_IN_GROUP
```

MQGET retrieval sequence

```
MQPUT MD.MsgFlags = MQMF_MSG_IN_GROUP
MQPUT MD.MsgFlags = MQMF_LAST_MSG_IN_GROUP

MQCMIT
```

The getting application chooses not to start processing any group until all of the messages within it have arrived. MQGMO_ALL_MSGS_AVAILABLE is therefore specified for the first message in the group; the option is ignored for subsequent messages within the group.

Once the first logical message of the group is retrieved, MQGMO_LOGICAL_ORDER is used to ensure that the remaining logical messages of the group are retrieved in order.

So, the getting application looks like this:

```
/* Wait for the first message in a group, or a message not in a group */
GMO.Options = MQGMO_SYNCPOINT | MQGMO_WAIT
              | MQGMO_ALL_MSGS_AVAILABLE | MQGMO_LOGICAL_ORDER
do while ( GroupStatus == MQGS_MSG_IN_GROUP )
  MQGET
  /* Process each remaining message in the group */
  ...

MQCMIT
```

For further examples of grouping messages, see “Application segmentation of logical messages” on page 138 and “Putting and getting a group that spans units of work”.

Putting and getting a group that spans units of work

In the previous case, messages or segments cannot start to leave the node (if its destination is remote) or start to be retrieved until all of the group has been put and the unit of work is committed. This may not be what you want if it takes a long time to put the whole group, or if queue space is limited on the node. To overcome this, the group can be put in several units of work.

If the group is put within multiple units of work, it is possible for some of the group to commit even when a failure of the putting application occurs. The application must therefore save status information, committed with each unit of work, which it can use after a restart to resume an incomplete group. The simplest place to record this information is in a STATUS queue. If a complete group has been successfully put, the STATUS queue is empty.

If segmentation is involved, the logic is similar. In this case, the StatusInfo must include the *Offset*.

Here is an example of putting the group in several units of work:

```
PMO.Options = MQPMO_LOGICAL_ORDER | MQPMO_SYNCPOINT

/* First UOW */

MQPUT MD.MsgFlags = MQMF_MSG_IN_GROUP
MQPUT MD.MsgFlags = MQMF_MSG_IN_GROUP
MQPUT MD.MsgFlags = MQMF_MSG_IN_GROUP
StatusInfo = GroupId,MsgSeqNumber from MQMD
MQPUT (StatusInfo to STATUS queue) PMO.Options = MQPMO_SYNCPOINT
MQCMIT

/* Next and subsequent UOWs */
MQPUT MD.MsgFlags = MQMF_MSG_IN_GROUP
```

MQGET retrieval sequence

```
MQPUT MD.MsgFlags = MQMF_MSG_IN_GROUP
MQPUT MD.MsgFlags = MQMF_MSG_IN_GROUP
MQGET (from STATUS queue) GMO.Options = MQGMO_SYNCPOINT
StatusInfo = GroupId,MsgSeqNumber from MQMD
MQPUT (StatusInfo to STATUS queue) PMO.Options = MQPMO_SYNCPOINT
MQCMIT

/* Last UOW */
MQPUT MD.MsgFlags = MQMF_MSG_IN_GROUP
MQPUT MD.MsgFlags = MQMF_MSG_IN_GROUP
MQPUT MD.MsgFlags = MQMF_LAST_MSG_IN_GROUP
MQGET (from STATUS queue) GMO.Options = MQGMO_SYNCPOINT
MQCMIT
```

If all the units of work have been committed, the entire group has been put successfully, and the STATUS queue is empty. If not, the group must be resumed at the point indicated by the status information. MQPMO_LOGICAL_ORDER cannot be used for the first put, but can thereafter.

Restart processing looks like this:

```
MQGET (StatusInfo from STATUS queue) GMO.Options = MQGMO_SYNCPOINT
if (Reason == MQRC_NO_MSG_AVAILABLE)
    /* Proceed to normal processing */
    ...

else
    /* Group was terminated prematurely */
    Set GroupId, MsgSeqNumber in MQMD to values from Status message
    PMO.Options = MQPMO_SYNCPOINT
    MQPUT MD.MsgFlags = MQMF_MSG_IN_GROUP

    /* Now normal processing is resumed.
       Assume this is not the last message */
    PMO.Options = MQPMO_LOGICAL_ORDER | MQPMO_SYNCPOINT
    MQPUT MD.MsgFlags = MQMF_MSG_IN_GROUP
    MQPUT MD.MsgFlags = MQMF_MSG_IN_GROUP
    StatusInfo = GroupId,MsgSeqNumber from MQMD
    MQPUT (StatusInfo to STATUS queue) PMO.Options = MQPMO_SYNCPOINT
    MQCMIT
```

From the getting application, you may want to start processing the messages in a group before the whole group has arrived. This improves response times on the messages within the group, and also means that storage is not required for the entire group.

For recovery reasons, each message must be retrieved within a unit of work. However, in order to realize the above benefits, several units of work must be used for each group of messages.

As with the corresponding putting application, this requires status information to be recorded somewhere automatically as each unit of work is committed. Again, the simplest place to record this information is on a STATUS queue. If a complete group has been successfully processed, the STATUS queue is empty.

Note: For intermediate units of work, you can avoid the MQGET calls from the STATUS queue by specifying that each MQPUT to the status queue is a segment of a message (that is, by setting the MQMF_SEGMENT flag), instead of putting a complete new message for each unit of work. In the last unit of work, a final segment is put to the status queue specifying MQMF_LAST_SEGMENT, and then the status information is cleared with an MQGET specifying MQGMO_COMPLETE_MSG.

MQGET retrieval sequence

During restart processing, instead of using a single MQGET to get a possible status message, browse the status queue with MQGMO_LOGICAL_ORDER until you reach the last segment (that is, until no further segments are returned). In the first unit of work after restart, also specify the offset explicitly when putting the status segment.

In the following example, we consider only messages within a group. It is assumed that the application's buffer is always large enough to hold the entire message, whether or not the message has been segmented. MQGMO_COMPLETE_MSG is therefore specified on each MQGET. The same principles apply if segmentation is involved (in this case, the StatusInfo must include the *Offset*).

For simplicity, we assume that a maximum of 4 messages should be retrieved within a single UOW:

```
msgs = 0      /* Counts messages retrieved within UOW */
              /* Should be no status message at this point */

/* Retrieve remaining messages in the group */
do while ( GroupStatus == MQGS_MSG_IN_GROUP )

    /* Process up to 4 messages in the group */
    GMO.Options = MQGMO_SYNCPOINT | MQGMO_WAIT
                 | MQGMO_LOGICAL_ORDER
    do while ( (GroupStatus == MQGS_MSG_IN_GROUP) && (msgs < 4) )
        MQGET
        msgs = msgs + 1
        /* Process this message */
        ...
    /* end while

    /* Have retrieved last message or 4 messages */
    /* Update status message if not last in group */
    MQGET (from STATUS queue) GMO.Options = MQGMO_SYNCPOINT
    if ( GroupStatus == MQGS_MSG_IN_GROUP )
        StatusInfo = GroupId,MsgSeqNumber from MQMD
        MQPUT (StatusInfo to STATUS queue) PMO.Options = MQPMO_SYNCPOINT
    MQCMIT
    msgs = 0
/* end while

if ( msgs > 0 )
    /* Come here if there was only 1 message in the group */
    MQCMIT
```

If all of the units of work have been committed, then the entire group has been retrieved successfully, and the STATUS queue is empty. If not, then the group must be resumed at the point indicated by the status information. MQGMO_LOGICAL_ORDER cannot be used for the first retrieve, but can thereafter.

Restart processing looks like this:

```
MQGET (from STATUS queue) GMO.Options = MQGMO_SYNCPOINT
if (Reason == MQRC_NO_MSG_AVAILABLE)
    /* Proceed to normal processing */
    ...

else
    /* Group was terminated prematurely */
    /* The next message on the group must be retrieved by matching
       the sequence number and group id with those retrieved from the
       status information. */
    GMO.Options = MQGMO_COMPLETE_MSG | MQGMO_SYNCPOINT | MQGMO_WAIT
```


MQGET retrieval sequence

```
MQGET GMO.MatchOptions = MQMO_MATCH_GROUP_ID | MQMO_MATCH_MSG_SEQ_NUMBER,
      MQMD.GroupId      = value from Status message,
      MQMD.MsgSeqNumber = value from Status message plus 1
msgs = 1
/* Process this message */
...

/* Now normal processing is resumed */
/* Retrieve remaining messages in the group */
do while ( GroupStatus == MQGS_MSG_IN_GROUP )

    /* Process up to 4 messages in the group */
    GMO.Options = MQGMO_COMPLETE_MSG | MQGMO_SYNCPOINT | MQGMO_WAIT
                | MQGMO_LOGICAL_ORDER
    do while ( (GroupStatus == MQGS_MSG_IN_GROUP) && (msgs < 4) )
        MQGET
        msgs = msgs + 1
        /* Process this message */
        ...

    /* Have retrieved last message or 4 messages */
    /* Update status message if not last in group */
    MQGET (from STATUS queue) GMO.Options = MQGMO_SYNCPOINT
    if ( GroupStatus == MQGS_MSG_IN_GROUP )
        StatusInfo = GroupId,MsgSeqNumber from MQMD
        MQPUT (StatusInfo to STATUS queue) PMO.Options = MQPMO_SYNCPOINT
    MQCMIT
    msgs = 0
```

Getting a particular message

To get a particular message from a queue, use the *MsgId* and *CorrelId* fields of the MQMD structure. Note, however, that applications can explicitly set these fields, so the values you specify may not identify a unique message. Table 5 shows which message is retrieved for the possible settings of these fields. These fields are ignored on input if you specify MQGMO_MSG_UNDER_CURSOR in the *GetMsgOpts* parameter of the MQGET call.

Table 5. Using message and correlation identifiers

To retrieve ...	<i>MsgId</i>	<i>CorrelId</i>
First message in the queue	MQMI_NONE	MQCI_NONE
First message that matches <i>MsgId</i>	Nonzero	MQCI_NONE
First message that matches <i>CorrelId</i>	MQMI_NONE	Nonzero
First message that matches both <i>MsgId</i> and <i>CorrelId</i>	Nonzero	Nonzero

In each case, *first* means the first message that satisfies the selection criteria (unless MQGMO_BROWSE_NEXT is specified, when it means the *next* message in the sequence satisfying the selection criteria).

On return, the MQGET call sets the *MsgId* and *CorrelId* fields to the message and correlation identifiers (respectively) of the message returned (if any).

If you set the *Version* field of the MQMD structure to 2 or 3, you can use the *GroupId*, *MsgSeqNumber*, and *Offset* fields. Table 6 on page 134 shows which message is retrieved for the possible settings of these fields.

Getting a specific message

Table 6. Using the group identifier

To retrieve ...	Match options
First message in the queue	MQMO_NONE
First message that matches <i>MsgId</i>	MQMO_MATCH_MSG_ID
First message that matches <i>CorrelId</i>	MQMO_MATCH_CORREL_ID
First message that matches <i>GroupId</i>	MQMO_MATCH_GROUP_ID
First message that matches <i>MsgSeqNumber</i>	MQMO_MATCH_MSG_SEQ_NUMBER
First message that matches <i>MsgToken</i>	MQMO_MATCH_MSG_TOKEN
First message that matches <i>Offset</i>	MQMO_MATCH_OFFSET

Notes:

1. MQMO_MATCH_XXX implies that the XXX field in the MQMD structure is set to the value to be matched.
2. The MQMO flags can be used in combination. For example, MQMO_MATCH_GROUP_ID, MQMO_MATCH_MSG_SEQ_NUMBER, and MQMO_MATCH_OFFSET can be used together to give the segment identified by the *GroupId*, *MsgSeqNumber*, and *Offset* fields.
3. If you specify MQGMO_LOGICAL_ORDER, the message you are trying to retrieve is affected because the option depends on state information controlled for the queue handle. For information about this, see “Logical and physical ordering” on page 126 and the *MQSeries Application Programming Reference* manual.
4. MQMO_MATCH_MSG_TOKEN is used only on queues managed by the OS/390 workload manager.
5. MQSeries for OS/390 does not support MQMO_MATCH_GROUP_ID, MQMO_MATCH_MSG_SEQ_NUMBER, or MQMO_MATCH_OFFSET.

Notes:

1. The MQGET call usually retrieves the first message from a queue. If you specify a particular message when you use the MQGET call, the queue manager has to search the queue until it finds that message. This can affect the performance of your application.
2. If you are using Version 2 or 3 of the MQMD structure, you can use the MQMO_MATCH_MSG_ID and MQMO_MATCH_CORREL_ID flags. This avoids having to reset the *MsgId* and *CorrelId* fields between MQGETs.

On MQSeries for OS/390, the queue attribute, *IndexType*, can be used to increase the speed of MQGET operations on the queue. For more information, see “Type of index”.

Type of index

This is supported on MQSeries for OS/390 only.

The queue attribute, *IndexType*, specifies the type of index that the queue manager maintains in order to increase the speed of MQGET operations on the queue.

You have four options:

Value	Description
-------	-------------

NONE	
------	--

	No index is maintained. Use this when messages are retrieved sequentially (see “Priority” on page 126).
--	---

MSGID

An index of message identifiers is maintained. Use this when messages are retrieved using the *MsgId* field as a selection criterion on the MQGET call (see “Getting a particular message” on page 133).

MSGTOKEN

An index of message tokens is maintained. Use this when messages are retrieved using the *MsgToken* field as a selection criterion on the MQGET call (see “MQSeries Workflow” on page 225).

CORRELID

An index of correlation identifiers is maintained. Use this when messages are retrieved using the *CorrelId* field as a selection criterion on the MQGET call (see “Getting a particular message” on page 133).

Notes:

1. If you are indexing using the MSGID option or CORRELID option, set the relative *MsgId* or *CorrelId* parameters in the MQMD. It is *not* beneficial to set both.
2. Indexes are ignored when browsing messages on a queue (see “Browsing messages on a queue” on page 149 for more information).
3. Avoid queues (indexed by *MsgId* or *CorrelId*) containing thousands of messages because this affects restart time. (This does not apply to nonpersistent messages as they are deleted at restart.)
4. MSGTOKEN is used to define queues managed by the OS/390 workload manager.

For a full description of the *IndexType* attribute, see the *MQSeries Application Programming Reference* manual. For conditions needed to change the *IndexType* attribute, see the *MQSeries MQSC Command Reference* manual.

Handling messages greater than 4 MB long

Messages can be too large for the application, queue, or queue manager. Depending on the environment, MQSeries provides a number of ways of dealing with messages that are longer than 4 MB.

On MQSeries for OS/390, you can increase the *MaxMsgLength* attribute, up to 100 MB. This value should be set to reflect the size of the messages using the queue. On other MQSeries Version 5 products you can:

1. Increase the queue and queue manager *MaxMsgLength* attributes.
2. Use segmented messages. (Messages can be segmented by either the application or the queue manager.)
3. Use reference messages.

Each of these approaches is described in the remainder of this section.

Increasing the maximum message length

The *MaxMsgLength* queue manager attribute defines the maximum length of a message that can be handled by a queue manager. Similarly, the *MaxMsgLength* queue attribute is the maximum length of a message that can be handled by a queue. The *default* maximum message length supported depends on the environment in which you are working.

Handling large messages

If you are handling large messages, you can alter these attributes independently. The attribute value can be set between 32768 bytes and 100 MB, or between 0 bytes and 100 MB on MQSeries for OS/390.

After changing one or both of the *MaxMsgLength* attributes, restart your applications and channels to ensure that the changes take effect.

Note that on MQSeries for OS/390 you are permitted to change the queue *MaxMsgLength* attribute, but not the queue manager *MaxMsgLength* attribute (which is fixed at 100 MB).

When these changes are made, the message length must be less than or equal to both the queue and the queue manager *MaxMsgLength* attributes. However, existing messages may be longer than either attribute.

If the message is too big for the queue, MQRC_MSG_TOO_BIG_FOR_Q is returned. Similarly, if the message is too big for the queue manager, MQRC_MSG_TOO_BIG_FOR_Q_MGR is returned.

This method of handling large messages is easy and convenient. However, consider the following factors before using it:

- Uniformity among queue managers is reduced. The maximum size of message data is determined by the *MaxMsgLength* for each queue (including transmission queues) on which the message will be put. This value is often defaulted to the queue manager's *MaxMsgLength*, especially for transmission queues. This makes it difficult to predict whether a message is too large when it is to travel to a remote queue manager.
- Usage of system resources is increased. For example, applications need larger buffers, and on some platforms, there may be increased usage of shared storage. Note that queue storage should be affected only if actually required for larger messages.
- Channel batching is affected. A large message still counts as just one message towards the batch count but needs longer to transmit, thereby increasing response times for other messages.

Message segmentation

Not available on MQSeries for OS/390.

Increasing the maximum message length as discussed on page 135 has some negative implications. Also, it could still result in the message being too large for the queue or queue manager. In these cases, a message can be segmented. For information about segments, see "Message groups" on page 32.

The next sections look at common uses for segmenting messages. For putting and destructively getting, it is assumed that the MQPUT or MQGET calls *always* operate within a unit of work. It is strongly recommended that this technique is always used, to reduce the possibility of incomplete groups being present in the network. Single-phase commit by the queue manager is assumed, but of course other coordination techniques are equally valid.

Also, in the getting applications, it is assumed that if multiple servers are processing the same queue, each server executes similar code, so that one server never fails to find a message or segment that it expects to be there (because it had specified MQGMO_ALL_MSGS_AVAILABLE or MQGMO_ALL_SEGMENTS_AVAILABLE earlier).

Segmentation and reassembly by queue manager

This is the simplest scenario, in which one application puts a message to be retrieved by another. The message may be large: not too large for either the putting or the getting application to handle in a single buffer, but possibly too large for the queue manager or a queue on which the message is to be put.

The only changes necessary for these applications are for the putting application to authorize the queue manager to perform segmentation if necessary,

```
PMO.Options = (existing options)
MQPUT MD.MsgFlags = MQMF_SEGMENTATION_ALLOWED
```

and for the getting application to ask the queue manager to reassemble the message if it has been segmented:

```
GMO.Options = MQGMO_COMPLETE_MSG | (existing options)
MQGET
```

The application buffer must be large enough to contain the reassembled message (unless the MQGMO_ACCEPT_TRUNCATED_MSG option is included).

If data conversion is necessary, it may have to be done by the getting application specifying MQGMO_CONVERT. This should be straightforward because the data conversion exit is presented with the complete message. Attempting to do data conversion in a sender channel will not be successful if the message is segmented, and the format of the data is such that the data-conversion exit cannot carry out the conversion on incomplete data.

Application segmentation

This example shows how to segment a single large message

Application segmentation is used for two main reasons:

1. Queue-manager segmentation alone is not adequate because the message is too large to be handled in a single buffer by the applications.
2. Data conversion must be performed by sender channels, and the format is such that the putting application needs to stipulate where the segment boundaries are to be in order for conversion of an individual segment to be possible.

However, if data conversion is not an issue, or if the getting application always uses MQGMO_COMPLETE_MSG, queue-manager segmentation can also be allowed by specifying MQMF_SEGMENTATION_ALLOWED. In our example, the application segments the message into four segments:

```
PMO.Options = MQPMO_LOGICAL_ORDER | MQPMO_SYNCPOINT

MQPUT MD.MsgFlags = MQMF_SEGMENT
MQPUT MD.MsgFlags = MQMF_SEGMENT
MQPUT MD.MsgFlags = MQMF_SEGMENT
MQPUT MD.MsgFlags = MQMF_LAST_SEGMENT

MQCMIT
```

If you do not use MQPMO_LOGICAL_ORDER, the application must set the *Offset* and the length of each segment. In this case, logical state is not maintained automatically.

The getting application cannot, or chooses not to, guarantee to have a buffer that will hold any reassembled message. It must therefore be prepared to process segments individually.

Handling large messages

For messages that are segmented, this application does not want to start processing one segment until all of the segments that constitute the logical message are present. MQGMO_ALL_SEGMENTS_AVAILABLE is therefore specified for the first segment. If you specify MQGMO_LOGICAL_ORDER and there is a current logical message, MQGMO_ALL_SEGMENTS_AVAILABLE is ignored.

Once the first segment of a logical message has been retrieved, MQGMO_LOGICAL_ORDER is used to ensure that the remaining segments of the logical message are retrieved in order.

No consideration is given to messages within different groups. If such messages do occur, they are processed in the order in which the first segment of each message appears on the queue.

```
GMO.Options = MQGMO_SYNCPOINT | MQGMO_LOGICAL_ORDER
              | MQGMO_ALL_SEGMENTS_AVAILABLE | MQGMO_WAIT
do while ( SegmentStatus == MQSS_SEGMENT )
  MQGET
  /* Process each remaining segment of the logical message */
  ...

MQCMIT
```

Application segmentation of logical messages

The messages must be maintained in logical order in a group, and some or all of them may be so large that they require application segmentation.

In our example, a group of four logical messages is to be put. All but the third message are large, and require segmentation which is performed by the putting application:

```
PMO.Options = MQPMO_LOGICAL_ORDER | MQPMO_SYNCPOINT

MQPUT MD.MsgFlags = MQMF_MSG_IN_GROUP      | MQMF_SEGMENT
MQPUT MD.MsgFlags = MQMF_MSG_IN_GROUP      | MQMF_SEGMENT
MQPUT MD.MsgFlags = MQMF_MSG_IN_GROUP      | MQMF_LAST_SEGMENT

MQPUT MD.MsgFlags = MQMF_MSG_IN_GROUP      | MQMF_SEGMENT
MQPUT MD.MsgFlags = MQMF_MSG_IN_GROUP      | MQMF_LAST_SEGMENT

MQPUT MD.MsgFlags = MQMF_MSG_IN_GROUP

MQPUT MD.MsgFlags = MQMF_LAST_MSG_IN_GROUP | MQMF_SEGMENT
MQPUT MD.MsgFlags = MQMF_LAST_MSG_IN_GROUP | MQMF_SEGMENT
MQPUT MD.MsgFlags = MQMF_LAST_MSG_IN_GROUP | MQMF_LAST_SEGMENT

MQCMIT
```

In the getting application, MQGMO_ALL_MESSAGES_AVAILABLE is specified on the first MQGET. This means that no messages or segments of a group are retrieved until the entire group is available. When the first physical message of a group has been retrieved, MQGMO_LOGICAL_ORDER is used to ensure that the segments and messages of the group are retrieved in order:

```
GMO.Options = MQGMO_SYNCPOINT | MQGMO_LOGICAL_ORDER
              | MQGMO_ALL_MESSAGES_AVAILABLE | MQGMO_WAIT

do while ( (GroupStatus != MQGS_LAST_MSG_IN_GROUP) ||
           (SegmentStatus != MQGS_LAST_SEGMENT) )
  MQGET
  /* Process a segment or complete logical message. Use the GroupStatus
```

and SegmentStatus information to see what has been returned */
...

MQCMIT

Note: If you specify MQGMO_LOGICAL_ORDER and there is a current group, MQGMO_ALL_MSGS_AVAILABLE is ignored.

Putting and getting a segmented message that spans units of work

You can put and get a segmented message that spans a unit of work in a similar way to “Putting and getting a group that spans units of work” on page 130.

You cannot, however, put or get segmented messages in a global unit of work.

Reference messages

This does not apply to MQSeries for OS/390.

This method allows a large object to be transferred from one node to another without the need for the object to be stored on MQSeries queues at either the source or the destination nodes. This is of particular benefit where the data already exists in another form, for example, for mail applications.

To do this, you need to specify a message exit at both ends of a channel. For information on how to do this, see the *MQSeries Intercommunication* book.

MQSeries defines the format of a reference message header (MQRMH). See the *MQSeries Application Programming Reference* manual for a description of this. This is recognized by means of a defined format name and may or may not be followed by actual data.

To initiate transfer of a large object, an application can put a message consisting of a reference message header with no data following it. As this message leaves the node, the message exit retrieves the object in an appropriate way and appends it to the reference message. It then returns the message (now larger than before) to the sending Message Channel Agent for transmission to the receiving MCA.

Another message exit is configured at the receiving MCA. When this message exit sees one of these messages, it creates the object using the object data that was appended and passes on the reference message *without* it. The reference message can now be received by an application and this application knows that the object (or at least the portion of it represented by this reference message) has been created at this node.

The maximum amount of object data that a sending message exit can append to the reference message is limited by the negotiated maximum message length for the channel. The exit can only return a single message to the MCA for each message that it is passed, so the putting application can put several messages to cause one object to be transferred. Each message must identify the *logical* length and offset of the object that is to be appended to it. However, in cases where it is not possible to know the total size of the object or the maximum size allowed by the channel, the sending message exit can be designed so that the putting application just puts a single message, and the exit itself puts the next message on the transmission queue when it has appended as much data as it can to the message it has been passed.

Handling large messages

Before using this method of dealing with large messages, consider the following:

- The MCA and the message exit run under an MQSeries user ID. The message exit (and therefore, the user ID) needs to access the object to either retrieve it at the sending end or create it at the receiving end; this may only be feasible in cases where the object is universally accessible. This raises a security issue.
- If the reference message with bulk data appended to it must travel through several queue managers before reaching its destination, the bulk data *is* present on MQSeries queues at the intervening nodes. However, no special support or exits need to be provided in these cases.
- Designing your message exit is made difficult if rerouting or dead-letter queuing is allowed. In these cases, the portions of the object may arrive out of order.
- When a reference message arrives at its destination, the receiving message exit creates the object. However, this is not synchronized with the MCA's unit of work, so if the batch is backed out, another reference message containing this same portion of the object will arrive in a later batch, and the message exit may attempt to recreate the same portion of the object. If the object is, for example, a series of database updates, this might be unacceptable. If so, the message exit must keep a log of which updates have been applied; this may require the use of an MQSeries queue.
- Depending on the characteristics of the object type, the message exits and applications may need to cooperate in maintaining use counts, so that the object can be deleted when it is no longer needed. An instance identifier may also be required; a field is provided for this in the reference message header (see the *MQSeries Application Programming Reference* manual).
- If a reference message is put as a distribution list, the object must be retrievable for each resulting distribution list or individual destination at that node. You may need to maintain use counts. Also consider the possibility that a given node may be the final node for some of the destinations in the list, but an intermediate node for others.
- Bulk data is not normally converted. This is because conversion takes place *before* the message exit is invoked. For this reason, conversion should not be requested on the originating sender channel. If the reference message passes through an intermediate node, the bulk data is converted when sent from the intermediate node, if requested.
- Reference messages cannot be segmented.

Using the MQRMH and MQMD structures

See the *MQSeries Application Programming Reference* manual for a description of the fields in the reference message header and the message descriptor.

In the MQMD structure, the *Format* field must be set to MQFMT_REF_MSG_HEADER. The MQHREF format, when requested on MQGET, is converted automatically by MQSeries along with any bulk data that follows.

Here is an example of the use of the *DataLogicalOffset* and *DataLogicalLength* fields of the MQRMH:

A putting application might put a reference message with:

- No physical data
- *DataLogicalLength* = 0 (this message represents the entire object)
- *DataLogicalOffset* = 0.

Assuming that the object is 70,000 bytes long, the sending message exit sends the first 40,000 bytes along the channel in a reference message containing:

- 40,000 bytes of physical data following the MQRMH
- *DataLogicalLength* = 40,000
- *DataLogicalOffset* = 0 (from the start of the object).

It then places another message on the transmission queue containing:

- No physical data
- *DataLogicalLength* = 0 (to the end of the object). You could specify a value of 30,000 here.
- *DataLogicalOffset* = 40,000 (starting from this point).

When this message exit is seen by the sending message exit, the remaining 30,000 bytes of data is appended, and the fields are set to:

- 30,000 bytes of physical data following the MQRMH
- *DataLogicalLength* = 30,000
- *DataLogicalOffset* = 40,000 (starting from this point).

The MQRMHF_LAST flag is also set.

For a description of the sample programs provided for the use of reference messages, see “Chapter 34. Sample programs (all platforms except OS/390)” on page 359.

Waiting for messages

If you want a program to wait until a message arrives on a queue, specify the MQGMO_WAIT option in the *Options* field of the MQGMO structure. Use the *WaitInterval* field of the MQGMO structure to specify the maximum time (in milliseconds) that you want an MQGET call to wait for a message to arrive on a queue.

If the message does not arrive within this time, the MQGET call completes with the MQRC_NO_MSG_AVAILABLE reason code.

You can specify an unlimited wait interval using the constant MQWI_UNLIMITED in the *WaitInterval* field. However, events outside your control could cause your program to wait for a long time, so use this constant with caution. IMS applications should not specify an unlimited wait interval because this would prevent the IMS system terminating. (When IMS terminates, it requires all dependent regions to end.) Instead, IMS applications should specify a finite wait interval; then, if the call completes without retrieving a message after that interval, issue another MQGET call with the wait option.

In the Windows 3.1 environment, while your application is waiting for an MQGET to return, MQSeries will still recover Windows messages to allow the application and the rest of Windows to function normally. You must ensure that your code that processes Windows program messages does not assume that the MQGET call returns data to the application immediately. If it attempts to access data that is not yet available, errors can easily occur. Also, if you attempt to make other MQI calls while the MQGET call is waiting, MQRC_CALL_IN_PROGRESS is returned to show that another call is busy.

Note: If more than one program is waiting on the same shared queue to *remove* a message, only one program is activated by a message arriving. However, if

Waiting for messages

more than one program is waiting to browse a message, all the programs can be activated. For more information, see the description of the *Options* field of the MQGMO structure in the *MQSeries Application Programming Reference* manual.

If the state of the queue or the queue manager changes before the wait interval expires, the following actions occur:

- If the queue manager enters the quiescing state, and you used the MQGMO_FAIL_IF_QUIESCING option, the wait is canceled and the MQGET call completes with the MQRC_Q_MGR_QUIESCING reason code. Without this option, the call remains waiting.
- On OS/390, if the connection (for a CICS or IMS application) enters the quiescing state, and you used the MQGMO_FAIL_IF_QUIESCING option, the wait is canceled and the MQGET call completes with the MQRC_CONN_QUIESCING reason code. Without this option, the call remains waiting.
- If the queue manager is forced to stop, or is canceled, the MQGET call completes with either the MQRC_Q_MGR_STOPPING or the MQRC_CONNECTION_BROKEN reason code.
- If the attributes of the queue (or a queue to which the queue name resolves) are changed so that get requests are now inhibited, the wait is canceled and the MQGET call completes with the MQRC_GET_INHIBITED reason code.
- If the attributes of the queue (or a queue to which the queue name resolves) are changed in such a way that the FORCE option is required, the wait is canceled and the MQGET call completes with the MQRC_OBJECT_CHANGED reason code.

If you want your application to wait on more than one queue, use the signal facility of MQSeries for OS/390 (see “Signaling”). For more information about the circumstances in which these actions occur, see the *MQSeries Application Programming Reference* manual.

Signaling

Signaling is supported only on MQSeries for OS/390, MQSeries for Tandem NonStop Kernel, and MQSeries for Windows Version 2.1.

Signaling is an option on the MQGET call to allow the operating system to notify (or *signal*) a program when an expected message arrives on a queue. This is similar to the “get with wait” function described on page 141 because it allows your program to continue with other work while waiting for the signal. However, if you use signaling, you can free the application thread and rely on the operating system to notify the program when a message arrives.

To set a signal

To set a signal, do the following in the MQGMO structure that you use on your MQGET call:

1. Set the MQGMO_SET_SIGNAL option in the *Options* field.
2. Set the maximum life of the signal in the *WaitInterval* field. This sets the length of time (in milliseconds) for which you want MQSeries to monitor the queue. Use the MQWI_UNLIMITED value to specify an unlimited life.

Note: IMS applications should not specify an unlimited wait interval because this would prevent the IMS system from terminating. (When IMS

terminates, it requires all dependent regions to end.) Instead, IMS applications should examine the state of the ECB at regular intervals (see step 3). A program can have signals set on several queue handles at the same time:

3. On MQSeries for Tandem NonStop Kernel, specify an application tag in the *Signal1* field. This can be used by an application to associate the IPC notification message with a particular MQGET call (see “When the message arrives”). On MQSeries for Windows Version 2.1, specify the handle of the window to which you want the signal sent in the *Signal1* field. On MQSeries for OS/390, specify the address of the *Event Control Block* (ECB) in the *Signal1* field. This notifies you of the result of your signal. The ECB storage must remain available until the queue is closed.
4. On MQSeries for Windows Version 2.1, specify an identifier for the signal message in the *Signal2* field. This specifies the Windows message that you receive when a suitable message arrives. Use a RegisterWindow message to find a suitable identifier.

Note: You cannot use the MQGMO_SET_SIGNAL option in conjunction with the MQGMO_WAIT option.

When the message arrives

When a suitable message arrives, the following occurs:

- On MQSeries for Tandem NonStop Kernel An Inter-Process Communication (IPC) message is sent to the \$RECEIVE queue of the process that made the MQGET call.
- On MQSeries for Windows Version 2.1, MQSeries sends a Windows message (identified in step 4) to the window you specified in your *Signal1* field. It also puts a completion code in the WPARAM field of the Windows message.
- On MQSeries for OS/390, a completion code is returned to the ECB.

The completion code describes one of the following:

- The message you set the signal for has arrived on the queue. The message is not reserved for the program that requested a signal, so the program must issue an MQGET call again to get the message.

Note: Another application could get the message in the time between you receiving the signal and you issuing another MQGET call.

- The wait interval you set has expired and the message you set the signal for did not arrive on the queue. MQSeries has canceled the signal.
- The signal has been canceled. This happens, for example, if the queue manager stops or the attribute of the queue is changed so that MQGET calls are no longer allowed.

When a suitable message is already on the queue, the MQGET call completes in the same way as an MQGET call without signaling. Also, if an error is detected immediately, the call completes and the return codes are set.

When the call is accepted and no message is immediately available, control is returned to the program so that it can continue with other work. None of the output fields in the message descriptor are set, but the *CompCode* and *Reason* parameters are set to MQCC_WARNING and MQRC_SIGNAL_REQUEST_ACCEPTED, respectively.

Signaling

For information on what MQSeries can return to your application when it makes an MQGET call using signaling, see the *MQSeries Application Programming Reference manual*.

On MQSeries for OS/390, if the program has no other work to do while it is waiting for the ECB to be posted, it can wait for the ECB using:

- For a CICS Transaction Server for OS/390 program, the EXEC CICS WAIT EXTERNAL command
- For batch and IMS programs, the OS/390 WAIT macro

If the state of the queue or the queue manager changes while the signal is set (that is, the ECB has not yet been posted), the following actions occur:

- If the queue manager enters the quiescing state, and you used the MQGMO_FAIL_IF QUIESCING option, the signal is canceled. The ECB is posted with the MQEC_Q_MGR QUIESCING completion code. Without this option, the signal remains set.
- If the queue manager is forced to stop, or is canceled, the signal is canceled. The signal is delivered with the MQEC_WAIT_CANCELED completion code.
- If the attributes of the queue (or a queue to which the queue name resolves) are changed so that get requests are now inhibited, the signal is canceled. The signal is delivered with the MQEC_WAIT_CANCELED completion code.

Notes:

1. If more than one program has set a signal on the same shared queue to remove a message, only one program is activated by a message arriving. However, if more than one program is waiting to browse a message, all the programs can be activated. The rules that the queue manager follows when deciding which applications to activate are the same as those for waiting applications: for more information, see the description of the *Options* field of the MQGMO structure in the *MQSeries Application Programming Reference manual*.
2. If there is more than one MQGET call waiting for the same message, with a mixture of wait and signal options, each waiting call is considered equally. For more information, see the description of the *Options* field of the MQGMO structure in the *MQSeries Application Programming Reference manual*.
3. Under some conditions, it is possible both for an MQGET call to retrieve a message and for a signal (resulting from the arrival of the same message) to be delivered. This means that when your program issues another MQGET call (because the signal was delivered), there could be no message available. You should design your program to test for this situation.

For information about how to set a signal, see the description of the MQGMO_SET_SIGNAL option and the *Signal1* field in the *MQSeries Application Programming Reference manual*.

Skipping backout

Skipping backout is supported only on MQSeries for OS/390.

As part of a unit of work, an application program can issue one or more MQGET calls to get messages from a queue. If the application program detects an error, it can back out the unit of work. This restores all the resources updated during that unit of work to the state they were in before the unit of work started, and reinstates the messages retrieved by the MQGET calls.

Skipping backout

Once reinstated, these messages are available to subsequent MQGET calls issued by the application program. In many cases, this does not cause a problem for the application program. However, in cases where the error leading to the backout cannot be circumvented, having the message reinstated on the queue can cause the application program to enter an 'MQGET-error-backout' loop.

To avoid this problem, specify the MQGMO_MARK_SKIP_BACKOUT option on the MQGET call. This marks the MQGET request as not being involved in application-initiated backout; that is, it should not be backed out. Use of this option means that when a backout occurs, updates to other resources are backed out as required, but the marked message is treated as if it had been retrieved under a new unit of work. The application program can then perform exception handling, such as informing the originator that the message has been discarded, and then commit the new unit of work, causing the message to be removed from the queue. If the new unit of work is backed out (for any reason) the message is reinstated on the queue.

Within a unit of work, there can be only one MQGET request marked as skipping backout; however, there can be several other messages that are not marked as skipping backout. Once a message has been marked as skipping backout, any further MQGET calls within the unit of work that specify MQGMO_MARK_SKIP_BACKOUT will fail with reason code MQRC_SECOND_MARK_NOT_ALLOWED.

Notes:

1. The marked message only skips backout if the unit of work containing it is terminated by an application request to back it out. If the unit of work is backed out for any other reason, the message is backed out on to the queue in the same way that it would be if it was not marked to skip backout.
2. Skip backout is not supported within DB2 stored procedures participating in units of work controlled by RRS. For example, an MQGET call with the MQGMO_MARK_SKIP_BACKOUT option will fail with the reason code MQRC_OPTION_ENVIRONMENT_ERROR.

Skipping backout

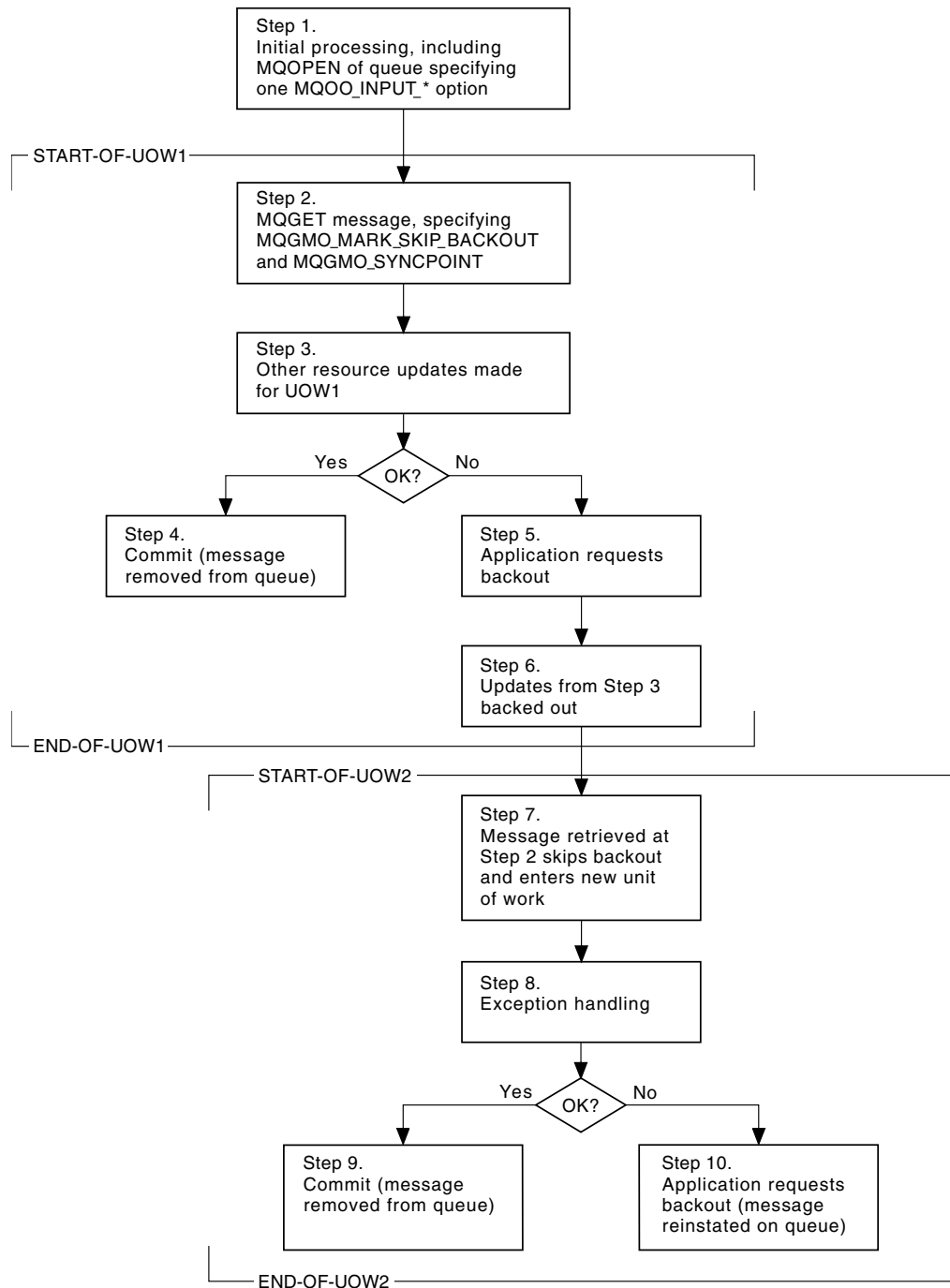


Figure 12. Skipping backout using MQGMO_MARK_SKIP_BACKOUT

Figure 12 illustrates a typical sequence of steps that an application program might contain when an MQGET request is required to skip backout:

Step 1 Initial processing occurs within the transaction, including an MQOPEN call to open the queue (specifying one of the MQOO_INPUT_* options in order to get messages from the queue in Step 2).

Step 2 MQGET is called, with MQGMO_SYNCPOINT and MQGMO_MARK_SKIP_BACKOUT. MQGMO_SYNCPOINT is required

because MQGET must be within a unit of work for MQGMO_MARK_SKIP_BACKOUT to be effective. In Figure 12 on page 146 this unit of work is referred to as UOW1.

- Step 3** Other resource updates are made as part of UOW1. These may include further MQGET calls (issued without MQGMO_MARK_SKIP_BACKOUT).
- Step 4** All updates from Steps 2 and 3 complete as required. The application program commits the updates and UOW1 ends. The message retrieved in Step 2 is removed from the queue.
- Step 5** Some of the updates from Steps 2 and 3 do not complete as required. The application program requests that the updates made during these steps are backed out.
- Step 6** The updates made in Step 3 are backed out.
- Step 7** The MQGET request made in Step 2 skips backout and becomes part of a new unit of work, UOW2.
- Step 8** UOW2 performs exception handling in response to UOW1 being backed out. (For example, an MQPUT call to another queue, indicating that a problem occurred that caused UOW1 to be backed out.)
- Step 9** Step 8 completes as required, the application program commits the activity, and UOW2 ends. As the MQGET request is part of UOW2 (see Step 7), this commit causes the message to be removed from the queue.
- Step 10**
Step 8 does not complete as required and the application program backs out UOW2. Because the get message request is part of UOW2 (see Step 7), it too is backed out and reinstated on the queue. It is now available to further MQGET calls issued by this or another application program (in the same way as any other message on the queue).

Application data conversion

When necessary, MCAs convert the message descriptor data into the required character set and encoding. Either end of the link (that is, the local MCA or the remote MCA) may do the conversion.

When an application puts messages on a queue, the local queue manager adds control information to the message descriptors to facilitate the control of the messages when they are processed by queue managers and MCAs. Depending on the environment, the message header data fields will be created in the character set and encoding of the local system.

When you move messages between systems, it is necessary, on some occasions, to convert the application data into the character set and encoding required by the receiving system. This can be done either from within application programs on the receiving system or by the MCAs on the sending system. If data conversion is supported on the receiving system, it is recommended to use application programs to convert the application data, rather than depending on the conversion having already occurred at the sending system.

MQGET data conversion

Application data is converted within an application program when the MQGMO_CONVERT option is specified in the *Options* field of the MQGMO structure passed to an MQGET call, and *all* of the following are true:

- The *CodedCharSetId* or *Encoding* fields set in the MQMD structure associated with the message on the queue differ from the *CodedCharSetId* or *Encoding* fields set in the MQMD structure specified on the MQGET call.
- The *Format* field in the MQMD structure associated with the message is not MQFMT_NONE.
- The *BufferLength* specified on the MQGET call is not zero.
- The message data length is not zero.
- The queue manager supports conversion between the *CodedCharSetId* and *Encoding* fields specified in the MQMD structures associated with the message and the MQGET call. See the *MQSeries Application Programming Reference* manual for details of the coded character set identifiers and machine encodings supported.
- The queue manager supports conversion of the message format. If the *Format* field of the MQMD structure associated with the message is one of the built-in formats, the queue manager is able to convert the message. If the *Format* is not one of the built-in formats, you need to write a data-conversion exit to convert the message.

If the sending MCA is to convert the data, the CONVERT(YES) keyword must be specified on the definition of each sender or server channel for which conversion is required. If the data conversion fails, the message is sent to the DLQ at the sending queue manager and the *Feedback* field of the MQDLH structure indicates the reason. If the message cannot be put on the DLQ, the channel will close and the unconverted message will remain on the transmission queue. Data conversion within applications rather than at sending MCAs avoids this situation.

As a general rule, data in the message that is described as “character” data by the built-in format or data-conversion exit is converted from the coded character set used by the message to that requested, and “numeric” fields are converted to the encoding requested.

For further details of the conversion processing conventions used when converting the built-in formats, and for information about writing your own data-conversion exits, see “Chapter 11. Writing data-conversion exits” on page 155. See also the *MQSeries Application Programming Reference* manual for information about the language support tables and about the supported machine encodings.

Conversion of EBCDIC newline characters

If you need to ensure that the data you send from an EBCDIC platform to an ASCII one is identical to the data you receive back again, you must control the conversion of EBCDIC newline characters. This can be done using a platform-dependent switch that forces MQSeries to use the unmodified conversion tables but you must be aware of the inconsistent behavior that may result.

The problem arises because the EBCDIC newline character is not converted consistently across platforms or conversion tables. As a result, if the data is displayed on an ASCII platform, the formatting may be incorrect. This would make it difficult, for example, to administer an AS/400 remotely from an ASCII platform using RUNMQSC.

See the *MQSeries System Administration* book for further information about converting EBCDIC-format data to ASCII format.

Browsing messages on a queue

To use the MQGET call to browse the messages on a queue:

1. Call MQOPEN to open the queue for browsing, specifying the MQOO_BROWSE option.
2. To browse the first message on the queue, call MQGET with the MQGMO_BROWSE_FIRST option. To find the message you want, you can call MQGET repeatedly with the MQGMO_BROWSE_NEXT option to step through many messages.
You *must* set the *MsgId* and *CorrelId* fields of the MQMD structure to null after each MQGET call in order to see all messages.
3. Call MQCLOSE to close the queue.

The browse cursor

When you open (MQOPEN) a queue for browsing, the call establishes a browse cursor for use with MQGET calls that use one of the browse options. You can think of the browse cursor as a logical pointer that is positioned before the first message on the queue.

You can have more than one browse cursor active (from a single program) by issuing several MQOPEN requests for the same queue.

When you call MQGET for browsing, use one of the following options in your MQGMO structure:

MQGMO_BROWSE_FIRST

Gets a copy of the first message that satisfies the conditions specified in your MQMD structure.

MQGMO_BROWSE_NEXT

Gets a copy of the next message that satisfies the conditions specified in your MQMD structure.

In both cases, the message remains on the queue.

When you open a queue, the browse cursor is positioned logically just before the first message on the queue. This means that if you make your MQGET call immediately after your MQOPEN call, you can use the MQGMO_BROWSE_NEXT option to browse the first message; you do not have to use the MQGMO_BROWSE_FIRST option.

The order in which messages are copied from the queue is determined by the *MsgDeliverySequence* attribute of the queue. (For more information, see "The order in which messages are retrieved from a queue" on page 126.)

Queues in FIFO (first in, first out) sequence

The first message in a queue in this sequence is the message that has been on the queue the longest.

Use MQGMO_BROWSE_NEXT to read the messages sequentially in the queue. You will see any messages put to the queue while you are browsing, as a queue in this sequence will have messages placed at the end. When the cursor has recognized that it has reached the end of the queue, the browse cursor will stay

Browsing messages

where it is and return with MQRC_NO_MSG_AVAILABLE. You may then either leave it there waiting for further messages or reset it to the beginning of the queue with a MQGMO_BROWSE_FIRST call.

Queues in priority sequence

The first message in a queue in this sequence is the message that has been on the queue the longest and has the highest priority at the time the MQOPEN call is issued.

Use MQGMO_BROWSE_NEXT to read the messages in the queue.

The browse cursor will point to the next message, working from the priority of the first message to finish with the message at the lowest priority. It will browse any messages put to the queue during this time as long as they are of equal or lower priority to the message identified by the current browse cursor.

Any messages put to the queue of higher priority can only be browsed by:

- Opening the queue for browse again, at which point a new browse cursor is established
- Using the MQGMO_BROWSE_FIRST option

Uncommitted messages

An uncommitted message is never visible to a browse; the browse cursor skips past it. Messages within a unit-of-work cannot be browsed until the unit-of-work is committed. Messages do not change their position on the queue when committed, so skipped uncommitted messages will not be seen, even when they *are* committed, unless you use the MQGMO_BROWSE_FIRST option and work through the queue again.

Change to queue sequence

If the message delivery sequence is changed from priority to FIFO while there are messages on the queue, the order of the messages that are already queued is not changed. Messages added to the queue subsequently take the default priority of the queue.

Browsing messages when message length unknown

To browse a message when you do not know the size of the message, and you do not wish to use the *MsgId*, *CorrelId*, or *GroupId* fields to locate the message, you can use the MQGMO_BROWSE_MSG_UNDER_CURSOR option (not supported on OS/390):

1. Issue an MQGET with:
 - Either the MQGMO_BROWSE_FIRST or MQGMO_BROWSE_NEXT option
 - The MQGMO_ACCEPT_TRUNCATED_MSG option
 - Buffer length zero

Note: If another program is likely to get the same message, consider using the MQGMO_LOCK option as well. MQRC_TRUNCATED_MSG_ACCEPTED should be returned.

2. Use the returned *DataLength* to allocate the storage needed.
3. Issue an MQGET with the MQGMO_BROWSE_MSG_UNDER_CURSOR.

The message pointed to is the last one that was retrieved; the browse cursor will not have moved. You can choose either to lock the message using the MQGMO_LOCK option, or to unlock a locked message using MQGMO_UNLOCK option.

The call fails if no MQGET with either the MQGMO_BROWSE_FIRST or MQGMO_BROWSE_NEXT options has been issued successfully since the queue was opened.

Restriction

MQGMO_LOCK and MQGMO_UNLOCK are not available on MQSeries for Tandem NonStop Kernel and MQSeries for OS/390.

Removing a message you have browsed

You can remove from the queue a message you have already browsed provided you have opened the queue for removing messages as well as for browsing. (You must specify one of the MQOO_INPUT_* options, as well as the MQOO_BROWSE option, on your MQOPEN call.)

To remove the message, call MQGET again, but in the *Options* field of the MQGMO structure, specify MQGMO_MSG_UNDER_CURSOR. In this case, the MQGET call ignores the *MsgId*, *CorrelId*, and *GroupId* fields of the MQMD structure.

In the time between your browsing and removal steps, another program may have removed messages from the queue, including the message under your browse cursor. In this case, your MQGET call returns a reason code to say that the message is not available.

Browsing messages in logical order

Browsing messages in logical order is supported on MQSeries Version 5 products only, except MQSeries for OS/390.

“Logical and physical ordering” on page 126 discusses the difference between the logical and physical order of messages on a queue. This distinction is particularly important when browsing a queue, because, in general, messages are not being deleted and browse operations do not necessarily start at the beginning of the queue. If an application browses through the various messages of one group (using logical order), it is important that logical order should be followed to reach the start of the next group, since the last message of one group may occur physically *after* the first message of the next group. The MQGMO_LOGICAL_ORDER option ensures that logical order is followed when scanning a queue.

MQGMO_ALL_MSGS_AVAILABLE (or MQGMO_ALL_SEGMENTS_AVAILABLE) needs to be used with care for browse operations. Consider the case of logical messages with MQGMO_ALL_MSGS_AVAILABLE. The effect of this is that a logical message is available only if all of the remaining messages in the group are also present. If they are not, the message is passed over. This can mean that when the missing messages arrive subsequently, they will not be noticed by a browse-next operation.

For example, if the following logical messages are present,

```
Logical message 1 (not last) of group 123
Logical message 1 (not last) of group 456
Logical message 2 (last)      of group 456
```

Browsing messages in logical order

and a browse function is issued with MQGMO_ALL_MSGS_AVAILABLE, the first logical message of group 456 is returned, leaving the browse cursor on this logical message. If the second (last) message of group 123 now arrives,

```
Logical message 1 (not last) of group 123
Logical message 2 (last)      of group 123
Logical message 1 (not last) of group 456 <=== browse cursor
Logical message 2 (last)      of group 456
```

and the same browse-next function is issued, it will not be noticed that group 123 is now complete, because the first message of this group is *before* the browse cursor.

In some cases (for example, if messages are retrieved destructively when the group is present in its entirety), it may be acceptable to use MQGMO_ALL_MSGS_AVAILABLE together with MQGMO_BROWSE_FIRST. Otherwise, the browse scan must be repeated in order to take note of newly arrived messages that have been missed; just issuing MQGMO_WAIT together with MQGMO_BROWSE_NEXT and MQGMO_ALL_MSGS_AVAILABLE does not take account of them. (This also happens to higher-priority messages that might arrive after scanning the messages is complete.)

The next sections look at browsing examples that deal with unsegmented messages; segmented messages follow similar principles.

Browsing messages in groups

In this example, the application browses through each message on the queue, in logical order.

Messages on the queue may either be grouped or not. For grouped messages, the application does not want to start processing any group until all of the messages within it have arrived. MQGMO_ALL_MSGS_AVAILABLE is therefore specified for the first message in the group; for subsequent messages in the group, this option is unnecessary.

MQGMO_WAIT is used in this example. However, although the wait can be satisfied if a new group arrives, for the reasons in “Browsing messages in logical order” on page 151, it will not be satisfied if the browse cursor has already passed the first logical message in a group, and the remaining messages now arrive. Nevertheless, waiting for a suitable interval ensures that the application does not constantly loop while waiting for new messages or segments.

MQGMO_LOGICAL_ORDER is used throughout, to ensure that the scan is in logical order. This contrasts with the destructive MQGET example, where because each group is being removed, MQGMO_LOGICAL_ORDER is not used when looking for the first (or only) message in a group.

It is assumed that the application’s buffer is always large enough to hold the entire message, whether or not the message has been segmented.

MQGMO_COMPLETE_MSG is therefore specified on each MQGET.

The following gives an example of browsing logical messages in a group:

```
/* Browse the first message in a group, or a message not in a group */
GMO.Options = MQGMO_BROWSE_NEXT | MQGMO_COMPLETE_MSG | MQGMO_LOGICAL_ORDER
              | MQGMO_ALL_MSGS_AVAILABLE | MQGMO_WAIT
MQGET GMO.MatchOptions = MQMO_MATCH_MSG_SEQ_NUMBER, MD.MsgSeqNumber = 1
/* Examine first or only message */
```

Browsing messages in logical order

```
...
GMO.Options = MQGMO_BROWSE_NEXT | MQGMO_COMPLETE_MSG | MQGMO_LOGICAL_ORDER
do while ( GroupStatus == MQGS_MSG_IN_GROUP )
  MQGET
  /* Examine each remaining message in the group */
  ...
```

The above group is repeated until MQRC_NO_MSG_AVAILABLE is returned.

Note that the Application Messaging Interface (AMI) supports simulated message grouping on MQSeries for OS/390. Refer to the *MQSeries Application Messaging Interface* book for further details.

Browsing and retrieving destructively

In this example, the application browses each of the logical messages within a group, before deciding whether to retrieve that group destructively.

The first part of this example is similar to the previous one. However in this case, having browsed an entire group, we may decide to go back and retrieve it destructively.

As each group is removed in this example, MQGMO_LOGICAL_ORDER is not used when looking for the first or only message in a group.

The following gives an example of browsing and then retrieving destructively:

```
GMO.Options = MQGMO_BROWSE_NEXT | MQGMO_COMPLETE_MSG | MQGMO_LOGICAL_ORDER
              | MQGMO_ALL_MESSAGES_AVAILABLE | MQGMO_WAIT
do while ( GroupStatus == MQGS_MSG_IN_GROUP )
  MQGET
  /* Examine each remaining message in the group (or as many as
     necessary to decide whether or not to get it destructively) */
  ...

if ( we want to retrieve the group destructively )

  if ( GroupStatus == ' ' )
    /* We retrieved an ungrouped message */
    GMO.Options = MQGMO_MSG_UNDER_CURSOR | MQGMO_SYNCPOINT
    MQGET GMO.MatchOptions = 0
    /* Process the message */
    ...

  else
    /* We retrieved one or more messages in a group. The browse cursor */
    /* will not normally be still on the first in the group, so we have */
    /* to match on the GroupId and MsgSeqNumber = 1. */
    /* Another way, which works for both grouped and ungrouped messages, */
    /* would be to remember the MsgId of the first message when it was */
    /* browsed, and match on that. */
    GMO.Options = MQGMO_COMPLETE_MSG | MQGMO_SYNCPOINT
    MQGET GMO.MatchOptions = MQMO_MATCH_GROUP_ID
                      | MQMO_MATCH_MSG_SEQ_NUMBER,
          (MQMD.GroupId = value already in the MD)
          MQMD.MsgSeqNumber = 1
    /* Process first or only message */
    ...

  GMO.Options = MQGMO_COMPLETE_MSG | MQGMO_SYNCPOINT
                | MQGMO_LOGICAL_ORDER
```

Browsing messages in logical order

```
do while ( GroupStatus == MQGS_MSG_IN_GROUP )
  MQGET
  /* Process each remaining message in the group */
  ...
```

Some cases where the MQGET call fails

If certain attributes of a queue are changed using the FORCE option on a command between issuing an MQOPEN and an MQGET call, the MQGET call fails and returns the MQRC_OBJECT_CHANGED reason code. The queue manager marks the object handle as being no longer valid. This also happens if the changes apply to any queue to which the queue name resolves. The attributes that affect the handle in this way are listed in the description of the MQOPEN call in the *MQSeries Application Programming Reference* manual. If your call returns the MQRC_OBJECT_CHANGED reason code, close the queue, reopen it, then try to get a message again.

If get operations are inhibited for a queue from which you are attempting to get messages (or any queue to which the queue name resolves), the MQGET call fails and returns the MQRC_GET_INHIBITED reason code. This happens even if you are using the MQGET call for browsing. You may be able to get a message successfully if you attempt the MQGET call at a later time, if the design of the application is such that other programs change the attributes of queues regularly.

If a dynamic queue (either temporary or permanent) has been deleted, MQGET calls using a previously acquired object handle fail and return the MQRC_Q_DELETED reason code.

Chapter 11. Writing data-conversion exits

Data-conversion exits are not supported on MQSeries for Windows or VSE/ESA.

The Message Descriptor of a message is created by your application when you do an MQPUT. Because MQSeries needs to be able to understand the contents of the MQMD regardless of the platform it is created on, it is converted automatically by the system.

Application data, however, is not converted automatically. If character data is being exchanged between platforms where the *CodedCharSetId* and *Encoding* fields differ, for example, between ASCII and EBCDIC, it is the responsibility of the application to arrange for conversion of the message. Application data conversion may be performed by the queue manager itself or by a user exit program, referred to as a *data-conversion exit*. The queue manager can perform data conversion itself, using one of its built-in conversion routines, if the application data is in one of the built-in formats (such as MQFMT_STRING). This chapter discusses the data-conversion exit facility that MQSeries provides for when the application data is not in a built-in format.

Control may be passed to the data-conversion exit during an MQGET call. This avoids converting across different platforms before reaching the final destination. However, if the final destination is a platform that does not support data conversion on the MQGET, you must specify CONVERT(YES) on the sender channel that sends the data to its final destination. This ensures that MQSeries converts the data during transmission. In this case, your data-conversion exit must reside on the system where the sender channel is defined.

The MQGET call is issued directly by the application. Set the *CodedCharSetId* and *Encoding* fields in the MQMD to the character set and encoding required. If your application is using the same character set and encoding as the queue manager, you can set *CodedCharSetId* to MQCCSI_Q_MGR, and *Encoding* to MQENC_NATIVE. After the MQGET call completes, these fields have the values appropriate to the message data returned. These may differ from the values required if the conversion was not successful. Your application should reset these fields to the values required prior to each MQGET call.

The conditions required for the data-conversion exit to be called are defined for the MQGET call in the *MQSeries Application Programming Reference* manual.

For a description of the parameters that are passed to the data-conversion exit, and detailed usage notes, see the *MQSeries Application Programming Reference* manual for the MQ_DATA_CONV_EXIT call and the MQDXP structure.

Programs that convert application data between different machine encodings and CCSIDs must conform to the MQSeries data conversion interface (DCI).

This chapter introduces data-conversion exits, under these headings:

- “Invoking the data-conversion exit” on page 156
- “Writing a data-conversion exit program” on page 157
- “Writing a data-conversion exit program for MQSeries for AS/400” on page 161
- “Writing a data-conversion exit for MQSeries for OS/2 Warp” on page 162
- “Writing a data-conversion exit program for MQSeries for OS/390” on page 164

Data-conversion exits

- “Writing a data-conversion exit for MQSeries for Tandem NonStop Kernel” on page 165
- “Writing a data-conversion exit for MQSeries on UNIX systems and Compaq (DIGITAL) OpenVMS” on page 166
- “Writing a data-conversion exit for MQSeries for Windows NT” on page 171

Invoking the data-conversion exit

A data-conversion exit is a user-written exit that receives control during the processing of an MQGET call. The exit is invoked if the following are true:

- The MQGMO_CONVERT option is specified on the MQGET call.
- Some or all of the message data is not in the requested character set or encoding.
- The *Format* field in the MQMD structure associated with the message is not MQFMT_NONE.
- The *BufferLength* specified on the MQGET call is not zero.
- The message data length is not zero.
- The message contains data that has a user-defined format. The user-defined format can occupy the entire message, or be preceded by one or more built-in formats. For example, the user-defined format might be preceded by an MQFMT_DEAD_LETTER_HEADER format. The exit is invoked to convert only the user-defined format; the queue manager converts any built-in formats that precede the user-defined format.

A user-written exit can also be invoked to convert a built-in format, but this happens only if the built-in conversion routines cannot convert the built-in format successfully.

There are some other conditions, described fully in the usage notes of the MQ_DATA_CONV_EXIT call in the *MQSeries Application Programming Reference* manual.

See the *MQSeries Application Programming Reference* manual for details of the MQGET call. Data-conversion exits cannot use MQI calls, other than MQXCNVC.

A new copy of the exit is loaded when an application attempts to retrieve the first message that uses that *Format* since the application connected to the queue manager. A new copy may also be loaded at other times if the queue manager has discarded a previously-loaded copy.

The data-conversion exit runs in an environment similar to that of the program which issued the MQGET call. As well as user applications, the program can be an MCA (message channel agent) sending messages to a destination queue manager that does not support message conversion. The environment includes address space and user profile, where applicable. The exit cannot compromise the queue manager’s integrity, since it does not run in the queue manager’s environment.

In a client-server environment, the exit is loaded at the server, and conversion takes place there.

Data conversion on OS/390

On OS/390, you must also be aware of the following:

- Exit programs can be written in assembler language only.
- Exit programs must be re-entrant, and capable of running anywhere in storage.

- Exit programs must restore the environment on exit to that at entry, and must free any storage obtained.
- Exit programs must not WAIT, or issue ESTAEs or SPIEs.
- Exit programs are normally invoked as if by OS/390 LINK in:
 - Non-authorized problem program state
 - Primary address space control mode
 - Non cross-memory mode
 - Non access-register mode
 - 31 bit addressing mode
 - TCB-PRB mode
- When used by a CICS application, the exit is invoked by EXEC CICS LINK, and should conform to the CICS programming conventions. The parameters are passed by pointers (addresses) in the CICS communication area (COMMAREA). Although not recommended, user exit programs can also make use of CICS API calls, with the following caution:
 - Do not issue syncpoints, as the results could influence units of work declared by the MCA.
 - Do not update any resources controlled by a resource manager other than MQSeries for OS/390, including those controlled by CICS Transaction Server for OS/390.
- For distributed queuing without CICS, the exit is loaded from the data set referenced by the CSQXLIB DD statement. In other environments, the exit is loaded from the same place as application programs.
- For distributed queuing using CICS, data-conversion exits are not supported.

Writing a data-conversion exit program

For OS/390, you must write data-conversion exits in assembler language. For other platforms, it is recommended that you use the C programming language.

To help you create a data-conversion exit program, the following are supplied:

- A skeleton source file
- A convert characters call
- A utility that creates a fragment of code that performs data conversion on data type structures This utility takes C input only. On OS/390, it produces assembler code.

These are described in subsequent sections.

For the procedure for writing the programs see:

- “Writing a data-conversion exit program for MQSeries for AS/400” on page 161
- “Writing a data-conversion exit for MQSeries for OS/2 Warp” on page 162
- “Writing a data-conversion exit program for MQSeries for OS/390” on page 164
- “Writing a data-conversion exit for MQSeries for Tandem NonStop Kernel” on page 165
- “Writing a data-conversion exit for MQSeries on UNIX systems and Compaq (DIGITAL) OpenVMS” on page 166
- “Writing a data-conversion exit for MQSeries for Windows NT” on page 171

Skeleton source file

These can be used as your starting point when writing a data-conversion exit program. The files supplied are listed in Table 7 on page 158.

Writing a data-conversion exit

Table 7. Skeleton source files

Platform	File
AIX	amqsvfc0.c
AS/400	QMQMSAMP/QCSRC(AMQSVFC4)
AT&T GIS UNIX	amqsvfcx.c
Compaq Tru64 UNIX	amqsvfcx.c
Digital OpenVMS	AMQSVFCX.C
HP-UX	amqsvfc0.c
OS/2	AMQSVFC0.C
OS/390	CSQ4BAX8 (1) CSQ4BAX9 (2) CSQ4CAX9 (3)
SINIX and DC/OSx	amqsvfcx.c
Sun Solaris	amqsvfc0.c
Tandem NSK	amqsvfcn
Windows NT	amqsvfc0.c
Notes: <ol style="list-style-type: none">1. Illustrates the MQXCVNC call.2. A wrapper for the code fragments generated by the utility for use in all environments except CICS.3. A wrapper for the code fragments generated by the utility for use in the CICS environment.	

Convert characters call

The MQXCNVC (Convert characters) call may be used from within a data-conversion exit program to convert character message data from one character set to another. For certain multibyte character sets (for example, UCS2 character sets), the appropriate options must be used.

No other MQI calls can be made from within the exit; an attempt to make such a call fails with reason code MQRC_CALL_IN_PROGRESS.

See the *MQSeries Application Programming Reference* manual for further information on the MQXCNVC call and appropriate options.

Utility for creating conversion-exit code

The commands for creating conversion-exit code are:

AS/400

CVTMQMMDTA (Convert MQSeries Data Type)

OS/2, Digital OpenVMS, Tandem NSK, Windows NT, and UNIX systems

crtmqcvx (Create MQSeries conversion-exit)

OS/390

CSQUCVX

The command for your platform produces a fragment of code that performs data conversion on data type structures, for use in your data-conversion exit program. The command takes a file containing one or more C language structure definitions. On OS/390, it then generates a data set containing assembler code fragments and

conversion functions. On other platforms, it generates a file with a C function to convert each structure definition. The utility requires access to the LE/370 run-time library SCEERUN.

Invoking the CSQUCVX utility on OS/390

Figure 13 shows an example of the JCL used to invoke the CSQUCVX utility.

```
//CVX      EXEC PGM=CSQUCVX
//STEPLIB DD DISP=SHR,DSN=th1qual.SCSQANLE
//         DD DISP=SHR,DSN=th1qual.SCSQLOAD
//         DD DISP=SHR,DSN=1e370qual.SCEERUN
//SYSPRINT DD SYSOUT=*
//CSQUINP DD DISP=SHR,DSN=MY.MQSERIES.FORMATS(MSG1)
//CSQUOUT DD DISP=OLD,DSN=MY.MQSERIES.EXIT(S(MSG1)
```

Figure 13. Sample JCL used to invoke the CSQUCVX utility

Data definition statements

The CSQUCVX utility requires DD statements with the following DDnames:

SYSPRINT

This specifies a data set or print spool class for reports and error messages.

CSQUINP

This specifies the sequential data set containing the definitions of the data structures to be converted.

CSQUOUT

This specifies the sequential data set where the conversion code fragments are to be written. The logical record length (LRECL) must be 80 and the record format (RECFM) must be FB.

Error messages in OS/2, Windows NT, and UNIX systems

The `crtmqcvx` command returns messages in the range AMQ7953 through AMQ7970. For other platforms, see the appropriate *System Management Guide* for your platform.

There are two main types of error:

- Major errors, such as syntax errors, when processing cannot continue.
A message is displayed on the screen giving the line number of the error in the input file. The output file may have been partially created.
- Other errors when a message is displayed stating that a problem has been found but parsing of the structure can continue.

The output file has been created and contains error information on the problems that have occurred. This error information is prefixed by `#error` so that the code produced will not be accepted by any compiler without intervention to rectify the problems.

Valid syntax

Your input file for the utility must conform to the C language syntax. If you are unfamiliar with C, refer to “Example of valid syntax for the input data set” on page 160.

In addition, you must be aware of the following rules:

- `typedef` is recognized only before the `struct` keyword.
- A structure tag is required on your structure declarations.

Writing a data-conversion exit

- Empty square brackets [] may be used to denote a variable length array or string at the end of a message.
- Multidimensional arrays and arrays of strings are not supported.
- The following additional data types are recognized:
 - MQBYTE
 - MQCHAR
 - MQSHORT
 - MQLONG

MQCHAR fields are code page converted, but MQBYTE is left untouched. If the encoding is different, MQSHORT and MQLONG are converted accordingly.

- The following should *not* be used:
 - float
 - double
 - pointers
 - bit-fields

This is because the utility for creating conversion-exit code does not provide the facility to convert these data types. To overcome this, you can write your own routines and call them from the exit.

Other points to note:

- Do not use sequence numbers in the input data set.
- If there are fields for which you want to provide your own conversion routines, declare them as MQBYTE, and then replace the generated CMQXCFBA macros with your own conversion code.

Example of valid syntax for the input data set

```
struct TEST { MQLONG   SERIAL_NUMBER;
              MQCHAR   ID[5];
              MQSHORT  VERSION;
              MQBYTE   CODE[4];
              MQLONG   DIMENSIONS[3];
              MQCHAR   NAME[24];
            } ;
```

This corresponds to the following declarations in the other programming languages:

COBOL:

```
10 TEST.
  15 SERIAL-NUMBER PIC S9(9) BINARY.
  15 ID            PIC X(5) .
  15 VERSION      PIC S9(4) BINARY.
  * CODE IS NOT TO BE CONVERTED
  15 CODE         PIC X(4) .
  15 DIMENSIONS   PIC S9(9) BINARY OCCURS 3 TIMES.
  15 NAME         PIC X(24) .
```

System/390 assembler: *Supported on OS/390 only*

```
TEST          EQU *
SERIAL_NUMBER DS F
ID            DS CL5
VERSION       DS H
CODE          DS XL4
DIMENSIONS    DS 3F
NAME          DS CL24
```

PL/I: *Supported on AIX, OS/390, OS/2 Warp, and Windows NT only*

```
DCL 1 TEST,
    2 SERIAL_NUMBER  FIXED BIN(31),
    2 ID              CHAR(5),
    2 VERSION         FIXED BIN(15),
    2 CODE             CHAR(4),          /* not to be converted */
    2 DIMENSIONS(3)  FIXED BIN(31),
    2 NAME            CHAR(24);
```

Writing a data-conversion exit program for MQSeries for AS/400

Follow these steps:

1. Name your message format. The name must fit in the *Format* field of the MQMD. The *Format* name should not have leading embedded blanks, and trailing blanks are ignored. The object's name must have no more than eight non-blank characters, because the *Format* is only eight characters long. Remember to use this name each time you send a message (our example uses the name *Format*).

2. Create a structure to represent your message. See "Valid syntax" on page 159 for an example.

3. Run this structure through the CVTMQMMDTA command to create a code fragment for your data-conversion exit.

The functions generated by the CVTMQMMDTA command use macros that are shipped in the file QMQM/H(AMQSVMHHA). These macros are written assuming that all structures are packed; they should be amended if this is not the case.

4. Take a copy of the supplied skeleton source file, QMQMSAMP/QCSRC(AMQSVFC4) and rename it. (Our example uses the name EXIT_MOD.)
5. Find the following comment boxes in the source file and insert code as described:

- a. Towards the bottom of the source file, a comment box starts with:


```
/* Insert the functions produced by the data-conversion exit */
```

Here, insert the code fragment generated in step 3.

- b. Near the middle of the source file, a comment box starts with:


```
/* Insert calls to the code fragments to convert the format's */
```

This is followed by a commented-out call to the function ConverttagSTRUCT.

Change the name of the function to the name of the function you added in step 5a above. Remove the comment characters to activate the function. If there are several functions, create calls for each of them.

- c. Near the top of the source file, a comment box starts with:


```
/* Insert the function prototypes for the functions produced by */
```

Here, insert the function prototype statements for the functions added in step 5a above.

If the message contains character data, the generated code calls MQXCNVC; this can be resolved by binding the service program QMQM/LIBMQM.

MQSeries for AS/400 data-conversion exit

6. Compile the source module, EXIT_MOD, as follows:

```
CRTCMOD MODULE(library/EXIT_MOD) +  
SRCFILE(QCSRC) +  
TERASPACE(*YES *TSIFC)
```

7. Create/link the program.

For nonthreaded applications, use the following:

```
CRTPGM PGM(library/Format) +  
MODULE(library/EXIT_MOD) +  
BNDSRVPGM(QMQM/LIBMQM) +  
ACTGRP(QMQM) +  
USRPRF(*USER)
```

In addition to creating the data-conversion exit for the basic environment, another is required in the threaded environment. This loadable object must be followed by _R. The LIBMQM_R library should be used to resolve calls to the MQXCNV. Both loadable objects are required for a threaded environment.

```
CRTPGM PGM(library/Format_R) +  
MODULE(library/EXIT_MOD) +  
BNDSRVPGM(QMQM/LIBMQM_R) +  
ACTGRP(QMQM) +  
USRPRF(*USER)
```

8. Place the output in the library list for the MQSeries job. It is recommended that, for production, data-conversion exit programs be stored in QSYS.

Notes:

1. If CVTMQMDTA uses packed structures, all MQSeries applications must use the `_Packed` qualifier.
2. Data-conversion exit programs must be re-entrant.
3. MQXCNV is the *only* MQI call that may be issued from a data-conversion exit.
4. The exit program should be compiled with the user profile compiler option set to *USER, so that the exit runs with the authority of the user.
5. Teraspace memory enablement is required for all user exits with Version 5.1 of MQSeries for AS/400, and TERASPACE(*YES *TSIFC) must be specified in the CRTCMOD and CRTBNDC commands.

Writing a data-conversion exit for MQSeries for OS/2 Warp

Follow these steps:

1. Name your message format. The name must fit in the *Format* field of the MQMD. The *Format* name should not have leading blanks. Trailing blanks are ignored. The object's name must have no more than eight non-blank characters, because the *Format* is only eight characters long.
A .DEF file called AMQSVFC2.DEF is also supplied in the samples directory, <drive:\directory>\MQM\TOOLS\C\SAMPLES. Take a copy of this file and rename it, for example, to MYFORMAT.DEF. Make sure that the name of the DLL being created and the name specified in MYFORMAT.DEF are the same. Overwrite the name FORMAT1 in MYFORMAT.DEF with the new format name. Remember to use this name each time you send a message.
2. Create a structure to represent your message. See "Valid syntax" on page 159 for an example.
3. Run this structure through the CRTMQCVX command to create a code fragment for your data-conversion exit.

MQSeries for OS/2 Warp data-conversion exit

The functions generated by the CRTMQCVX command use macros which are written assuming that all structures are packed; they should be amended if this is not the case.

4. Take a copy of the supplied skeleton source file, AMQSVFC0.C, renaming it to the name of your message format that you decided on in step 1 (that is, MYFORMAT.C in this example). AMQSVFC0.C is in <drive:\directory>\MQM\TOOLS\C\SAMPLES (where <drive:\directory> was specified at installation).

The skeleton includes a sample header file AMQSVMHA.H in the same directory. Make sure that your include path points to this directory to pick up this file.

The AMQSVMHA.H file contains macros that are used by the code generated by the CRTMQCVX command. If the structure to be converted contains character data, then these macros call MQXCNVC.

5. Find the following comment boxes in the source file and insert code as described:

- a. Towards the bottom of the source file, a comment box starts with:

```
/* Insert the functions produced by the data-conversion exit */
```

Here, insert the code fragment generated in step 3.

- b. Near the middle of the source file, a comment box starts with:

```
/* Insert calls to the code fragments to convert the format's */
```

This is followed by a commented-out call to the function ConverttagSTRUCT.

Change the name of the function to the name of the function you added in step 5a above. Remove the comment characters to activate the function. If there are several functions, create calls for each of them.

- c. Near the top of the source file, a comment box starts with:

```
/* Insert the function prototypes for the functions produced by */
```

Here, insert the function prototype statements for the functions added in step 5a above.

6. Resolve this call by linking the routine with the library MQMVX.LIB, in the directory <drive:\directory>\MQM\TOOLS\LIB.
7. Create the following command file:

```
icc /Ge- /I<drive:\directory>\mqm\tools\c\samples \
/I<drive:\directory>\mqm\tools\c\include MYFORMAT.C \
<drive:\directory>\mqm\tools\lib\mqm.lib MYFORMAT.DEF \
<drive:\directory>\mqm\tools\lib\mqmvx.lib
```

where <drive:\directory> is specified at installation.

Issue the command file to compile your exit as a DLL file.

8. Place the output in the \mqm\exits subdirectory. The path used to look for the data-conversion exits is given in the qm.ini file as DefaultExitPath. This path is set for each queue manager and the exit will only be looked for in that path or paths.

Notes:

1. If CVTMQCVX uses packed structures, all MQSeries applications must be compiled in this way.
2. Data-conversion exit programs must be re-entrant.

MQSeries for OS/2 Warp data-conversion exit

3. MQXCNVC is the *only* MQI call that may be issued from a data-conversion exit.

Writing a data-conversion exit program for MQSeries for OS/390

Follow these steps:

1. Take the supplied source skeleton CSQ4BAX9 (for non-CICS environments) or CSQ4CAX9 (for CICS) as your starting point.
2. Run the CSQUCVX utility.
3. Follow the instructions in the prolog of CSQ4BAX9 or CSQ4CAX9 to incorporate the routines generated by the CSQUCVX utility, in the order that the structures occur in the message you want to convert.
4. The utility assumes that the data structures are not packed, that the implied alignment of the data is honored, and that the structures start on a full-word boundary, with bytes being skipped as required (as between ID and VERSION in the “Example of valid syntax for the input data set” on page 160). If the structures are packed, you will need to omit the CMQXCALA macros that are generated. You are therefore strongly recommended to declare your structures in such a way that all fields are named and no bytes are skipped; in the “Example of valid syntax for the input data set” on page 160, you would add a field “MQBYTE DUMMY;” between ID and VERSION.
5. The supplied exit returns an error if the input buffer is shorter than the message format to be converted. Although the exit converts as many complete fields as possible, the error causes an unconverted message to be returned to the application. If you want to allow short input buffers to be converted as far as possible, including partial fields, change the TRUNC= value on the CSQXCDDFA macro to YES: no error is returned, so the application receives a converted message. The application is responsible for handling the truncation.
6. Add any other special processing code that you need.
7. Rename the program to your data format name.
8. Compile and link-edit your program like a batch application program (unless it is for use with CICS applications). The macros in the code generated by the utility are in the library, **thlqual.SCSQMACS**.
If the message contains character data, the generated code will call MQXCNVC. If your exit uses this call, link-edit it with the exit stub program CSQASTUB. The stub is language-independent and environment-independent. Alternatively, you can load the stub dynamically using the dynamic call name CSQXCNVC. See “Dynamically calling the MQSeries stub” on page 315 for more information.
Place the link-edited module in your application load library, and in a data set that is referenced by the CSQXLIB DD statement of your task procedure started by your channel initiator.
9. If the exit is for use by CICS applications, compile and link-edit it like a CICS application program, including CSQASTUB if required. Place it in your CICS application program library. Define the program to CICS in the usual way, specifying EXECKEY(CICS) in the definition.

Note: Although the LE/370 run-time libraries are needed for running the CSQUCVX utility (see step 2), they are not needed for link-editing or running the data-conversion exit itself (see steps 8 and 9).

See “Writing MQSeries-IMS bridge applications” on page 269 for information about data conversion within the MQSeries-IMS bridge.

Writing a data-conversion exit for MQSeries for Tandem NonStop Kernel

Dynamically bound libraries are not supported by MQSeries for Tandem NonStop Kernel. Data conversion exits (and channel exits) are implemented by including statically bound stub functions in the MQSeries libraries and executables that can be replaced using the REPLACE bind option.

A data conversion exit **must** be called DATACONVEXIT (see sample AMQSVFCN), and can be bound into the chosen executable (or library) using the TACL macro BEXITE.

Note: This procedure modifies the target executable; you are recommended to make a back-up copy of the target executable or library before using the macro.

Exit functions, once compiled, must be bound directly into the target executable or library to be accessible by MQSeries. The following TACL macro is used for this purpose:

BEXITE

Usage: BEXITE *target-executable-or-library source-exit-file-or-library*

For example, to bind the sample data conversion exit into the sample MQSGETA, follow these steps:

1. Compile the exit function DATACONVEXIT (CSAMP AMQSVFCN).
2. Compile the get application (CSAMP AMQSGET0).
3. Bind the get application (BSAMP AMQSGET).
4. Bind the exit function into the get application (BEXITE AMQSGET AMQSVFCO).

Alternatively, if all applications are to have this data conversion exit, the following steps would create both a user library and an application with the exit bound in:

1. Compile the exit function DATACONVEXIT (CSAMP AMQSVFCN).
2. Compile the get application (CSAMP AMQSGET0).
3. Bind the exit function into the user library (BEXITE ZMQSLIB.MQMLIBC AMQSVFCO).
4. Bind the get application with the modified library (BSAMP AMQSGET).

If the data conversion exit is to be used by channels processing within MQSeries, it must also be bound into the caller executable by the system administrator. For example:

```
BEXITE ZMQSEXE.MQMCACAL AMQSVFCO
```

Use the TACL macro BDCXALL to bind the data conversion exit into all required MQSeries processes. For example:

```
BDCXALL source-exit-file-or-library
```

Reusing data-conversion exit programs

In other MQSeries Version 2 products, a data-conversion exit is required for each application-defined format to be supported. The data-conversion exit programs are named according to the *Format* value (from MQMD) of the message to be

MQSeries for Tandem NonStop Kernel data-conversion exit

converted. The format for which conversion is being requested can be determined from the *Format* field of the *MsgDesc* parameter. The appropriate data-conversion exit program can therefore be invoked from MQDATACONVEXIT(). The parameters supplied to MQDATACONVEXIT() can be supplied to the invoked data-conversion function.

Writing a data-conversion exit for MQSeries on UNIX systems and Compaq (DIGITAL) OpenVMS

For SINIX and DC/OSx, data-conversion exits must not use DCE.

Follow these steps:

1. Name your message format. The name must fit in the *Format* field of the MQMD, and be in uppercase, for example, MYFORMAT. The *Format* name should not have leading blanks. Trailing blanks are ignored. The object's name must have no more than eight non-blank characters because the *Format* is only eight characters long. Remember to use this name each time you send a message.
2. Create a structure to represent your message. See "Valid syntax" on page 159 for an example.
3. Run this structure through the **crtmqcvx** command to create a code fragment for your data-conversion exit.

The functions generated by the **crtmqcvx** command use macros which are written assuming that all structures are packed; they should be amended if this is not the case.

4. Take a copy of the supplied skeleton source file renaming it to the name of your message format that you decided on in step 1 (that is, MYFORMAT.C).

Note: On MQSeries for AIX, Compaq Tru64 UNIX, HP-UX, and Sun Solaris the skeleton source file is called amqsvfc0.c. On MQSeries for AT&T GIS UNIX, Compaq (DIGITAL) OpenVMS, and SINIX and DC/OSx the skeleton source file is called amqsvfcx.c.

The skeleton includes a sample header file amqsvmha.h in the directory /usr/mqm/inc (on AIX) or /opt/mqm/inc (on other UNIX systems). Make sure that your include path points to this directory to pick up this file.

The amqsvmha.h file contains macros that are used by the code generated by the **crtmqcvx** command. If the structure to be converted contains character data, then these macros call MQXCNVC.

5. Find the following comment boxes in the source file and insert code as described:

- a. Towards the bottom of the source file, a comment box starts with:
/* Insert the functions produced by the data-conversion exit */

Here, insert the code fragment generated in step 3.

- b. Near the middle of the source file, a comment box starts with:
/* Insert calls to the code fragments to convert the format's */

This is followed by a commented-out call to the function ConverttagSTRUCT.

MQSeries on UNIX systems, Compaq (DIGITAL) OpenVMS data-conversion exit

Change the name of the function to the name of the function you added in step 5a above. Remove the comment characters to activate the function. If there are several functions, create calls for each of them.

- c. Near the top of the source file, a comment box starts with:

```
/* Insert the function prototypes for the functions produced by */
```

Here, insert the function prototype statements for the functions added in step 5a above.

6. Resolve this call by linking the routine with the library libmqm. For threaded programs, the routine must be linked with the library libmqm_r (AIX and HP-UX only).
7. Compile your exit as a shared library, using MQStart as the entry point. To do this, see "Compiling data-conversion exits on UNIX" on page 168, or "Compiling data-conversion exits on Digital OpenVMS" on page 168.
8. Place the output in the default system directory, /var/mqm/exits, to ensure that it can be loaded when required. The path used to look for the data-conversion exits is given in the qm.ini file. This path can be set for each queue manager and the exit is only looked for in that path or paths.

Notes:

1. If **crtmqcvx** uses packed structures, all MQSeries applications must be compiled in this way.
2. Data-conversion exit programs must be re-entrant.
3. MQXCNVC is the *only* MQI call that may be issued from a data-conversion exit.

UNIX environment

There are two environments to consider: non threaded and threaded.

Non-threaded environment

The loadable object must have its name in upper case, for example MYFORMAT. The libmqm library should be used to resolve the calls to MQXCNVC.

Threaded environment

In addition to creating the data-conversion exit for the basic environment, another is required in the threaded environment. This loadable object must be followed by `_r` (on AIX, HP-UX, and Tru64 V4.0) and `_d` (on Sun Solaris) to indicate that it is a DCE-threaded version. The libmqm_r library (on AIX, HP-UX, and Tru64 V4.0) and the lmqmcs_d library (on Sun Solaris) should be used to resolve the calls to MQXCNVC. Note that both loadable objects (non-threaded and threaded) are required for a threading environment.

If you are running MQI clients, all data conversion is performed by the proxy running on the machine to which the client is attached. This means that any data conversion exits are run on the server, in the environment of the proxy, and not as part of the client application.

For most platforms, the proxy/responder program is a threaded program. Consequently, the data conversion exit must be compiled with appropriate options to run in this threaded environment. Whether or not the client application is threaded is irrelevant.

On the MQSeries V5 for UNIX systems, the proxy is threaded. The model of threads used depends on whether the DCE option has been installed.

MQSeries on UNIX systems, Compaq (DIGITAL) OpenVMS data-conversion exit

Note: If the data-conversion exits are in a mixed non-threaded and threaded environment, the calling environment is detected and the appropriate object loaded. The shared object should be placed in /var/mqm/exits to ensure it can be loaded when required.

Compiling data-conversion exits on Digital OpenVMS

The names of the routines which are called by the data-conversion exit must be made universal.

```
$ CC /INCLUDE_DIRECTORY=MQS_INCLUDE AMQSVFCX.C
$ LINK /SYS$SHARE:[SYSLIB]MYFORMAT AMQSVFCX.OBJ,MYFORMAT/OPTIONS
```

The contents of MYFORMAT.OPT vary depending on which platform you are working on:

On Alpha:

```
SYS$SHARE:MQM/SHAREABLE
SYS$SHARE:MQMCS/SHAREABLE
SYMBOL_VECTOR=(MQSTART=PROCEDURE)
```

On VAX:

```
SYS$SHARE:MQM/SHAREABLE
SYS$SHARE:MQMCS/SHAREABLE
UNIVERSAL=MQSTART
```

If you are using threaded applications linked with the pthread library, you must also build a second copy of the data-conversion exit with the thread options and libraries:

```
$ CC /INCLUDE_DIRECTORY=MQS_INCLUDE AMQSVFCX.C
$ LINK /SYS$SHARE:[SYSLIB]MYFORMAT AMQSVFCX.OBJ,MYFORMAT/OPTIONS
```

Again, the contents of MYFORMAT.OPT vary depending on which platform you are working on:

On Alpha:

```
SYS$SHARE:MQM_R/SHAREABLE
SYS$SHARE:MQMCS_R/SHAREABLE
SYS$SHARE:CMA$OPEN_RTL.EXE/SHAREABLE
SYMBOL_VECTOR-(MQSTART=PROCEDURE)
```

On VAX:

```
SYS$SHARE:MQM_R/SHAREABLE
SYS$SHARE:MQMCS_R/SHAREABLE
SYS$SHARE:CMA$OPEN_RTL.EXE/SHAREABLE
UNIVERSAL=MQSTART
```

Compiling data-conversion exits on UNIX

The following sections give examples of how to compile a data conversion exit on the UNIX platforms.

On all platforms, the entry point to the module is MQStart.

On AIX 4.2

```
$ cc -c -I/usr/mqm/inc MYFORMAT.C
$ ld MYFORMAT.o -e MQStart -o MYFORMAT -b:SRE -H512 -T512 -lmqm -lc
$ cp MYFORMAT /var/mqm/exits
```

MQSeries on UNIX systems, Compaq (DIGITAL) OpenVMS data-conversion exit

If you are using threaded applications linked with the pthreads library or you are running client applications, you must build a second copy of the conversion exit with the thread options and libraries.

```
$ cc_r -c -I/usr/mqm/inc MYFORMAT.C
$ ld MYFORMAT.o -e MQStart -o MYFORMAT_r -bM:SRE -H512 \
-T512 -lmqm_r -lpthreads -lc_r
$ cp MYFORMAT_r /var/mqm/exits
```

On AIX 4.3

```
$ cc -c -I/usr/mqm/inc MYFORMAT.C
$ ld MYFORMAT.o -e MQStart -o MYFORMAT -bM:SRE -H512 -T512 -lmqm -lc
$ cp MYFORMAT /var/mqm/exits
```

You must build conversion exits for the threaded environment using the draft 7 Posix threads interface rather than the draft 10 interface which is the AIX 4.3 default.

```
$ xlc_r7 -c -I/usr/mqm/inc MYFORMAT.C
$ ld MYFORMAT.o -eMQStart -o MYFORMAT_r -bM:SRE -H512 -T512 \
-lmqm_r -lpthreads_compat -lpthreads -lc_r
$ cp MYFORMAT_r /var/mqm/exits
```

On AT&T GIS UNIX

```
$ cc -c -K PIC -I/opt/mqm/inc MYFORMAT.C
$ ld -G MYFORMAT.O -o MYFORMAT
$ cp MYFORMAT /opt/mqm/lib
```

On Compaq Tru64 UNIX Version 4.0

This example shows how to compile a data-conversion exit program in a nonthreaded environment:

```
$ cc -std1 -c -I /opt/mqm/inc myformat.c
$ cc -std1 -shared -o myformat myformat.o -L /opt/mqm/lib -lmqm -e MQStart -lc
$ cp myformat /var/mqm/exits
```

This example shows how to compile a data-conversion exit program in a threaded environment:

```
$ cc -std1 -c -I /opt/mqm/inc myformat.c
$ cc -std1 -shared -pthread -o myformat_r myformat.o -L /opt/mqm/lib \
-lmqm_r -e MQStart -lc
$ cp myformat /var/mqm/exits
```

On Compaq Tru64 UNIX Version 5.0

This example shows how to compile a data-conversion exit program:

```
$ cc -std1 -c -I /opt/mqm/inc myformat.c
$ cc -std1 -shared -pthread -o myformat myformat.o -L /opt/mqm/lib \
-lmqm -e MQStart -lc
$ cp myformat /var/mqm/exits
```

On HP-UX Version 10.20

```
$ CC -c -Aa +z -I/opt/mqm/inc MYFORMAT.C
$ ld -b MYFORMAT.o -o MYFORMAT -L /opt/mqm/lib -lmqm +IMQStart
$ cp MYFORMAT /var/mqm/exits
```

If you are using threaded applications linked with the pthreads library or you are running client applications, you must build a second copy of the conversion exit with the thread options and libraries.

```
$ CC -c -Aa +z -I/opt/mqm/inc MYFORMAT.C
$ ld -b MYFORMAT.o -o MYFORMAT_r -L /opt/mqm/lib \
-lmqm_r -lcma -lc_r +IMQStart
$ cp MYFORMAT_r /var/mqm/exits
```

MQSeries on UNIX systems, Compaq (DIGITAL) OpenVMS data-conversion exit

On HP-UX Version 11.00

```
$ CC -c -Aa +z -I/opt/mqm/inc MYFORMAT.C  
$ ld -b MYFORMAT.o -o MYFORMAT -L /opt/mqm/lib -lmqm +IMQStart  
$ cp MYFORMAT /var/mqm/exits
```

If you are using threaded applications linked with the POSIX Draft 10 pthreads library, or you are running client applications, you must build the conversion exit for Draft 10 threads.

```
$ CC -c -Aa +z -I/opt/mqm/inc MYFORMAT.C  
$ ld -b MYFORMAT.o -o MYFORMAT_r -L/opt/mqm/lib -lmqm_r -lpthread -lc  
+IMQStart  
$ cp MYFORMAT_r /var/mqm/exits
```

If you are using threaded applications linked with the POSIX Draft 4 (DCE) pthreads library, or you are running client applications, you must build the conversion exit for Draft 4 threads.

```
$ CC -c -Aa +z -I/opt/mqm/inc -D_PTHREADS_DRAFT4 MYFORMAT.C  
$ ld -b MYFORMAT.o -o MYFORMAT_d -L/opt/mqm/lib -lmqm_d -ldr -lcma -lc  
+IMQStart  
$ cp MYFORMAT_d /var/mqm/exits
```

On SINIX

```
$ cc -c -K PIC -I/opt/mqm/inc -lproc -lext MYFORMAT.C  
$ ld -G MYFORMAT.o -o MYFORMAT  
$ cp MYFORMAT /opt/mqm/lib
```

On DC/OSx

```
$ cc -c -K PIC -I/opt/mqm/inc -liconv -lproc -lext MYFORMAT.C  
$ ld -G MYFORMAT.o -o MYFORMAT  
$ cp MYFORMAT /opt/mqm/lib
```

On Sun Solaris

If your application uses no threading calls or Posix V10 threading calls:

```
cc -c -KPIC -I/opt/mqm/inc MYFORMAT.C  
  
ld -G /opt/SUNWspro/SC4.0/lib/crt1.o \  
/opt/SUNWspro/SC4.0/lib/crti.o \  
/opt/SUNWspro/SC4.0/lib/crtn.o \  
/opt/SUNWspro/SC4.0/lib/values-xt.o \  
MYFORMAT.o -o MYFORMAT -lmqm -lthread -lsocket -lc -lnsl -ldl  
  
cp MYFORMAT /var/mqm/exits
```

If your application requires DCE threading (for example, if it is a CICS application):

```
cc -c -KPIC -I/opt/mqm/inc MYFORMAT.C  
  
ld -G /opt/SUNWspro/SC4.0/lib/crt1.o \  
/opt/SUNWspro/SC4.0/lib/crti.o \  
/opt/SUNWspro/SC4.0/lib/crtn.o \  
/opt/SUNWspro/SC4.0/lib/values-xt.o \  
MYFORMAT.o -o MYFORMAT_d -ldce -lnsl -lthread -lm -lsocket \  
-lmqmc_d -lmqm -lc -ldl  
  
cp MYFORMAT /var/mqm/exits
```

Note: The SC4.0 directory name varies depending on the release of compiler.

MQSeries on UNIX systems, Compaq (DIGITAL) OpenVMS data-conversion exit

If you want to run applications using both the Posix V10-threaded and the DCE-threaded variants on a single queue manager:

1. Build a Posix V10 type of data-conversion exit. Name it MYFORMAT and place it in the appropriate exit directory.
2. Build a DCE-threaded type of data-conversion exit. Name it MYFORMAT_d and place it in the appropriate exit directory.

Two object files are generated; one of which loads the MYFORMAT data-conversion exit, and the other of which loads the MYFORMAT_d data-conversion exit.

Writing a data-conversion exit for MQSeries for Windows NT

Follow these steps:

1. Name your message format. The name must fit in the *Format* field of the MQMD. The *Format* name should not have leading blanks. Trailing blanks are ignored. The object's name must have no more than eight non-blank characters, because the *Format* is only eight characters long.

A .DEF file called amqsvfcn.def is also supplied in the samples directory, <drive:\directory>\Program Files\MQSeries\Tools\C\Samples. Take a copy of this file and rename it, for example, to MYFORMAT.DEF. Make sure that the name of the DLL being created and the name specified in MYFORMAT.DEF are the same. Overwrite the name FORMAT1 in MYFORMAT.DEF with the new format name.

Remember to use this name each time you send a message.

2. Create a structure to represent your message. See "Valid syntax" on page 159 for an example.
3. Run this structure through the CRTMQCVX command to create a code fragment for your data-conversion exit.

The functions generated by the CVTMQCVX command use macros which are written assuming that all structures are packed; they should be amended if this is not the case.

4. Take a copy of the supplied skeleton source file, amqsvfc0.c, renaming it to the name of your message format that you decided on in step 1 (that is, MYFORMAT).

amqsvfc0.c is in

<drive:\directory>\Program Files\MQSeries\Tools\C\Samples (where <drive:\directory> was specified at installation).

The skeleton includes a sample header file amqsvmha.h in the same directory. Make sure that your include path points to this directory to pick up this file.

The amqsvmha.h file contains macros that are used by the code generated by the CRTMQCVX command. If the structure to be converted contains character data, then these macros call MQXCNVC.

5. Find the following comment boxes in the source file and insert code as described:

- a. Towards the bottom of the source file, a comment box starts with:

```
/* Insert the functions produced by the data-conversion exit */
```

Here, insert the code fragment generated in step 3.

- b. Near the middle of the source file, a comment box starts with:

```
/* Insert calls to the code fragments to convert the format's */
```

MQSeries for Windows NT data-conversion exit

This is followed by a commented-out call to the function `ConverttagSTRUCT`.

Change the name of the function to the name of the function you added in step 5a above. Remove the comment characters to activate the function. If there are several functions, create calls for each of them.

- c. Near the top of the source file, a comment box starts with:

```
/* Insert the function prototypes for the functions produced by */
```

Here, insert the function prototype statements for the functions added in step 5a above.

6. Resolve this call by linking the routine with the library `MQMVX.LIB`, in the directory `<drive:\directory>\Program Files\MQSeries\Tools\Lib`.
7. Create the following command file:

```
cl -I <drive:\directory>\Program Files\MQSeries\Tools\C\Include -Tp \
MYFORMAT.C -LD -DEFAULTLIB \
<drive:\directory>\Program Files\MQSeries\Tools\Lib\mqm.lib \
<drive:\directory>\Program Files\MQSeries\Tools\Lib\mqmvx.lib \
MYFORMAT.DEF
```

where `<drive:\directory>` is specified at installation,

Issue the command file to compile your exit as a DLL file.

8. Place the output in the `C:\WINNT\Profiles\All Users\Application Data\MQSeries\EXITS` subdirectory. The path used to look for the data-conversion exits is given in the registry. This path can be set for each queue manager and the exit is only looked for in that path or paths.

Notes:

1. If `CVTMQCVX` uses packed structures, all MQSeries applications must be compiled in this way.
2. Data-conversion exit programs must be re-entrant.
3. `MQXCNVC` is the *only* MQI call that may be issued from a data-conversion exit.

Chapter 12. Inquiring about and setting object attributes

Attributes are the properties that define the characteristics of an MQSeries object. They affect the way that an object is processed by a queue manager. The attributes of each type of MQSeries object are described in detail in the *MQSeries Application Programming Reference* manual.

Some attributes are set when the object is defined, and can be changed only by using the MQSeries commands; an example of such an attribute is the default priority for messages put on a queue. Other attributes are affected by the operation of the queue manager and may change over time; an example is the current depth of a queue.

You can inquire about the current values of all these attributes using the MQINQ call. The MQI also provides an MQSET call with which you can change some queue attributes. You cannot use the MQI calls to change the attributes of any other type of object; instead you must use:

For MQSeries for OS/390

The ALTER operator commands (or the DEFINE commands with the REPLACE option), which are described in the *MQSeries MQSC Command Reference*.

For MQSeries for AS/400

The CHGMQMx CL commands, which are described in the *MQSeries for AS/400 System Administration* book, or you can use the MQSC facility.

For MQSeries for Tandem NonStop Kernel

The MQM screen-based interface, which is described in the *MQSeries for Tandem NonStop Kernel System Management Guide*, or you can use the MQSC facility.

For MQSeries for VSE/ESA

The panel interface, which is described in the *MQSeries for VSE/ESA System Management Guide*.

For MQSeries for all other platforms

The MQSC facility, described in the *MQSeries MQSC Command Reference*.

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 must identify the attributes using the keywords shown in the descriptions of the commands in the above books.

Both the MQINQ and the MQSET calls use arrays of selectors to identify those attributes you want to inquire about or set. There is a selector for each attribute you can work with. The selector name has a prefix, determined by the nature of the attribute:

MQCA_

These selectors refer to attributes that contain character data (for example, the name of a queue).

MQIA_

These selectors refer to attributes that contain either numeric values (such

Object attributes

as *CurrentQueueDepth*, the number of messages on a queue) or a constant value (such as *SyncPoint*, whether or not the queue manager supports syncpoints).

Before you use the MQINQ or MQSET calls your application must be connected to the queue manager, and you must use the MQOPEN call to open the object for setting or inquiring about attributes. These operations are described in “Chapter 7. Connecting and disconnecting a queue manager” on page 89 and “Chapter 8. Opening and closing objects” on page 97.

Inquiring about the attributes of an object

Use the MQINQ call to inquire about the attributes of any type of MQSeries object.

As input to this call, you must supply:

- A connection handle.
- An object handle.
- The number of selectors.
- An array of attribute selectors, each selector having the form MQCA_* or MQIA_*. Each selector represents an attribute whose value you want to inquire about, and each selector must be valid for the type of object that the object handle represents. You can specify selectors in any order.
- The number of integer attributes that you are inquiring about. Specify zero if you are not inquiring about integer attributes.
- The length of the character attributes buffer in *CharAttrLength*. This must be at least the sum of the lengths required to hold each character attribute string. Specify zero if you are not inquiring about character attributes.

The output from MQINQ is:

- A set of integer attribute values copied into the array. The number of values is determined by *IntAttrCount*. If either *IntAttrCount* or *SelectorCount* is zero, this parameter is not used.
- The buffer in which character attributes are returned. The length of the buffer is given by the *CharAttrLength* parameter. If either *CharAttrLength* or *SelectorCount* is zero, this parameter is not used.
- A completion code. If the completion code gives a warning, this means that the call completed only partially. In this case, you should examine the reason code.
- A reason code. There are three partial-completion situations:
 - The selector does not apply to the queue type
 - There is not enough space allowed for integer attributes
 - There is not enough space allowed for character attributes

If more than one of these situations arise, the first one that applies is returned.

Namelist can be inquired only on AIX, AS/400, HP-UX, OS/2 Warp, OS/390, Sun Solaris, Compaq Tru64 UNIX and Windows NT.

If you open a queue for output or inquire and it resolves to a non-local cluster queue you can only inquire the queue name, queue type, and common attributes. The values of the common attributes are those of the chosen queue if MQOO_BIND_ON_OPEN was used. The values are those of an arbitrary one of the possible cluster queues if either MQOO_BIND_NOT_FIXED was used or

MQOO_BIND_AS_Q_DEF was used and the *DefBind* queue attribute was MQBND_BIND_NOT_FIXED. See the *MQSeries Queue Manager Clusters* book for more information.

Note: The values returned by the call are a snapshot of the selected attributes. The attributes can change before your program acts on the returned values.

There is a description of the MQINQ call in the *MQSeries Application Programming Reference* manual.

Some cases where the MQINQ call fails

If you open an alias to inquire about its attributes, you are returned the attributes of the alias queue (the MQSeries object used to access another queue), not those of the base queue. However, the definition of the base queue to which the alias resolves is also opened by the queue manager, and if another program changes the usage of the base queue in the interval between your MQOPEN and MQINQ calls, your MQINQ call fails and returns the MQRC_OBJECT_CHANGED reason code. The call also fails if the attributes of the alias queue object are changed.

Similarly, when you open a remote queue to inquire about its attributes, you are returned the attributes of the local definition of the remote queue only.

If you specify one or more selectors that are not valid for the type of queue about whose attributes you are inquiring, the MQINQ call completes with a warning and sets the output as follows:

- For integer attributes, the corresponding elements of *IntAttrs* are set to MQIAV_NOT_APPLICABLE.
- For character attributes, the corresponding portions of the *CharAttrs* string are set to asterisks.

If you specify one or more selectors that are not valid for the type of object about whose attributes you are inquiring, the MQINQ call fails and returns the MQRC_SELECTOR_ERROR reason code.

It is not possible to call MQINQ to look at a model queue. You will have to use either the MQSC facility or use the commands available on your platform.

Setting queue attributes

You can set only the following queue attributes using the MQSET call:

- *InhibitGet* (but not for remote queues)
- *DistList*
- *InhibitPut*
- *TriggerControl*
- *TriggerType*
- *TriggerDepth*
- *TriggerMsgPriority*
- *TriggerData*

The MQSET call has the same parameters as the MQINQ call. However for MQSET, all parameters except the completion code and reason code are input parameters. There are no partial-completion situations.

Using MQSET

Note: You cannot use the MQI to set the attributes of MQSeries objects other than locally-defined queues.

There is a description of the MQSET call in the *MQSeries Application Programming Reference* manual.

Chapter 13. Committing and backing out units of work

This chapter describes how to commit and back out any recoverable get and put operations that have occurred in a unit of work. The following terms, described below, are used in this topic:

- Commit
- Back out
- Syncpoint coordination
- Syncpoint
- Unit of work
- Single-phase commit
- Two-phase commit

If you are familiar with these transaction processing terms, you can skip to “Syncpoint considerations in MQSeries applications” on page 178.

Commit and back out

When a program puts a message on a queue within a unit of work, that message is made visible to other programs only when the program *commits* the unit of work. To commit a unit of work, *all* updates must be successful to preserve data integrity. If the program detects an error and decides that the put operation should not be made permanent, it can *back out* the unit of work. When a program performs a back out, MQSeries restores the queue by removing the messages that were put on the queue by that unit of work. The way in which the program performs the commit and back out operations depends on the environment in which the program is running.

Similarly, when a program gets a message from a queue within a unit of work, that message remains on the queue until the program commits the unit of work, but the message is not available to be retrieved by other programs. The message is permanently deleted from the queue when the program commits the unit of work. If the program backs out the unit of work, MQSeries restores the queue by making the messages available to be retrieved by other programs.

Changes to queue attributes (either by the MQSET call or by commands) are not affected by the committing or backing out of units of work.

Syncpoint coordination, syncpoint, unit of work

Syncpoint coordination is the process by which units of work are either committed or backed out with data integrity.

The decision to commit or back out the changes is taken, in the simplest case, at the end of a transaction. However, it can be more useful for an application to synchronize data changes at other logical points within a transaction. These logical points are called *syncpoints* (or *synchronization points*) and the period of processing a set of updates between two syncpoints is called a *unit of work*. Several MQGET calls and MQPUT calls can be part of a single unit of work. The maximum number of messages within a unit of work can be controlled by the DEFINE MAXSMSGS command on OS/390, or by the MAXUMSGS attribute of the ALTER QMGR command on other platforms. See the *MQSeries Command Reference* book for details of these commands.

Commit and back out

Single-phase commit

A *single-phase commit* process is one in which a program can commit updates to a queue without coordinating its changes with other resource managers.

Two-phase commit

A *two-phase commit* process is one in which updates that a program has made to MQSeries queues can be coordinated with updates to other resources (for example, databases under the control of DB2). Under such a process, updates to *all* resources are committed or backed out together.

To help handle units of work, MQSeries provides the *BackoutCount* attribute. This is incremented each time a message, within a unit of work, is backed out. If the message repeatedly causes the unit of work to abend, the value of the *BackoutCount* finally exceeds that of the *BackoutThreshold*. This value is set when the queue is defined. In this situation, the application can choose to remove the message from the unit of work and put it onto another queue, as defined in *BackoutRequeueQName*. When the message is moved, the unit of work can commit.

This chapter introduces committing and backing out units of work, under these headings:

- “Syncpoint considerations in MQSeries applications”
- “Syncpoints in MQSeries for OS/390 applications” on page 179
- “Syncpoints in CICS for AS/400 applications” on page 182
- “Syncpoints in MQSeries for OS/2 Warp, MQSeries for Windows NT, MQSeries for AS/400, and MQSeries on UNIX systems” on page 182
- “Syncpoints in MQSeries for Tandem NonStop Kernel applications” on page 188
- “Interfaces to the AS/400 external syncpoint manager” on page 187
- “General XA support” on page 190

Syncpoint considerations in MQSeries applications

Two-phase commit is supported under:

- MQSeries for AIX
- MQSeries for AS/400
- MQSeries for HP-UX
- MQSeries for OS/2 Warp
- MQSeries for Sun Solaris
- Compaq Tru64 UNIX
- MQSeries for Tandem NonStop Kernel
- MQSeries for Windows NT
- CICS for MVS/ESA 4.1
- CICS Transaction Server for OS/390
- CICS on Open Systems
- TXSeries for Windows NT
- IMS/ESA
- OS/390 batch with RRS
- Other external coordinators using the X/Open XA interface

Single-phase commit is supported under:

- MQSeries for AS/400
- MQSeries for Compaq (DIGITAL) OpenVMS
- MQSeries for OS/2 Warp
- MQSeries for Tandem NonStop Kernel
- MQSeries on UNIX systems
- MQSeries for VSE/ESA

- MQSeries for Windows
- MQSeries for Windows NT
- CICS for OS/2
- CICS for Windows NT V2.0
- OS/390 batch

Note: For further details on external interfaces see “Interfaces to external syncpoint managers” on page 185, and the XA documentation *CAE Specification Distributed Transaction Processing: The XA Specification*, published by The Open Group. Transaction managers (such as CICS, IMS, Encina, and Tuxedo) can participate in two-phase commit, coordinated with other recoverable resources. This means that the queuing functions provided by MQSeries can be brought within the scope of a unit of work, managed by the transaction manager.

Samples shipped with MQSeries show MQSeries coordinating XA-compliant databases. For further information about these samples, see “Chapter 34. Sample programs (all platforms except OS/390)” on page 359.

In your MQSeries application, you can specify on every put and get call whether you want the call to be under syncpoint control. To make a put operation operate under syncpoint control, use the MQPMO_SYNCPOINT value in the *Options* field of the MQPMO structure when you call MQPUT. For a get operation, use the MQGMO_SYNCPOINT value in the *Options* field of the MQGMO structure. If you do not explicitly choose an option, the default action depends on the platform. The syncpoint control default on OS/390 and Tandem NSK is ‘yes’; for all other platforms, it is ‘no’.

Except on OS/390 batch with RRS, if a program issues the MQDISC call while there are uncommitted requests, an implicit syncpoint occurs. If the program ends abnormally, an implicit backout occurs. On OS/390, an implicit syncpoint occurs if the program ends normally without first calling MQDISC.

For MQSeries for OS/390 programs, you can use the MQGMO_MARK_SKIP_BACKOUT option to specify that a message should not be backed out if backout occurs (in order to avoid an ‘MQGET-error-backout’ loop). For information about using this option, see “Skipping backout” on page 144.

For information on committing and backing out units of work in MQSeries for VSE/ESA, see the *MQSeries for VSE/ESA V2R1 System Management Guide*.

Syncpoints in MQSeries for OS/390 applications

This section explains how to use syncpoints in transaction manager (CICS and IMS) and batch applications.

Syncpoints in CICS Transaction Server for OS/390 and CICS for MVS/ESA applications

In a CICS application you establish a syncpoint by using the EXEC CICS SYNCPOINT command. To back out all changes to the previous syncpoint, you can use the EXEC CICS SYNCPOINT ROLLBACK command. For more information, see the *CICS Application Programming Reference* manual.

Syncpointing and CICS

If other recoverable resources are also involved in the unit of work, the queue manager (in conjunction with the CICS syncpoint manager) participates in a two-phase commit protocol; otherwise, the queue manager performs a single-phase commit process.

If a CICS application issues the MQDISC call, no implicit syncpoint is taken. If the application closes down normally, any open queues are closed and an implicit commit occurs. If the application closes down abnormally, any open queues are closed and an implicit backout occurs.

Syncpoints in IMS applications

In an IMS application, you establish a syncpoint by using IMS calls such as GU (get unique) to the IOPCB and CHKP (checkpoint). To back out all changes since the previous checkpoint, you can use the IMS ROLB (rollback) call. For more information, see the following books:

- *IMS/ESA Version 4 Application Programming: DL/I Calls*
- *IMS/ESA Version 4 Application Programming: Design Guide*
- *IMS/ESA® Version 5 Application Programming: Database Manager*
- *IMS/ESA Version 5 Application Programming: Design Guide*

The queue manager (in conjunction with the IMS syncpoint manager) participates in a two-phase commit protocol if other recoverable resources are also involved in the unit of work.

All open handles are closed by the IMS adapter at a syncpoint (except in a nonmessage batch-oriented BMP). This is because a different user could initiate the next unit of work and MQSeries security checking is performed when the MQCONN, MQCONNX, and MQOPEN calls are made, not when the MQPUT or MQGET calls are made. The handles are closed at the beginning of the MQI call following the IMS call which initiated the syncpoint.

If you have not installed IMS APAR PN83757, handles are also closed after a ROLB call unless you are running IMS Version 3 or are running a nonmessage BMP.

If an IMS application (either a BMP or an MPP) issues the MQDISC call, open queues are closed but no implicit syncpoint is taken. If the application closes down normally, any open queues are closed and an implicit commit occurs. If the application closes down abnormally, any open queues are closed and an implicit backout occurs.

Syncpoints in OS/390 batch applications

For batch applications, you can use the MQSeries syncpoint management calls: MQCMIT and MQBACK. For backward compatibility, CSQBCMT and CSQBBAK are available as synonyms.

Note: If you need to commit or back out updates to resources managed by different resource managers, such as MQSeries and DB2, within a single unit of work you could use RRS. For further information see “Transaction management and recoverable resource manager services” on page 181.

Committing changes using the MQCMIT call

As input, you must supply the connection handle (*hconn*), which is returned by the MQCONN or MQCONNX call.

Syncpointing and batch

The output from MQCMIT is a completion code and a reason code. The call completes with a warning if the syncpoint was completed but the queue manager backed out the put and get operations since the previous syncpoint.

Successful completion of the MQCMIT call indicates to the queue manager that the application has reached a syncpoint and that all put and get operations made since the previous syncpoint have been made permanent.

There is a description of the MQCMIT call in the *MQSeries Application Programming Reference* manual.

Backing out changes using the MQBACK call

As input, you must supply a connection handle (*Hconn*). Use the handle that is returned by the MQCONN or MQCONNX call.

The output from MQBACK is a completion code and a reason code.

The output indicates to the queue manager that the application has reached a syncpoint and that all gets and puts that have been made since the last syncpoint have been backed out.

There is a description of the MQBACK call in the *MQSeries Application Programming Reference* manual.

Transaction management and recoverable resource manager services

Transaction management and recoverable resource manager services (RRS) is an OS/390 facility to provide two-phase syncpoint support across participating resource managers. An application can update recoverable resources managed by various OS/390 resource managers such as MQSeries and DB2, and then commit or back out these updates as a single unit of work. RRS provides the necessary unit-of-work status logging during normal execution, coordinates the syncpoint processing, and provides appropriate unit-of-work status information during subsystem restart.

MQSeries for OS/390 RRS participant support enables MQSeries applications in the batch, TSO, and DB2 stored procedure environments to update both MQSeries and non-MQSeries resources (for example, DB2) within a single logical unit of work. For information about RRS participant support, see the *MVS Programming: Resource Recovery* manual.

Your MQSeries application can use either MQCMIT and MQBACK or the equivalent RRS calls, SRRCMIT and SRRBACK. See “RRS batch adapter” on page 215 for more information.

RRS availability: If RRS is not active on your OS/390 system, any MQSeries call issued from a program linked with either RRS stub (CSQBRSTB or CSQBRRSI) returns MQRC_ENVIRONMENT_ERROR.

DB2 stored procedures: If you use DB2 stored procedures with RRS you must be aware of the following guidelines:

- DB2 stored procedures that use RRS must be WLM-managed.
- If a DB2-managed stored procedure contains MQSeries calls, and it is linked with either RRS stub (CSQBRSTB or CSQBRRSI), the MQCONN or MQCONNX call returns MQRC_ENVIRONMENT_ERROR.

DB2 stored procedures

- If a WLM-managed stored procedure contains MQSeries calls, and is linked with a non-RRS stub, the MQCONN or MQCONNX call returns MQRC_ENVIRONMENT_ERROR, unless it is the first MQSeries call executed since the stored procedure address space started.
- If your DB2 stored procedure contains MQSeries calls and is linked with a non-RRS stub, MQSeries resources updated in that stored procedure are not committed until the stored procedure address space ends, or until a subsequent stored procedure does an MQCMIT (using an MQSeries Batch/TSO stub).
- Multiple copies of the same stored procedure can execute concurrently in the same address space. You should ensure that your program is coded in a re-entrant manner if you want DB2 to use a single copy of your stored procedure. Otherwise you may receive MQRC_HCONN_ERROR on any MQSeries call in your program.
- You must not code MQCMIT or MQBACK in a WLM-managed DB2 stored procedure.
- All programs must be designed to run in Language Environment® (LE).

Syncpoints in CICS for AS/400 applications

MQSeries for AS/400 participates in CICS for AS/400 units of work. You can use the MQI within a CICS for AS/400 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 for AS/400 application, you cannot log off CICS for AS/400 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 for AS/400.

Syncpoints in MQSeries for OS/2 Warp, MQSeries for Windows NT, MQSeries for AS/400, and MQSeries on UNIX systems

Syncpoint support operates on two types of units of work: local and global.

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

A *global* unit of work is one in which resources belonging to other resource managers, such as databases, are also updated. MQSeries can coordinate such units of work itself. They can also be coordinated by an external commitment controller such as another transaction manager or the OS/400 commitment controller.

For full integrity, a two-phase commit procedure must be used. Two-phase commit can be provided by XA-compliant transaction managers and databases such as IBM's TXSeries and UDB and also by the OS/400 V4R4 commitment controller. MQSeries Version 5 products (except MQSeries for AS/400 and MQSeries for OS/390) can coordinate global units of work using a two-phase commit process.

MQSeries for AS/400 cannot coordinate a global unit of work but can participate in one being controlled by the OS/400 commitment controller.

Local units of work

Units of work that involve only the queue manager are called *local* units of work. Syncpoint coordination is provided by the queue manager itself (internal coordination) using a single-phase commit process.

To start a local unit of work, the application issues MQGET, MQPUT, or MQPUT1 requests specifying the appropriate syncpoint option. The unit of work is committed using MQCMIT or rolled back using MQBACK. However, the unit of work also ends when the connection between the application and the queue manager is broken, whether intentionally or unintentionally.

If an application disconnects (MQDISC) from a queue manager while a unit of work is still active, the unit of work is committed. If, however, the application terminates without disconnecting, the unit of work is rolled back as the application is deemed to have terminated abnormally.

Global units of work

Use global units of work when you also need to include updates to resources belonging to other resource managers. Here the coordination may be internal or external to the queue manager:

Internal syncpoint coordination

Queue manager coordination of global units of work is supported only on MQSeries Version 5 products except for MQSeries for AS/400 and MQSeries for OS/390. It is not supported in an MQSeries client environment.

Here, the coordination is performed by MQSeries. To start a global unit of work, the application issues the MQBEGIN call.

As input to the MQBEGIN call, you must supply the connection handle (*Hconn*), which is returned by the MQCONN or MQCONNX call. This handle represents the connection to the MQSeries queue manager.

Again, the application issues MQGET, MQPUT, or MQPUT1 requests specifying the appropriate syncpoint option. This means that MQBEGIN can be used to initiate a global unit of work that updates local resources, resources belonging to other resource managers, or both. Updates made to resources belonging to other resource managers are made using the API of that resource manager. However, it is not possible to use the MQI to update queues that belong to other queue managers. MQCMIT or MQBACK must be issued before starting further units of work (local or global).

The global unit of work is committed using MQCMIT; this initiates a two-phase commit of all the resource managers involved in the unit of work. A two-phase commit process is used whereby resource managers (for example, XA-compliant database managers such as DB2, Oracle, and Sybase) are firstly all asked to prepare to commit. Only if all are prepared are they asked to commit. If any resource manager signals that it cannot commit, each is asked to back out instead. Alternatively, MQBACK can be used to roll back the updates of all the resource managers.

Syncpointing, other platforms

If an application disconnects (MQDISC) while a global unit of work is still active, the unit of work is committed. If, however, the application terminates without disconnecting, the unit of work is rolled back as the application is deemed to have terminated abnormally.

The output from MQBEGIN is a completion code and a reason code.

When MQBEGIN is used to start a global unit of work, all the external resource managers that have been configured with the queue manager are included. However, the call starts a unit of work but completes with a warning if:

- There are no participating resource managers (that is, no resource managers have been configured with the queue manager)

or

- One or more resource managers are not available.

In these cases, the unit of work should include updates to only those resource managers that were available when the unit of work was started.

If one of the resource managers is unable to commit its updates, all of the resource managers are instructed to roll back their updates, and MQCMIT completes with a warning. In unusual circumstances (typically, operator intervention), an MQCMIT call may fail if some resource managers commit their updates but others roll them back; the work is deemed to have completed with a 'mixed' outcome. Such occurrences are diagnosed in the error log of the queue manager so remedial action may be taken.

An MQCMIT of a global unit of work succeeds if all of the resource managers involved commit their updates.

For a description of the MQBEGIN call, see the *MQSeries Application Programming Reference* manual.

External syncpoint coordination

This occurs when a syncpoint coordinator other than MQSeries has been selected; for example, CICS, Encina, or Tuxedo. In this situation, MQSeries for OS/2 Warp, MQSeries on UNIX systems, and MQSeries for Windows NT register their interest in the outcome of the unit of work with the syncpoint coordinator so that they can commit or roll back any uncommitted get or put operations as required. The external syncpoint coordinator determines whether one- or two-phase commitment protocols are provided.

When an external coordinator is used MQCMIT, MQBACK, and MQBEGIN may not be issued. Calls to these functions fail with the reason code MQRC_ENVIRONMENT_ERROR.

The way in which an externally coordinated unit of work is started is dependent on the programming interface provided by the syncpoint coordinator. An explicit call may, or may not, be required. If an explicit call is required, and you issue an MQPUT call specifying the MQPMO_SYNCPOINT option when a unit of work is not started, the completion code MQRC_SYNCPOINT_NOT_AVAILABLE is returned.

The scope of the unit of work is determined by the syncpoint coordinator. The state of the connection between the application and the queue manager affects the success or failure of MQI calls that an application issues, not the state of the unit

of work. It is, for example, 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 known as a pending disconnect.

Interfaces to external syncpoint managers

MQSeries for OS/2 Warp, MQSeries on UNIX systems, and MQSeries for Windows NT support coordination of transactions by external syncpoint managers which utilize the X/Open XA interface. This support is available only on server configurations. The interface is not available to client applications.

Some XA transaction managers (not CICS on Open Systems or Encina) require that each XA resource manager supplies its name. This is the string called name in the XA switch structure. The resource manager for MQSeries on UNIX systems is named "MQSeries_XA_RMI". For further details on XA interfaces refer to XA documentation *CAE Specification Distributed Transaction Processing: The XA Specification*, published by The Open Group.

In an XA configuration, MQSeries on UNIX systems, MQSeries for OS/2 Warp, and MQSeries for Windows NT fulfil the role of an XA Resource Manager. An XA syncpoint coordinator can manage a set of XA Resource Managers, and synchronize the commit or backout of transactions in both Resource Managers. This is how it works for a statically-registered resource manager:

1. An application notifies the syncpoint coordinator that it wishes to start a transaction.
2. The syncpoint coordinator issues a call to any resource managers that it knows of, to notify them of the current transaction.
3. The application issues calls to update the resources managed by the resource managers associated with the current transaction.
4. The application requests that the syncpoint coordinator either commit or roll back the transaction.
5. The syncpoint coordinator issues calls to each resource manager using two-phase commit protocols to complete the transaction as requested.

The XA specification requires each Resource Manager to provide a structure called an *XA Switch*. This structure declares the capabilities of the Resource Manager, and the functions that are to be called by the syncpoint coordinator.

There are two versions of this structure:

MQRMIXASwitch

Static XA resource management

MQRMIXASwitchDynamic

Dynamic XA resource management

The structure is found in the following libraries:

mqmxa.lib

OS/2, and Windows NT XA library for Static resource management

mqmenc.lib

AIX, HP-UX, Sun Solaris, and Windows NT Encina XA library for Dynamic resource management

Syncpointing, other platforms

libmqmxa.a

UNIX systems XA library (non-threaded) for both Static and Dynamic resource management

libmqmxa_r.a

UNIX systems (except Compaq Tru64 UNIX V5.0, and Sun Solaris) XA library (threaded) for both Static and Dynamic resource management

The method that must be used to link them to an XA syncpoint coordinator is defined by the coordinator, and you will need to consult the documentation provided by that coordinator to determine how to enable MQSeries to cooperate with your XA syncpoint coordinator.

The *xa_info* structure that is passed on any *xa_open* call by the syncpoint coordinator should be the name of the queue manager that is to be administered. This takes the same form as the queue manager name passed to MQCONN or MQCONNX, and may be blank if the default queue manager is to be used.

Restrictions

- On OS/2, all functions declared in the XA switch are declared as `_System` functions.
- On Windows NT, all functions declared in the XA switch are declared as `_cdecl` functions.
- Only one queue manager may be administered by an external syncpoint coordinator at a time. This is due to the fact that the coordinator has an effective connection to each queue manager, and is therefore subject to the rule that only one connection is allowed at a time.
- All applications that are run using the syncpoint coordinator can connect only to the queue manager that is administered by the coordinator because they are already effectively connected to that queue manager. They must issue MQCONN or MQCONNX to obtain a connection handle and must issue MQDISC before they exit. Alternatively, they can use the CICS user exit 15 for CICS for OS/2 V2 and V3, and CICS for Windows NT V2, or the exit UE014015 for TXSeries for Windows NT V4 and CICS on Open Systems.

The features not implemented are:

- Association migration
- Asynchronous calls

Because CICS Transaction Server V4 is 32-bit, changes are required to the source of CICS user exits. The supplied samples have been updated to work with CICS Transaction Server V4 as shown in Table 8.

Table 8. Linking MQSeries for OS/2 Warp with CICS Version 3 applications

User exit	CICS V2 source	CICS V2 dll	TS V4 source	TS V4 dll
exit 15	amqzsc52.c	faaexp15.dll	amqzsc53.c	faaex315.dll
exit 17	amqzsc72.c	faaexp17.dll	amqzsc73.c	faaex317.dll

For CICS Transaction Server V4, the supplied user exits faaex315.dll and faaex317.dll should be renamed to the standard names faaexp15.dll and faaexp17.dll.

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)
```

MQSeries for AS/400 is also able to perform local units of work containing only updates to MQSeries resources. The choice between local units of work and participation in global units of work coordinated by OS/400 is made in each application when the application calls MQPUT, MQPUT1, or MQGET, specifying MQPMO_SYNCPOINT or MQGMO_SYNCPOINT, or MQBEGIN. If commitment control is not active when the first such call is issued, MQSeries starts a local unit of work and all further units of work for this connection to MQSeries will also use local units of work, regardless of whether commitment control is subsequently started. To commit or back out a local unit of work, you use MQCMIT or MQBACK respectively in the same way as the other MQSeries V5 products. The OS/400 commit and rollback calls such as the CL command COMMIT have no effect on MQSeries local units of work.

If you wish to use MQSeries for AS/400 with native OS/400 commitment control as an external syncpoint coordinator, you must ensure that any job with commitment control is active and that you are using MQSeries in a single-threaded job. If you call MQPUT, MQPUT1 or MQGET, specifying MQPMO_SYNCPOINT or MQGMO_SYNCPOINT, in a multi-threaded job in which commitment control has been started, the call will fail with a reason code of MQRC_SYNCPOINT_NOT_AVAILABLE.

It is possible to use local units of work and the MQCMIT and MQBACK calls in a multi-threaded 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

MQSeries for AS/400 syncpointing

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

When you are using OS/400 commitment control as an external syncpoint coordinator with MQSeries for AS/400, OS/400 performs a two-phase commit protocol in which MQSeries participates. Because each unit of work is committed in two phases, there is the possibility that the queue manager becomes unavailable for the second phase after having voted to commit in the first phase. This can happen, for example, if the queue manager's internal jobs are ended. In this situation, the job log performing the commit will contain message CPF835F indicating that a commit or rollback operation failed. The messages preceding this will indicate the cause of the problem, whether it occurred during a commit or rollback operation, and also the Logical Unit of Work ID (LUWID) for the failed unit of work.

If the problem was caused by the failure of the MQSeries API commitment resource during the commit or rollback of a prepared unit of work, you can use the WRKMQMTRN command to complete the operation and restore the integrity of the transaction. The command requires that you know the LUWID of the unit of work to commit and back out.

Syncpoints in MQSeries for Tandem NonStop Kernel applications

When using MQSeries for Tandem NonStop Kernel, transaction management is performed under the control of the Tandem TM/MP product, rather than by MQSeries itself.

The effects of this difference are:

- The default SYNCPOINT option for the MQPUT and MQGET calls is SYNCPOINT, rather than NO_SYNCPOINT.
- To use the default (SYNCPOINT) option for MQPUT, MQGET, and MQPUT1 operations, the application must have an active TM/MP Transaction that defines the unit of work to be committed. An application initiates a TM/MP transaction

MQSeries for Tandem NonStop Kernel syncpointing

by calling the `BEGINTRANSACTION()` function. All `MQPUT`, `MQPUT1`, and `MQGET` operations performed by the application while this transaction is active are within the same unit of work (transaction). Any other database operations performed by the application are also within this UOW. Note that there are system-imposed limits on the number and size of messages that can be written and deleted within a single TM/MP transaction. When the application has completed the UOW, the TM/MP transaction is ended (the UOW is committed) using the `ENDTRANSACTION()` function. If any error is encountered, the application can cancel the TM/MP transaction (backout the UOW) using the `ABORTTRANSACTION()` function. Consequently, the standard Version 2 functions `MQCMIT()` and `MQBACK()` are not supported on this product. If they are called, an error is returned.

- If an application uses the `NO_SYNCPOINT` option for `MQPUT`, `MQGET`, and `MQPUT1` operations, MQSeries starts a TM/MP transaction itself, performs the queuing operation, and commits the transaction before returning to the application. Each operation is therefore performed in its own UOW and, once complete, cannot be backed out by the application using TM/MP.
- A TM/MP transaction does not need to be active for MQI calls other than `MQGET`, `MQPUT`, and `MQPUT1`.
- Because TM/MP can cause previously performed `MQGET`, `MQPUT`, and `MQPUT1` operations to be backed out without notification, the current queue-depth and input-and-output-open counts of queues can become inaccurate. The MQSeries Status Server (MQSS) corrects such inaccuracies. `PEN` call corrects the value of these at configurable intervals. However, applications should be coded to be resilient to inaccuracies in these quantities, especially in an environment that may involve backed-out transactions.
- The back-out count attribute cannot be maintained in the same way as on standard Version 2 implementations. Also, the harden backout count attribute is not used.
- The `MQRC_SYNCPOINT_LIMIT_REACHED` reason code is used by MQSeries for Tandem NonStop Kernel V2.2.0.1 to inform an application that the system-imposed limit on the number of I/O operations within a single TM/MP transaction has been reached. If the application specified the `SYNCPOINT` option, it should cancel the transaction (back out the UOW) and retry with a smaller number of operations in that UOW.
- The `MQRC_UOW_CANCELED` reason code informs the application that the UOW (TM/MP transaction) has been canceled, either by the system itself (TM/MP imposes some system-wide resource-usage thresholds that will cause this), by user action, or by the initiator of the transaction itself.

General XA support

General XA support

General XA support is not supported on AS/400, Compaq (DIGITAL) OpenVMS, or Tandem NonStop Kernel.

An XA switch load module is provided to enable you to link CICS with MQSeries on UNIX systems. Additionally, sample source code files are provided to enable you to develop the XA switches for other transaction messages. The names of the switch load modules provided are:

Table 9. Essential Code for CICS applications

Description	C (source)	C (exec) - add one of the following to your XAD.Stanza
XA initialization routine	amqzscix.c	amqzsc - CICS for AIX Version 2.1, amqzsc - TXSeries for AIX, Version 4.2, amqzsc - TXSeries for HP-UX, Version 4.2, amqzsc - CICS for Siemens Nixdorf SINIX Version 2.2, amqzsc - TXSeries for Sun Solaris, Version 4.2
	amqzscin.c	mqmc4swi - TXSeries for Windows NT, Version 4.2

Chapter 14. Starting MQSeries applications using triggers

Triggering is not supported on MQSeries for Windows.

Some MQSeries applications that serve queues run continuously, so they are always available to retrieve messages that arrive on the queues. However, this may not be desirable when the number of messages arriving on the queues is unpredictable. In this case, applications could be consuming system resources even when there are no messages to retrieve.

MQSeries provides a facility that enables an application to be started automatically when there are messages available to retrieve. This facility is known as *triggering*.

For information about triggering channels see the *MQSeries Intercommunication* book.

This chapter introduces triggering, under these headings:

- “What is triggering?”
- “Prerequisites for triggering” on page 196
- “Conditions for a trigger event” on page 198
- “Controlling trigger events” on page 202
- “Designing an application that uses triggered queues” on page 204
- “Trigger monitors” on page 206
- “Properties of trigger messages” on page 209
- “When triggering does not work” on page 210

What is triggering?

The queue manager defines certain conditions as constituting “trigger events”. If triggering is enabled for a queue and a trigger event occurs, the queue manager sends a *trigger message* to a queue called an *initiation queue*. The presence of the trigger message on the initiation queue indicates that a trigger event has occurred.

Trigger messages generated by the queue manager are not persistent. This has the effect of reducing logging (thereby improving performance), and minimizing duplicates during restart, so improving restart time.

The program which processes the initiation queue is called a *trigger-monitor application*, and its function is to read the trigger message and take appropriate action, based on the information contained in the trigger message. Normally this action would be to start some other application to process the queue which caused the trigger message to be generated. From the point of view of the queue manager, there is nothing special about the trigger-monitor application—it is simply another application that reads messages from a queue (the initiation queue).

If triggering is enabled for a queue, you have the option to create a *process-definition object* associated with it. This object contains information about the application that processes the message which caused the trigger event. If the process definition object is created, the queue manager extracts this information and places it in the trigger message, for use by the trigger-monitor application. The name of the process definition associated with a queue is given by the *ProcessName* local-queue attribute. Each queue can specify a different process definition, or several queues can share the same process definition.

Triggering

On MQSeries Version 5 products except MQSeries for OS/390, in the case of triggering a channel, you do not need to create a process definition object; the transmission queue definition is used instead. When a trigger event occurs, the transmission queue definition contains information about the application that processes the message that caused the event. Again, when the queue manager generates the trigger message, it extracts this information and places it in the trigger message.

On MQSeries for VSE/ESA, a trigger event is defined to activate the MQSeries trigger API Handler, that is, the MQ02 CICS Transaction. The trigger API handler executes a CICS LINK to the application program or a CICS START to the application transaction depending on whether you defined a program name or a transaction name in the queue definition. For more information, see the *MQSeries for VSE/ESA V2R1 System Management Guide*.

Triggering is supported by MQSeries Clients in the Compaq (DIGITAL) OpenVMS, OS/2, UNIX systems, Windows systems 3.1, Windows systems 95, and Windows NT environments. An application running in a client environment is the same as one running in a full MQSeries environment, except that you link it with the client libraries. However the trigger monitor and the application to be started must both be in the same environment.

Triggering involves:

Application queue

An *application queue* is a local queue, which, when it has triggering set on and when the conditions are met, requires that trigger messages are written.

Process Definition

An application queue can have a *process definition object* associated with it that holds details of the application that will get messages from the application queue. (See the *MQSeries Application Programming Reference* manual for a list of attributes.)

On MQSeries Version 5 products except MQSeries for OS/390, the process definition object is optional in the case of triggering channels.

Transmission queue

Transmission queues are needed if trigger-starting channels.

For a transmission queue on AIX, HP-UX, OS/2 Warp, OS/400, Sun Solaris, Compaq Tru64 UNIX, or Windows NT, the *TriggerData* attribute of the transmission queue can specify the name of the channel to be started. This can replace the process definition for triggering channels, but is used only when a process definition is not created.

Trigger event

A *trigger event* is an event that causes a trigger message to be generated by the queue manager. This is usually a message arriving on an application queue, but it can also occur at other times (see “Conditions for a trigger event” on page 198). MQSeries has a range of options to allow you to control the conditions that cause a trigger event (see “Controlling trigger events” on page 202).

Trigger message

The queue manager creates a *trigger message* when it recognizes a trigger event (see “Conditions for a trigger event” on page 198). It copies into the trigger message information about the application to be started. This

information comes from the application queue and the process definition object associated with the application queue. Trigger messages have a fixed format (see “Format of trigger messages” on page 209).

Initiation queue

An *initiation queue* is a local queue on which the queue manager puts trigger messages. A queue manager can own more than one initiation queue, and each one is associated with one or more application queues. A shared queue, a local queue accessible by queue managers in a queue-sharing group, can be an initiation queue on MQSeries for OS/390.

Trigger monitor

A *trigger monitor* is a continuously-running program that serves one or more initiation queues. When a trigger message arrives on an initiation queue, the trigger monitor retrieves the message. The trigger monitor uses the information in the trigger message. It issues a command to start the application that is to retrieve the messages arriving on the application queue, passing it information contained in the trigger message header, which includes the name of the application queue.

Note: You must issue the runmqchi command to start the trigger monitor, except on MQSeries Version 5 products.

(For more information, see “Trigger monitors” on page 206.)

To understand how triggering works, consider Figure 14, which is an example of trigger type FIRST, (MQTT_FIRST).

Triggering

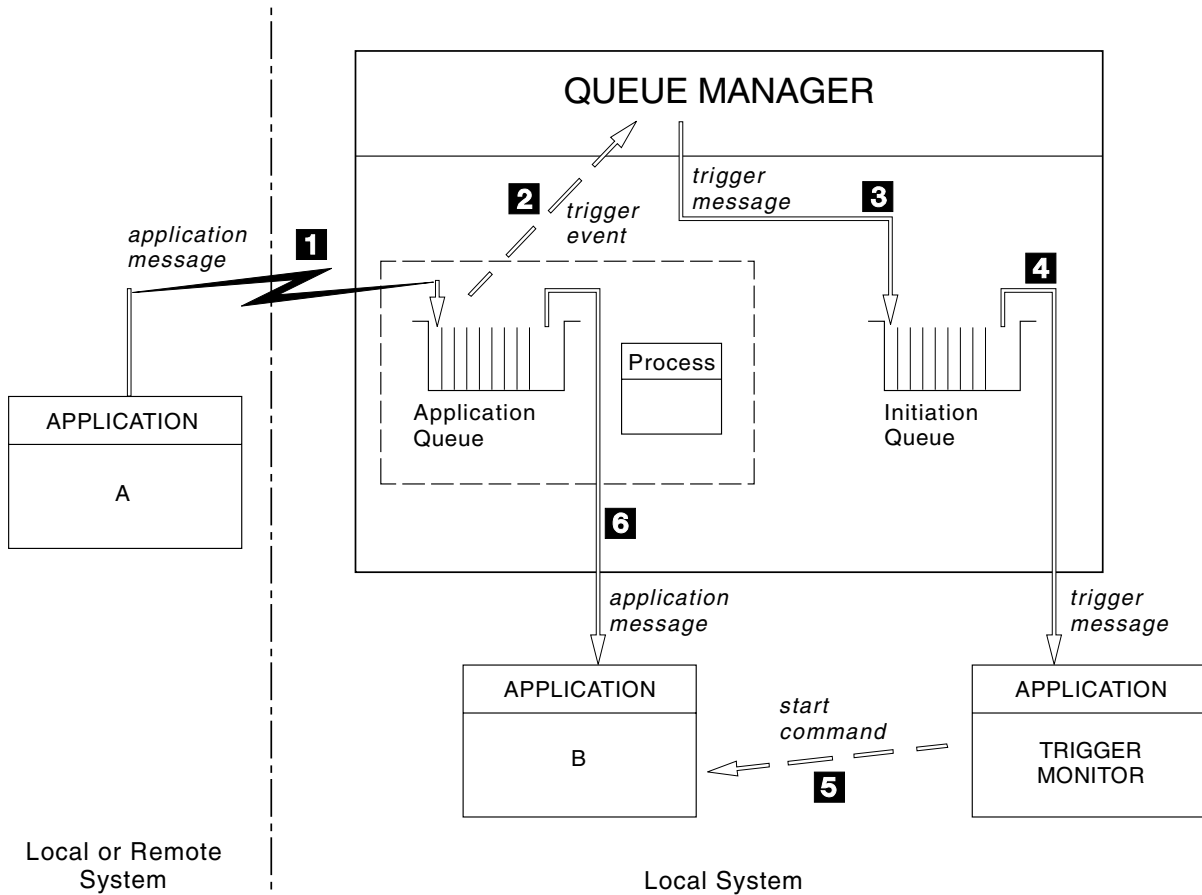


Figure 14. Flow of application and trigger messages

In Figure 14, the sequence of events is:

1. Application A, which can be either local or remote to the queue manager, puts a message on the application queue. Note that no application has this queue open for input. However, this fact is relevant only to trigger type FIRST and DEPTH.
2. The queue manager checks to see if the conditions are met under which it has to generate a trigger event. They are, and a trigger event is generated. Information held within the associated process definition object is used when creating the trigger message.
3. The queue manager creates a trigger message and puts it on the initiation queue associated with this application queue, but only if an application (trigger monitor) has the initiation queue open for input.
4. The trigger monitor retrieves the trigger message from the initiation queue.
5. The trigger monitor issues a command to start application B (the server application).
6. Application B opens the application queue and retrieves the message.

Notes:

1. If the application queue is open for input, by any program, and has triggering set for FIRST or DEPTH, no trigger event will occur since the queue is already being served.

2. If the initiation queue is not open for input, the queue manager will not generate any trigger messages, it will wait until an application opens the initiation queue for input.
3. When using triggering for channels, you are recommended to use trigger type FIRST or DEPTH.

So far, the relationship between the queues within triggering has been only on a one to one basis. Consider Figure 15.

An application queue has a process definition object associated with it that holds

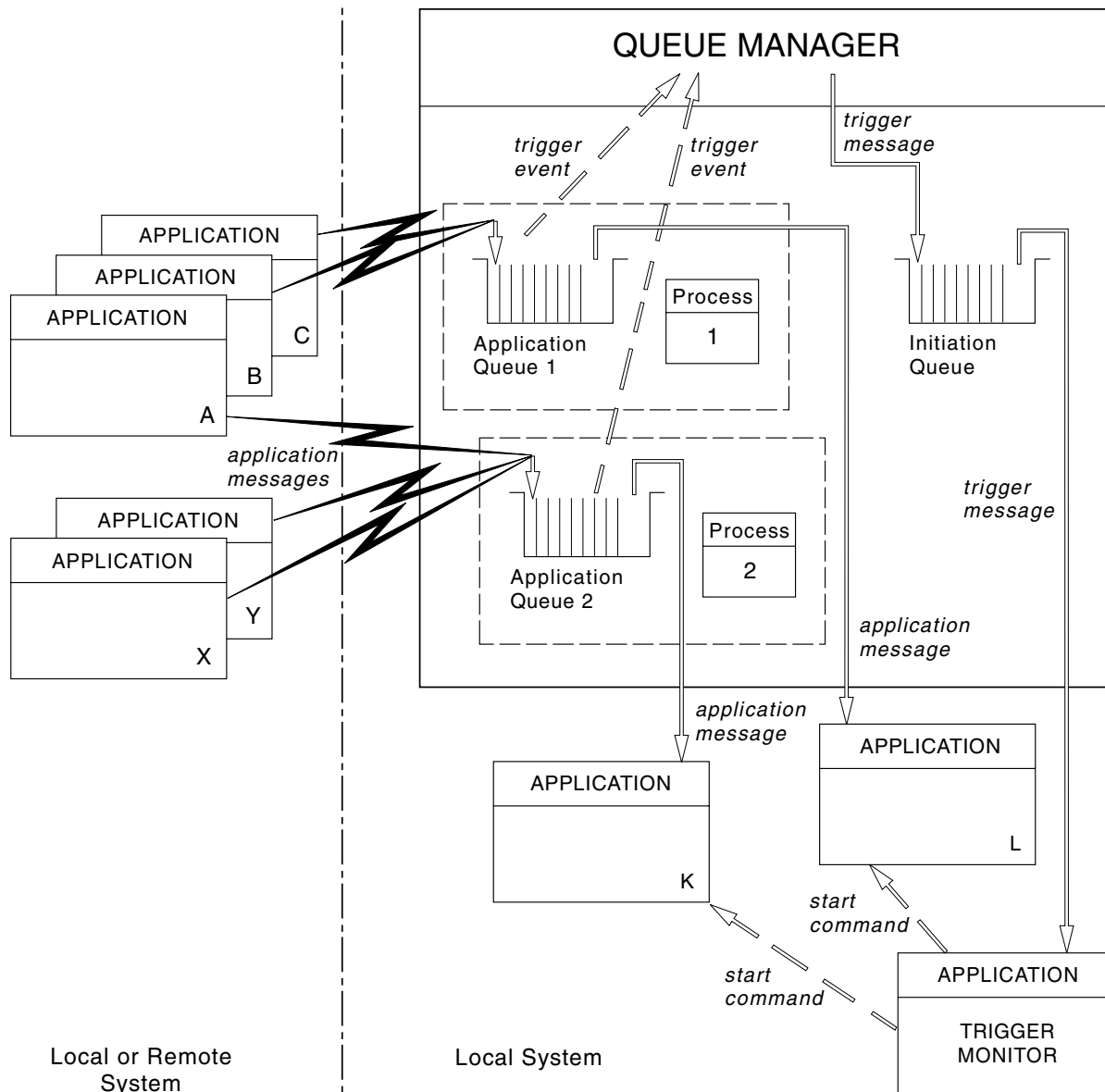


Figure 15. Relationship of queues within triggering

details of the application that will process the message. The queue manager places the information in the trigger message, so only one initiation queue is necessary. The trigger monitor extracts this information from the trigger message and starts the relevant application to deal with the message on each application queue.

Triggering

On MQSeries Version 5 products except MQSeries for OS/390, in the case of triggering a channel, the process definition object is optional. The transmission queue definition can determine the channel to be triggered.

Prerequisites for triggering

Before your application can take advantage of triggering, follow the steps below:

1. Either:

- a. Create an initiation queue for your application queue. For example:

```
DEFINE QLOCAL (initiation.queue) REPLACE      +
      LIKE (SYSTEM.DEFAULT.LOCAL.QUEUE)      +
      DESCR ('initiation queue description')
```

or

- b. Determine the name of a local queue that already exists and can be used by your application, and specify its name in the *InitiationQName* field of the application queue.

You can think of this task as associating the initiation queue with the application queue. A queue manager can own more than one initiation queue—you may want some of your application queues to be served by different programs, in which case you could use one initiation queue for each serving program, although you do not have to. Here is an example of how to create an application queue:

```
DEFINE QLOCAL (application.queue) REPLACE  +
      LIKE (SYSTEM.DEFAULT.LOCAL.QUEUE)    +
      DESCR ('appl queue description')      +
      INITQ ('initiation.queue')           +
      PROCESS ('process.name')             +
      TRIGGER                               +
      TRIGTYPE (FIRST)
```

2. If you are triggering an application, create a process definition object to contain information relating to the application that is to serve your application queue. For example, to trigger-start a CICS payroll transaction called PAYR:

```
|
| DEFINE PROCESS (process.name) +
| REPLACE +
| DESCR ('process description') +
| APPLTYPE ('CICS') +
| APPLICID ('PAYR') +
| USERDATA ('Payroll data')
```

Here is an extract from an MQSeries for AS/400 CL program that creates a process definition object:

```
/* Queue used by AMQSINQA */
CRTMQMQ QNAME('SYSTEM.SAMPLE.INQ') +
      QTYPE(*LCL) REPLACE(*YES) +
      MQMNAME +
      TEXT('queue for AMQSINQA') +
      SHARE(*YES) /* Shareable */+
      DFTMSGPST(*YES)/* Persistent messages OK */+
      TRGENBL(*YES) /* Trigger control on */+
      TRGTYPE(*FIRST)/* Trigger on first message*/+
      PRCNAME('SYSTEM.SAMPLE.INQPROCESS') +
      INITQNAME('SYSTEM.SAMPLE.TRIGGER')

/* Process definition */
CRTMQMPRC PRCNAME('SYSTEM.SAMPLE.INQPROCESS') +
      REPLACE(*YES) +
```


Triggering prerequisites

```

MQMNAME                                     +
TEXT('trigger process for AMQSINQA')       +
ENVDATA('JOBPTY(3)') /* Submit parameter */+
APPID('AMQSINQA') /* Program name */

```

When the queue manager creates a trigger message, it copies information from the attributes of the process definition object into the trigger message.

Platform	To create a process definition object
UNIX systems, Digital OpenVMS, OS/2, Windows NT	Use DEFINE PROCESS or use SYSTEM.DEFAULT.PROCESS and modify using ALTER PROCESS
OS/390	Use DEFINE PROCESS (see sample code in step 2), or use the operations and control panels.
OS/400	Use a CL program containing code as in step 2.

- If you are triggering a channel on a product other than an MQSeries Version 5 product or MQSeries for VSE/ESA or MQSeries for OS/390, you need to create a process definition. Otherwise, you can create a transmission queue definition and specify the *ProcessName* attribute as blanks.

The *TrigData* attribute can contain the name of the channel to be triggered or it can be left blank. If it is left blank, the channel initiator searches the channel definition files until it finds a channel that is associated with the named transmission queue. When the queue manager creates a trigger message, it copies information from the *TrigData* attribute of the transmission queue definition into the trigger message.

- If you have created a process definition object to specify properties of the application that is to serve your application queue, associate the process object with your application queue by naming it in the *ProcessName* attribute of the queue.

Platform	Use commands
UNIX systems, Digital OpenVMS, OS/2, Windows NT	ALTER QLOCAL
OS/390	ALTER QLOCAL
AS/400	CHGMQM

- Start instances of the trigger monitors (or trigger servers in MQSeries for AS/400) that are to serve the initiation queues you have defined. See “Trigger monitors” on page 206 for more information.

If you wish to be aware of any undelivered trigger messages, make sure your queue manager has a dead-letter (undelivered-message) queue defined. Specify the name of the queue in the *DeadLetterQName* queue manager field.

You can then set the trigger conditions you require, using the attributes of the queue object that defines your application queue. For more information on this, see “Controlling trigger events” on page 202.

Conditions for a trigger event

References to shared queues in this section mean shared queues in a queue-sharing group, only available on MQSeries for OS/390.

The queue manager creates a trigger message when the following conditions are satisfied:

1. A message is *put* on a queue.
2. The message has a priority greater than or equal to the threshold trigger priority of the queue. This priority is set in the *TriggerMsgPriority* local queue attribute—if it is set to zero, any message qualifies.
3. The number of messages on the queue with priority greater than or equal to *TriggerMsgPriority* was previously, depending on *TriggerType*:
 - Zero (for trigger type MQTT_FIRST)
 - Any number (for trigger type MQTT_EVERY)
 - *TriggerDepth* minus 1 (for trigger type MQTT_DEPTH)

Notes:

- a. For non-shared local queues, the queue manager counts both committed and uncommitted messages when it assesses whether the conditions for a trigger event exist. Consequently an application may be started when there are no messages for it to retrieve because the messages on the queue have not been committed. In this situation, you are strongly recommended to consider using the wait option with a suitable *WaitInterval*, so that the application waits for its messages to arrive.
 - b. For local shared queues the queue manager counts committed messages only.
4. For triggering of type FIRST or DEPTH, no program has the application queue open for removing messages (that is, the *OpenInputCount* local queue attribute is zero).

Notes:

- a. For shared queues, special conditions apply when multiple queue managers have trigger monitors running against a queue. In this situation, if one or more queue managers have the queue open for input shared, then the trigger criteria on the other queue managers will be treated as *TriggerType* MQTT_FIRST and *TriggerMsgPriority* zero. When all the queue managers close the queue for input, the trigger conditions revert to those specified in the queue definition.
 - b. For shared queues, this condition is applied on a per queue manager basis. That is, a queue manager's *OpenInputCount* for a queue must be zero for a trigger message to be generated for the queue by that queue manager. However, if any queue manager in the queue-sharing group has the queue open using the MQOO_INPUT_EXCLUSIVE option, no trigger message will be generated for that queue by any of the queue managers in the queue-sharing group.
5. On MQSeries for OS/390, if the application queue is one with a *Usage* attribute of MQUS_NORMAL, get requests for it are not inhibited (that is, the *InhibitGet* queue attribute is MQQA_GET_ALLOWED). Also, if the triggered application queue is one with a *Usage* attribute of MQUS_XMITQ, get requests for it are not inhibited.
 6. Either:
 - The *ProcessName* local queue attribute for the queue is not blank, and the process definition object identified by that attribute has been created.

or

- The *ProcessName* local queue attribute for the queue is all blank, but the queue is a transmission queue. As the process definition is optional, the *TriggerData* attribute may also contain the name of the channel to be started. This option is available only on MQSeries for AS/400, OS/2, HP-UX, AIX, Sun Solaris, Compaq Tru64 UNIX, and Windows NT. In this case, the trigger message contains attributes with the following values:

ProcessName: blanks
TriggerData: trigger data
ApplType: MQAT_UNKNOWN
ApplId: blanks
EnvData: blanks
UserData: blanks

7. An initiation queue has been created, and has been specified in the *InitiationQName* local queue attribute. Also:
 - Get requests are not inhibited for the initiation queue (that is, the *InhibitGet* queue attribute is MQQA_GET_ALLOWED).
 - Put requests must not be inhibited for the initiation queue (that is, the *InhibitPut* queue attribute must be MQQA_PUT_ALLOWED).
 - The *Usage* attribute of the initiation queue must be MQUS_NORMAL.
 - In environments where dynamic queues are supported, the initiation queue must not be a dynamic queue that has been marked as logically deleted.
8. A trigger monitor currently has the initiation queue open for removing messages (that is, the *OpenInputCount* local queue attribute is greater than zero).
9. The trigger control (*TriggerControl* local queue attribute) for the application queue is set to MQTC_ON. To do this, set the *trigger* attribute when you define your queue, or use the ALTER QLOCAL command.
10. The trigger type (*TriggerType* local queue attribute) is not MQTT_NONE.

If all of the above required conditions are met, and the message that caused the trigger condition is put as part of a unit of work, the trigger message does not become available for retrieval by the trigger monitor application until the unit of work completes, whether the unit of work is committed *or* backed out.

11. A suitable message is placed on the queue, for a *TriggerType* of MQTT_FIRST or MQTT_DEPTH, and the queue:
 - Was not previously empty (MQTT_FIRST)

or

- Had *TriggerDepth* or more messages (MQTT_DEPTH)

and conditions 2 through 10 (excluding 3) are satisfied, if in the case of MQTT_FIRST a sufficient interval (*TriggerInterval* queue-manager attribute) has elapsed since the last trigger message was written for this queue.

This is to allow for a queue server that ends before processing all of the messages on the queue. The purpose of the trigger interval is to reduce the number of duplicate trigger messages that are generated.

Note: If you stop and restart the queue manager, the *TriggerInterval* “timer” is reset. There is a small window during which it is possible to produce two trigger messages. The window exists when the queue’s trigger

Trigger event conditions

attribute is set to enabled at the same time as a message arrives and the queue was not previously empty (MQTT_FIRST) or had *TriggerDepth* or more messages (MQTT_DEPTH).

12. The only application serving a queue issues an MQCLOSE call, for a *TriggerType* of MQTT_FIRST or MQTT_DEPTH, and there is at least:
 - One (MQTT_FIRST)

or

- *TriggerDepth* (MQTT_DEPTH)

messages on the queue of sufficient priority (condition 2 on page 198), and conditions 6 through 10 are also satisfied.

This is to allow for a queue server that issues an MQGET call, finds the queue empty, and so ends; however, in the interval between the MQGET and the MQCLOSE calls, one or more messages arrive.

Notes:

- a. If the program serving the application queue does not want to retrieve all the messages, this can cause a closed loop. Each time the program closes the queue, the queue manager creates another trigger message which causes the trigger monitor to start the server program again.
 - b. If the program serving the application queue backs out its get request (or if the program abends) before it closes the queue, the same happens.
 - c. To prevent such a loop occurring, you could use the *BackoutCount* field of MQMD to detect messages that are repeatedly backed out. For more information, see “Messages that are backed out” on page 35.
13. The following conditions are satisfied using MQSET or a command:

a.

- *TriggerControl* is changed to MQTC_ON

or

- *TriggerControl* is already MQTC_ON and the value of either *TriggerType*, *TriggerMsgPriority*, or *TriggerDepth* (if relevant) is changed,

and there is at least:

- One (MQTT_FIRST or MQTT_EVERY)

or

- *TriggerDepth* (MQTT_DEPTH)

messages on the queue of sufficient priority (condition 2 on page 198), and conditions 4 through 10 (excluding 8) are also satisfied.

This is to allow for an application or operator changing the triggering criteria, when the conditions for a trigger to occur are already satisfied.

- b. The *InhibitPut* queue attribute of an initiation queue changes from MQQA_PUT_INHIBITED to MQQA_PUT_ALLOWED, and there is at least:

- One (MQTT_FIRST or MQTT_EVERY)

or

- *TriggerDepth* (MQTT_DEPTH)

Trigger event conditions

messages of sufficient priority (condition 2 on page 198) on any of the queues for which this is the initiation queue, and conditions 4 through 10 are also satisfied. (One trigger message is generated for each such queue satisfying the conditions.)

This is to allow for trigger messages not being generated because of the MQQA_PUT_INHIBITED condition on the initiation queue, but this condition now having been changed.

- c. The *InhibitGet* queue attribute of an application queue changes from MQQA_GET_INHIBITED to MQQA_GET_ALLOWED, and there is at least:
 - One (MQTT_FIRST or MQTT EVERY)

or

- *TriggerDepth* (MQTT_DEPTH)

messages of sufficient priority (condition 2 on page 198) on the queue, and conditions 4 through 10, excluding 5, are also satisfied.

This allows applications to be triggered only when they are able to retrieve messages from the application queue.

- d. A trigger-monitor application issues an MQOPEN call for input from an initiation queue, and there is at least:
 - One (MQTT_FIRST or MQTT EVERY)

or

- *TriggerDepth* (MQTT_DEPTH)

messages of sufficient priority (condition 2 on page 198) on any of the application queues for which this is the initiation queue, and conditions 4 through 10 (excluding 8) are also satisfied, and no other application has the initiation queue open for input (one trigger message is generated for each such queue satisfying the conditions).

This is to allow for messages arriving on queues while the trigger monitor is not running, and for the queue manager restarting and trigger messages (which are nonpersistent) being lost.

14. MSGDLVSQ is set correctly. If you set MSGDLVSQ=FIFO, messages are delivered to the queue in a First In First Out basis. The priority of the message is ignored and the default priority of the queue is assigned to the message. If *TriggerMsgPriority* is set to a higher value than the default priority of the queue, no messages are triggered. If *TriggerMsgPriority* is set equal to or lower than the default priority of the queue, triggering occurs for type FIRST, EVERY, and DEPTH. For information about these types, see the description of the *TriggerType* field under “Controlling trigger events” on page 202. If you set MSGDLVSQ=PRIORITY and the message priority is equal to or greater than the *TriggerMsgPriority* field, messages only *count* towards a trigger event. In this case, again triggering occurs for type FIRST, EVERY, and DEPTH. As an example, if you put 100 messages of lower priority than the *TriggerMsgPriority*, the effective queue depth for triggering purposes is still zero. If you then put another message on the queue, but this time the priority is greater than or equal to the *TriggerMsgPriority*, the effective queue depth increases from zero to one and the condition for *TriggerType* FIRST is satisfied.

Trigger event conditions

Notes:

1. From step 12 (where trigger messages are generated as a result of some event other than a message arriving on the application queue), the trigger message is not put as part of a unit of work. Also, if the *TriggerType* is `MQTT_EVERY`, and if there are one or more messages on the application queue, only one trigger message is generated.
2. If MQSeries segments a message during `MQPUT`, a trigger event will not be processed until all of the segments have been successfully placed on the queue. However, once message segments are on the queue, MQSeries treats them as individual messages for triggering purposes. For example, a single logical message split into three pieces will cause only one trigger event to be processed when it is first `MQPUT` and segmented. However, each of the three segments will cause their own trigger events to be processed as they are moved through the MQSeries network.

Controlling trigger events

You control trigger events using some of the attributes that define your application queue. You can enable and disable triggering, and you can select the number or priority of the messages that count toward a trigger event. There is a full description of these attributes in the *MQSeries Application Programming Reference* manual.

The relevant attributes are:

TriggerControl

Use this attribute to enable and disable triggering for an application queue.

TriggerMsgPriority

The minimum priority that a message must have for it to count toward a trigger event. If a message of priority less than *TriggerMsgPriority* arrives on the application queue, the queue manager ignores the message when it determines whether to create a trigger message. If *TriggerMsgPriority* is set to zero, all messages count toward a trigger event.

TriggerType

In addition to the trigger type `NONE` (which disables triggering just like setting the *TriggerControl* to `OFF`), you can use the following trigger types to set the sensitivity of a queue to trigger events:

EVERY

A trigger event occurs every time a message arrives on the application queue. Use this type of trigger if you want a serving program to process only one message, then end.

FIRST A trigger event occurs only when the number of messages on the application queue changes from zero to one. Use this type of trigger if you want a serving program to start when the first message arrives on a queue, continue until there are no more messages to process, then end. Also see “Special case of trigger type `FIRST`” on page 203.

DEPTH

A trigger event occurs only when the number of messages on the application queue reaches the value of the *TriggerDepth* attribute. A typical use of this type of triggering is for starting a program when all the replies to a set of requests are received.

Triggering by depth

With triggering by depth, the queue manager disables triggering (using the *TriggerControl* attribute) after it creates a trigger message. Your application must reenables triggering itself (by using the MQSET call) after this has happened.

The action of disabling triggering is not under syncpoint control, so triggering cannot be reenables simply by backing out a unit of work. If a program backs out a put request that caused a trigger event, or if the program abends, you must reenables triggering by using the MQSET call or the ALTER QLOCAL command.

TriggerDepth

The number of messages on a queue that causes a trigger event when using triggering by depth.

The conditions that must be satisfied for a queue manager to create a trigger message are described in “Conditions for a trigger event” on page 198.

Example of the use of trigger type EVERY

Consider an application that generates requests for motor insurance. The application might send request messages to a number of insurance companies, specifying the same reply-to queue each time. It could set a trigger of type EVERY on this reply-to queue so that each time a reply arrives, the reply could trigger an instance of the server to process the reply.

Example of the use of trigger type FIRST

Consider an organization with a number of branch offices that each transmit details of the day’s business to the head office. They all do this at the same time, at the end of the working day, and at the head office there is an application that processes the details from all the branch offices. The first message to arrive at the head office could cause a trigger event which starts this application. This application would continue processing until there are no more messages on its queue.

Example of the use of trigger type DEPTH

Consider a travel agency application that creates a single request to confirm a flight reservation, to confirm a reservation for a hotel room, to rent a car, and to order some travelers’ checks. The application could separate these items into four request messages, sending each to a separate destination. It could set a trigger of type DEPTH on its reply-to queue (with the depth set to the value 4), so that it is restarted only when all four replies have arrived.

If another message (possibly from a different request) arrives on the reply-to queue before the last of the four replies, the requesting application is triggered early. To avoid this, when DEPTH triggering is being used to collect multiple replies to a request, you should always use a new reply-to queue for each request.

Special case of trigger type FIRST

With trigger type FIRST, if there is already a message on the application queue when another message arrives, the queue manager does not usually create another trigger message. However, the application serving the queue might not actually

Controlling trigger events

open the queue (for example, the application might end, possibly because of a system problem). If an incorrect application name has been put into the process definition object, the application serving the queue will not pick up any of the messages. In these situations, if another message arrives on the application queue, there is no server running to process this message (and any other messages on the queue).

To deal with this, the queue manager creates another trigger message if another message arrives on the application queue, but only if a predefined time interval has elapsed since the queue manager created the last trigger message for that queue. This time interval is defined in the queue manager attribute *TriggerInterval*. Its default value is 999 999 999 milliseconds.

You should consider the following points when deciding on a value for the trigger interval to be used in your application:

- If *TriggerInterval* is set to a low value, trigger type FIRST might behave like trigger type EVERY (this depends on the rate that messages are being put onto the application queue, which in turn may depend on other system activity). This is because, if the trigger interval is very small, another trigger message is generated each time a message is put onto the application queue, even though the trigger type is FIRST, not EVERY. (Trigger type FIRST with a trigger interval of zero is equivalent to trigger type EVERY.)
- If a unit of work is backed out (see “Trigger messages and units of work”) and the trigger interval has been set to a high value (or the default value), one trigger message is generated when the unit of work is backed out. However, if you have set the trigger interval to a low value or to zero (causing trigger type FIRST to behave like trigger type EVERY) many trigger messages can be generated. If the unit of work is backed out, all the trigger messages are still made available. The number of trigger messages generated depends on the trigger interval, the maximum number being reached when trigger interval has been set to zero.

Designing an application that uses triggered queues

You have seen how to set up, and control, triggering for your applications. Here are some tips you should consider when you design your application.

Trigger messages and units of work

Trigger messages created because of trigger events that are not part of a unit of work are put on the initiation queue, outside any unit of work, with no dependence on any other messages, and are available for retrieval by the trigger monitor immediately.

Trigger messages created because of trigger events that *are* part of a unit of work are put on the initiation queue as part of the same unit of work. Trigger monitors cannot retrieve these trigger messages until the unit of work completes. This applies whether the unit of work is committed or backed out.

If the queue manager fails to put a trigger message on an initiation queue, it will be put on the dead-letter (undelivered-message) queue.

Notes:

1. The queue manager counts both committed and uncommitted messages when it assesses whether the conditions for a trigger event exist.

With triggering of type FIRST or DEPTH, trigger messages are made available even if the unit of work is backed out so that a trigger message is always available when the required conditions are met. For example, consider a put request within a unit of work for a queue that is triggered with trigger type FIRST. This causes the queue manager to create a trigger message. If another put request occurs, from another unit of work, this does not cause another trigger event because the number of messages on the application queue has now changed from one to two, which does not satisfy the conditions for a trigger event. Now if the first unit of work is backed out, but the second is committed, a trigger message is still created.

However, this does mean that trigger messages are sometimes created when the conditions for a trigger event are *not* satisfied. Applications that use triggering must always be prepared to handle this situation. It is recommended that you use the wait option with the MQGET call, setting the *WaitInterval* to a suitable value.

2. For local shared queues (that is, shared queues in a queue-sharing group) the queue manager counts committed messages only.

Getting messages from a triggered queue

When you design applications that use triggering, you must be aware that there may be a delay between a program being started by a trigger monitor, and other messages becoming available on the application queue. This can happen when the message that causes the trigger event is committed before the others.

To allow time for messages to arrive, always use the wait option when you use the MQGET call to remove messages from a queue for which trigger conditions are set. The *WaitInterval* should be sufficient to allow for the longest reasonable time between a message being put and that put call being committed. If the message is arriving from a remote queue manager, this time is affected by:

- The number of messages that are put before being committed
- The speed and availability of the communication link
- The sizes of the messages

For an example of a situation where you should use the MQGET call with the wait option, consider the same example we used when describing units of work. This was a put request within a unit of work for a queue that is triggered with trigger type FIRST. This event causes the queue manager to create a trigger message. If another put request occurs, from another unit of work, this does not cause another trigger event because the number of messages on the application queue has not changed from zero to one. Now if the first unit of work is backed out, but the second is committed, a trigger message is still created. So the trigger message is created at the time the first unit of work is backed out. If there is a significant delay before the second message is committed, the triggered application may need to wait for it.

With triggering of type DEPTH, a delay can occur even if all relevant messages are eventually committed. Suppose that the *TriggerDepth* queue attribute has the value 2. When two messages arrive on the queue, the second causes a trigger message to be created. However, if the second message is the first to be committed, it is at that time the trigger message becomes available. The trigger monitor starts the server program, but the program can retrieve only the second message until the first one is committed. So the program may need to wait for the first message to be made available.

Using triggered queues

You should design your application so that it terminates if no messages are available for retrieval when your wait interval expires. If one or more messages arrive subsequently, you should rely on your application being retriggered to process them. This method prevents applications being idle, and unnecessarily using resources.

Trigger monitors

To a queue manager, a trigger monitor is like any other application that serves a queue. However, a trigger monitor serves initiation queues.

A trigger monitor is usually a continuously-running program. When a trigger message arrives on an initiation queue, the trigger monitor retrieves that message. It uses information in the message to issue a command to start the application that is to process the messages on the application queue.

The trigger monitor must pass sufficient information to the program it is starting so that the program can perform the right actions on the right application queue.

A channel initiator is an example of a special type of trigger monitor for message channel agents. In this situation however, you must use either trigger type FIRST or DEPTH.

MQSeries for OS/390 trigger monitors

The following trigger monitor is provided for CICS Transaction Server for OS/390 and CICS for MVS/ESA:

CKTI You need to start one instance of CKTI for each initiation queue (see the *MQSeries for OS/390 System Administration Guide* for information on how to do this). CKTI passes the MQTM structure of the trigger message to the program it starts by EXEC CICS START TRANSID. The started program gets this information by using the EXEC CICS RETRIEVE command. A program can use the EXEC CICS RETRIEVE command with the RTRANSID option to determine how the program was started; if the value returned is CKTI, the program was started by MQSeries for OS/390. For an example of how to use CKTI, see the source code supplied for module CSQ4CVB2 in the Credit Check sample application supplied with MQSeries for OS/390. See “The Credit Check sample” on page 451 for a full description.

The following trigger monitor is provided for IMS/ESA:

CSQQTRMN

You need to start one instance of CSQQTRMN for each initiation queue (see the *MQSeries for OS/390 System Administration Guide* for information on how to do this). CSQQTRMN passes the MQTMC2 structure of the trigger message to the programs it starts.

MQSeries for OS/2 Warp, Digital OpenVMS, Tandem NSK, UNIX systems, AS/400, and Windows NT trigger monitors

The following trigger monitors are provided for the server environment:

amqstrg0

This is a sample trigger monitor that provides a subset of the function provided by **runmqtrm**. See “Chapter 34. Sample programs (all platforms except OS/390)” on page 359 for more information on amqstrg0.

runmqtrm

runmqtrm [-m *QMgrName*] [-q *InitQ*] is the command. The default is SYSTEM.DEFAULT.INITIATION.QUEUE on the default queue manager. It calls programs for the appropriate trigger messages. This trigger monitor supports the default application type.

The command string passed by the trigger monitor to the operating system is built as follows:

1. The *ApplId* from the relevant PROCESS definition (if created)
2. The MQTMC2 structure, enclosed in quotation marks
3. The *EnvData* from the relevant PROCESS definition (if created)

where *ApplId* is the name of the program to run - as it would be entered on the command line.

The parameter passed is the MQTMC2 character structure. A command string is invoked which has this string, exactly as provided, in 'quotation marks', in order that the system command will accept it as one parameter.

The trigger monitor will not look to see if there is another message on the initiation queue until the completion of the application it has just started. If the application has a lot of processing to do, this may mean that the trigger monitor cannot keep up with the number of trigger messages arriving. You have two options:

- Have more trigger monitors running
- Run the started applications in the background

If you choose to have more trigger monitors running you have control over the maximum number of applications that can run at any one time. If you choose to run applications in the background, there is no restriction imposed by MQSeries on the number of applications that can run.

To run the started application in the background under OS/2, or Windows NT, within the *ApplId* field you must prefix the name of your application with a START command. For example:

```
START AMQSECHA /B
```

To run the started application in the background on UNIX systems, you must put an '&' at the end of the *EnvData* of the PROCESS definition.

The following trigger monitors are provided for the MQSeries client:

runmqtmc

This is the same as **runmqtrm** except that it links with the MQSeries client libraries.

For CICS:

The amqltmc0 trigger monitor is provided for CICS. It works in the same way as the standard trigger monitor, **runmqtrm**, but you run it in a different way and it triggers CICS transactions.

It is supplied as a CICS program and you must define it with a 4-character transaction name. Enter the 4-character name to start the trigger monitor. It uses the default queue manager (as named in the qm.ini file or, on MQSeries for Windows NT, the registry), and the SYSTEM.CICS.INITIATION.QUEUE.

Trigger monitors

If you want to use a different queue manager or queue, you must build the trigger monitor the MQTMC2 structure: this requires you to write a program using the EXEC CICS START call, because the structure is too long to add as a parameter. Then, pass the MQTMC2 structure as data to the START request for the trigger monitor.

When you use the MQTMC2 structure, you only need to supply the *StrucId*, *Version*, *QName*, and *QMgrName* parameters to the trigger monitor as it does not reference any other fields.

Messages are read from the initiation queue and used to start CICS transactions, using EXEC CICS START, assuming the APPL_TYPE in the trigger message is MQAT_CICS. The reading of messages from the initiation queue is performed under CICS syncpoint control.

Messages are generated when the monitor has started and stopped as well as when an error occurs. These messages are sent to the CSMT transient data queue.

Here are the available versions of the trigger monitor:

Version	Use
amqltmc0	CICS for OS/2 Version 2, CICS for Windows NT Version 2 TXSeries for AIX, Version 4
amqltmc3	CICS Transaction Server for OS/2, Version 4
amqltmc4	TXSeries for Windows NT, Version 4

If you need a trigger monitor for other environments, you need to write a program that can process the trigger messages that the queue manager puts on the initiation queues. Such a program should:

1. Use the MQGET call to wait for a message to arrive on the initiation queue.
2. Examine the fields of the MQTM structure of the trigger message to find the name of the application to start and the environment in which it runs.
3. Issue an environment-specific start command. For example, in OS/390 batch, submit a job to the internal reader.
4. Convert the MQTM structure to the MQTMC2 structure if required.
5. Pass either the MQTMC2 or MQTM structure to the started application. This may contain user data.
6. Associate with your application queue the application that is to serve that queue. You do this by naming the process definition object (if created) in the *ProcessName* attribute of the queue.

Use DEFINE QLOCAL or ALTER QLOCAL. On AS/400 you can also use CRTMQMQ or CHGMQMQ.

For more information on the trigger monitor interface, see the *MQSeries Application Programming Reference* manual.

MQSeries for AS/400 trigger monitors

The following are provided:

AMQSTRG4

This is a trigger monitor that submits an OS/400 job for the process that is to be started, but this means there is a processing overhead associated with each trigger message.

AMQSERV4

This is a trigger server. For each trigger message, this server runs the command for the process in its own job, and can call CICS transactions.

Both the trigger monitor and the trigger server pass an MQTMC2 structure to the programs they start. For a description of this structure, see the *MQSeries Application Programming Reference* manual. Both of these samples are delivered in both source and executable forms.

Properties of trigger messages

The following sections describe some other properties of trigger messages.

Persistence and priority of trigger messages

Trigger messages are not persistent as there is no requirement for them to be so. However, the conditions for generating triggering events *do* persist, hence trigger messages will be generated whenever these conditions are met. In the event that a trigger message is lost, the continued existence of the application message on the application queue will guarantee that the queue manager will generate a trigger message as soon as all the conditions are met.

If a unit of work is rolled-back, any trigger messages it generated will always be delivered.

Trigger messages take the default priority of the initiation queue.

Queue manager restart and trigger messages

Following the restart of a queue manager, when an initiation queue is next opened for input, a trigger message may be put to this initiation queue if an application queue associated with it has messages on it, and is defined for triggering.

Trigger messages and changes to object attributes

Trigger messages are created according to the values of the trigger attributes in force at the time of the trigger event. If the trigger message is not made available to a trigger monitor until later (because the message that caused it to be generated was put within a unit of work), any changes to the trigger attributes in the meantime have no effect on the trigger message. In particular, disabling triggering does not prevent a trigger message being made available once it has been created. Also, the application queue may no longer exist at the time the trigger message is made available.

Format of trigger messages

The format of a trigger message is defined by the MQTM structure. This has the following fields, which the queue manager fills when it creates the trigger message, using information in the object definitions of the application queue and of the process associated with that queue:

StrucId

The structure identifier.

Version

The version of the structure.

QName

The name of the application queue on which the trigger event occurred.

Trigger message properties

When the queue manager creates a trigger message, it fills this field using the *QName* attribute of the application queue.

ProcessName

The name of the process definition object that is associated with the application queue. When the queue manager creates a trigger message, it fills this field using the *ProcessName* attribute of the application queue.

TriggerData

A free-format field for use by the trigger monitor. When the queue manager creates a trigger message, it fills this field using the *TriggerData* attribute of the application queue. On an MQSeries Version 5 product (except MQSeries for OS/390) or MQSeries for VSE/ESA, this field can be used to specify the name of the channel to be triggered.

ApplType

The type of the application that the trigger monitor is to start. When the queue manager creates a trigger message, it fills this field using the *ApplType* attribute of the process definition object identified in *ProcessName*.

ApplId A character string that identifies the application that the trigger monitor is to start. When the queue manager creates a trigger message, it fills this field using the *ApplId* attribute of the process definition object identified in *ProcessName*. When you use an MQSeries for OS/390-supplied trigger monitor (CKTI or CSQQTRMN) the *ApplId* attribute of the process definition object is a CICS or IMS transaction identifier.

EnvData

A character field containing environment-related data for use by the trigger monitor. When the queue manager creates a trigger message, it fills this field using the *EnvData* attribute of the process definition object identified in *ProcessName*. The MQSeries for OS/390-supplied trigger monitors (CKTI or CSQQTRMN) do not use this field, but other trigger monitors may choose to use it.

UserData

A character field containing user data for use by the trigger monitor. When the queue manager creates a trigger message, it fills this field using the *UserData* attribute of the process definition object identified in *ProcessName*. This field can be used to specify the name of the channel to be triggered.

There is a full description of the trigger message structure in the *MQSeries Application Programming Reference* manual.

When triggering does not work

A program is not triggered if the trigger monitor cannot start the program or the queue manager cannot deliver the trigger message.

If a trigger message is created but cannot be put on the initiation queue (for example, because the queue is full or the length of the trigger message is greater than the maximum message length specified for the initiation queue), the trigger message is put instead on the dead-letter (undelivered-message) queue.

If the put operation to the dead-letter queue cannot complete successfully, the trigger message is discarded and a warning message is sent to the console (OS/390) or to the system operator (AS/400), or put on the error log.

Putting the trigger message on the dead-letter queue may generate a trigger message for that queue. This second trigger message is discarded if it adds a message to the dead-letter queue.

If the program is triggered successfully but abends before it gets the message from the queue, use a trace utility (for example, CICS AUXTRACE if the program is running under CICS) to find out the cause of the failure.

How CKTI detects errors

If the CKTI trigger monitor in MQSeries for OS/390 detects an error in the structure of a trigger message, or if it cannot start a program, it puts the trigger message on the dead-letter (undelivered-message) queue. CKTI adds a dead-letter header structure (MQDLH) to the trigger message. It uses a feedback code in the *Reason* field of this structure to explain why it put the message on the dead-letter (undelivered-message) queue.

An instance of CKTI stops serving an initiation queue if it attempts to get a trigger message from the queue and finds that the attributes of the queue have changed since it last accessed that queue. The attributes could have been changed by another program, or by an operator using the commands or operations and control panels of MQSeries. CKTI produces an error message, which includes a reason code, explaining the action it has taken.

How CSQQTRMN detects errors

If the CSQQTRMN trigger monitor in MQSeries for OS/390 detects an error in the structure of a trigger message, or if it cannot start a program, it puts the trigger message on the dead-letter (undelivered-message) queue and sends a diagnostic message to a user specified LTERM (the default is MASTER). CSQQTRMN adds a dead-letter header structure (MQDLH) to the trigger message. It uses a feedback code in the *Reason* field of this structure to explain why it put the message on the dead-letter (undelivered-message) queue. If any other errors are detected, CSQQTRMN sends a diagnostic message to the specified LTERM, and then terminates.

How RUNMQTRM detects errors

If the RUNMQTRM trigger monitor in MQSeries for OS/2 Warp and MQSeries on UNIX systems detects an error in either the:

- Structure of a trigger message
- Application type is unsupported

or it either:

- Cannot start a program
- Detects a data-conversion error

it puts the trigger message on the dead-letter (undelivered-message) queue, having added a dead-letter header structure (MQDLH) to the message. It uses a feedback code in the *Reason* field of this structure to explain why it put the message on the dead-letter (undelivered-message) queue.

Chapter 15. Using and writing applications on MQSeries for OS/390

MQSeries for OS/390 applications can be made up from programs that run in many different environments. This means they can take advantage of the facilities available in more than one environment.

This chapter explains the MQSeries facilities available to programs running in each of the supported environments. In addition,

- For information on using the MQSeries-CICS bridge, see “Chapter 16. Using and writing MQSeries-CICS bridge applications for OS/390” on page 231.
- For information on using IMS and the IMS Bridge, see “Chapter 17. IMS and IMS Bridge applications on MQSeries for OS/390” on page 265.

This chapter introduces MQSeries for OS/390 applications, under these headings:

- “Environment-dependent MQSeries for OS/390 functions”
- “Program debugging facilities” on page 214
- “Syncpoint support” on page 214
- “Recovery support” on page 215
- “The MQSeries for OS/390 interface with the application environment” on page 215
- “Writing OS/390 OpenEdition applications” on page 220
- “The API-crossing exit for OS/390” on page 221
- “MQSeries Workflow” on page 225
- “Application programming with shared queues” on page 226

Important notice

Distributed queuing using CICS ISC is retained for compatibility with previous releases; there will be no further enhancements to this function. Therefore you are recommended to use the channel initiator for distributed queuing.

Environment-dependent MQSeries for OS/390 functions

The main differences to be considered between MQSeries functions in the environments in which MQSeries for OS/390 runs are:

- MQSeries for OS/390 supplies the following trigger monitors:
 - CKTI for use in the CICS environment
 - CSQQTRMN for use in the IMS environment

You must write your own module to start applications in other environments.

- Syncpointing using two-phase commit is supported in the CICS and IMS environments. It is also supported in the OS/390 batch environment using transaction management and recoverable resource manager services (RRS). Single-phase commit is supported in the OS/390 environment by MQSeries itself.
- For the batch and IMS environments, the MQI provides calls to connect programs to, and to disconnect them from, a queue manager. Programs can connect to more than one queue manager.

Environment-dependent functions

- A CICS system can connect to only one queue manager. This can be made to happen when CICS is initiated if the subsystem name is defined in the CICS system start-up job. The MQI connect and disconnect calls are tolerated, but have no effect, in the CICS environment.
- The API-crossing exit allows a program to intervene in the processing of all MQI calls. This exit is available in the CICS environment only.
- In CICS on multiprocessor systems, some performance advantage is gained because MQI calls can be executed under multiple OS/390 TCBs. For more information, see the *MQSeries for OS/390 Concepts and Planning Guide*.

These features are summarized in Table 10.

Table 10. OS/390 environmental features

	CICS	IMS	Batch/TSO
Trigger monitor supplied	Yes	Yes	No
Two-phase commit	Yes	Yes	Yes
Single-phase commit	Yes	No	Yes
Connect/disconnect MQI calls	Tolerated	Yes	Yes
API-crossing exit	Yes	No	No
Note: Two-phase commit is supported in the Batch/TSO environment using RRS.			

Program debugging facilities

MQSeries for OS/390 provides a trace facility that you can use to debug your programs in all environments. Additionally, in the CICS environment you can use:

- The CICS Execution Diagnostic Facility (CEDF)
- The CICS Trace Control Transaction (CETR)
- The MQSeries for OS/390 API-crossing exit

On the OS/390 platform, you can use any available interactive debugging tool that is supported by the programming language you are using.

Syncpoint support

The synchronization of the start and end of units of work is necessary in a transaction processing environment so that transaction processing can be used safely. This is fully supported by MQSeries for OS/390 in the CICS and IMS environments. Full support means cooperation between resource managers so that units of work can be committed or backed out in unison, under control of CICS or IMS. Examples of resource managers are DB2, CICS File Control, IMS, and MQSeries for OS/390.

OS/390 batch applications can use MQSeries for OS/390 calls to give a single-phase commit facility. This means that an application-defined set of queue operations can be committed, or backed out, without reference to other resource managers.

Two-phase commit is also supported in the OS/390 batch environment using transaction management and recoverable resource manager services (RRS). For further information see “Transaction management and recoverable resource manager services” on page 181.

Recovery support

If the connection between a queue manager and a CICS or IMS system is broken during a transaction, some units of work may not be backed out successfully. However, these units of work are resolved by the queue manager (under the control of the syncpoint manager) when its connection with the CICS or IMS system is reestablished.

The MQSeries for OS/390 interface with the application environment

To allow applications running in different environments to send and receive messages through a message queuing network, MQSeries for OS/390 provides an *adapter* for each of the environments it supports. These adapters are the interface between the application programs and an MQSeries for OS/390 subsystem. They allow the programs to use the MQI.

The batch adapter

The *batch adapter* provides access to MQSeries for OS/390 resources for programs running in:

- Task (TCB) mode
- Problem or Supervisor state
- Primary address space control mode

The programs must not be in cross-memory mode.

Connections between application programs and MQSeries for OS/390 are at the task level. The adapter provides a single connection thread from an application task control block (TCB) to MQSeries for OS/390.

The adapter supports a single-phase commit protocol for changes made to resources owned by MQSeries for OS/390; it does not support multiphase-commit protocols.

RRS batch adapter

The transaction management and recoverable resource manager services (RRS) adapter:

- Uses OS/390 RRS for commit control.
- Supports simultaneous connections to multiple MQSeries subsystems running on a single OS/390 instance from a single task.
- Provides OS/390-wide coordinated commitment control (via OS/390 RRS) for recoverable resources accessed via OS/390 RRS compliant recoverable managers for:
 - Applications that connect to MQSeries using the RRS batch adapter.
 - DB2 stored procedures executing in a DB2 stored procedures address space that is managed by an OS/390 workload manager (WLM).
- Supports the ability to switch an MQSeries batch thread between TCBS.

RRS batch adapter

MQSeries for OS/390 provides two RRS batch adapters:

CSQBRSTB

This adapter requires you to change any MQCMIT and MQBACK statements in your MQSeries application to SRRCMIT and SRRBACK respectively. (If you code MQCMIT or MQBACK in an application linked with CSQBRSTB, you will receive MQRC_ENVIRONMENT_ERROR.)

CSQBRRSI

This adapter allows your MQSeries application to use either MQCMIT and MQBACK or SRRCMIT and SRRBACK.

Note: CSQBRSTB and CSQBRRSI are shipped with linkage attributes AMODE(31) RMODE(ANY). If your application loads either stub below the 16 MB line, you must first relink the stub with RMODE(24).

Migration

It is possible to migrate existing Batch/TSO MQSeries applications to exploit RRS coordination with few or no changes. If you link-edit your MQSeries application with the CSQBRRSI adapter, MQCMIT and MQBACK syncpoint your unit of work across MQSeries and all other RRS-enabled resource managers. If you link-edit your MQSeries application with the CSQBRSTB adapter you must change MQCMIT and MQBACK to SRRCMIT and SRRBACK respectively. The latter approach may be preferable as it clearly indicates that the syncpoint is not restricted to MQSeries resources only.

The CICS adapter

Note to users

If you are using the CICS adapter from an MQSeries for OS/390 system, ensure that CICS is able to obtain sufficient storage to accommodate messages up to 100 MB long.

A CICS system can have only one connection to an MQSeries for OS/390 queue manager, and this connection is managed by the MQSeries for OS/390 *CICS adapter*. The CICS adapter provides access to MQSeries for OS/390 resources for CICS programs.

In addition to providing access to the MQI calls, the adapter provides:

- A trigger monitor (or task initiator) program that can start programs automatically when certain trigger conditions on a queue are met. For more information, see “Chapter 14. Starting MQSeries applications using triggers” on page 191.
- An API-crossing exit that can be invoked before and after each MQI call. For more information, see “The API-crossing exit for OS/390” on page 221.
- A trace facility to help you when debugging programs.
- Facilities that allow the MQI calls to be executed under multiple OS/390 TCBs. For more information, see the *MQSeries for OS/390 Concepts and Planning Guide*.

CICS adapter performance considerations

This section describes how the CICS adapter optimizes the performance of a CICS to MQSeries connection.

There are a number of factors to be taken into consideration when performance is critical:

First MQI call

In general, the first MQI call of a task takes longer to perform than subsequent calls. This is because the environment must be set up. For example, the adapter must acquire storage and security information, and control blocks must be allocated and formatted.

MQGET and the SIGNAL option

Using the SIGNAL option with an MQGET call imposes an additional overhead. This is because the SIGNAL option can produce a CICS GETMAIN in the adapter, which is used to record the address of the ECB so that it can be posted if the queue manager abends.

API-crossing exit

Using the API-crossing exit also imposes a host processor overhead on each MQI call. The overhead in handling the exit parameter block and the invocations are minimal, but the exit can be invoked twice for each MQI call through EXEC CICS LINK.

CICS tracing

CICS tracing in the adapter also increases the pathlength of an MQI call. A large number of trace entries can be generated depending on how busy the system is. There is no control over the granularity of the trace entries produced in the adapter. Therefore, tracing should only be switched on if necessary.

MQGET and the WAIT option

Using MQGET with the WAIT option is less efficient if the task has been put into a wait until a message arrives. The adapter implements the wait as a form of CICS wait. When a message arrives, the adapter effectively re-issues the MQGET call for the application.

Therefore, use the WAIT option with care.

MQCLOSE

Issuing an MQCLOSE call is not always necessary because MQSeries automatically closes any unclosed handles when the task ends.

MQPUT1

If there is only one message to be put, MQPUT1 is more efficient than an MQOPEN-MQPUT-MQCLOSE sequence because only one flow is generated between the MQSeries and the adapter, instead of three.

If there are multiple messages to be put, MQOPEN-MQPUT...MQPUT-MQCLOSE should be used.

EXEC CICS RETURN

Implicit syncpointing generated by EXEC CICS RETURN is more efficient than issuing the explicit syncpoint call EXEC CICS SYNCPOINT followed by EXEC CICS RETURN.

The EXEC CICS RETURN call accommodates all the work needed for syncpointing and task termination into one flow to MQSeries instead of the two separate flows used when explicit syncpointing is used.

Two-phase commit

A two-phase commit consumes more resources than a single-phase commit, both in host processor cost and response time. This is because a two-phase commit involves one more flow to MQSeries and more physical logging. If

The CICS adapter

an application is restricted to recoverable updates in MQSeries and no other resource managers, CICS invokes the adapter for a single-phase commit.

Syncpoint bypassing

The adapter does not use the read-only commit feature in CICS. When a transaction is restricted to non-recoverable or non-destructive work in MQSeries, syncpointing is bypassed because it is not necessary. The clean-up process is performed when the task ends.

Statistics collection

Statistics collection by connection and by task is done on a per MQI call basis and cannot be switched off. This overhead is negligible.

You can use the CKQC transaction to display statistics for the current connection.

The adapter supports a two-phase commit protocol for changes made to resources owned by MQSeries for OS/390, with CICS acting as the syncpoint coordinator.

The CICS adapter also supplies facilities (for use by system programmers and administrators) for managing the CICS-MQSeries for OS/390 connection, and for collecting task and connection statistics. These facilities are described in the *MQSeries for OS/390 System Administration Guide*.

Adapter trace points

Application programmers can use trace points related to the MQI calls—for example, CSQCGMGD (GET Message Data)—for debugging CICS application programs. System programmers can use trace points related to system events, such as recovery and task switching, for diagnosing system-related problems. For full details of trace points in the CICS adapter, see the *MQSeries for OS/390 Problem Determination Guide*.

Some trace data addresses are passed by applications. If the address of the trace data is in the private storage area of the CICS region, the contents of the area are traced when necessary. For example, this would be done for the trace entries CSQCGMGD (GET Message Data) or CSQCMPMGD (PUT Message Data). If the address is not in the private storage area, message CSQC416I is written to the CICS trace—this contains the address in error.

Abends

This section describes some of the things you must consider with regard to CICS AEY9 and QLOP abends. For information about all other abends, see the *MQSeries for OS/390 Messages and Codes* manual.

CICS AEY9 abends: A transaction does *not* abend with a CICS AEY9 code if it issues an MQI call before the adapter is enabled. Instead, it receives return code MQCC_FAILED and reason code MQRC_ADAPTER_NOT_AVAILABLE.

For more information about CICS AEY9 abends, see the *CICS Messages and Codes* manual.

QLOP abends: Tasks abend with the abend code QLOP if a second MQI call is made after a call has been returned with completion code MQCC_FAILED and one of these reason codes:

- MQRC_CONNECTION_BROKEN
- MQRC_Q_MGR_NAME_ERROR
- MQRC_Q_MGR_NOT_AVAILABLE

MQRC_Q_MGR_STOPPING
 MQRC_CONNECTION_STOPPING
 MQRC_CONNECTION_NOT_AUTHORIZED

This runaway mechanism can be activated only after the adapter has been enabled once. Before the adapter has been enabled, such a task will loop with reason code set to MQRC_ADAPTER_NOT_AVAILABLE. To avoid this, ensure that your applications respond to the above reason codes either by terminating abnormally or by issuing an EXEC CICS SYNCPOINT ROLLBACK and terminating normally.

If the application does not terminate at this point, it might not issue any further MQSeries calls even if the connection between MQSeries and CICS is re-established. Once MQSeries is reconnected to CICS, new transactions can use MQI calls as before.

Using the CICS Execution Diagnostic Facility

You can use the CICS execution diagnostic facility (CEDF) to monitor applications that use the CICS adapter. For details of how to use CEDF, see the *CICS Application Programming Guide*.

CEDF uses standard formatting to display MQI calls.

- Before the MQI call is executed:
 - CEDF displays the addresses of the call parameters
 - You can use the Working Storage key to verify or modify their contents
 - You can skip the call by overtyping the command with NOOP
- After the call has completed:
 - The results are returned in the program's storage
 - The return code and reason code are displayed in the call parameter list
 - You can modify them before returning to the application program

See the *MQSeries for OS/390 Problem Determination Guide* for examples of the output produced by this facility.

The IMS adapter

Note to users

If you are using the IMS adapter from an MQSeries for OS/390 system, ensure that IMS is able to obtain sufficient storage to accommodate messages up to 100 MB long.

The *IMS adapter* provides access to MQSeries for OS/390 resources for

- On-line message processing programs (MPPs)
- Interactive Fast Path programs (IFPs)
- Batch message processing programs (BMPs)

To use these resources, the programs must be running in task (TCB) mode and problem state; they must not be in cross-memory mode or access-register mode.

The adapter provides a connection thread from an application task control block (TCB) to MQSeries. The adapter supports a two-phase commit protocol for changes made to resources owned by MQSeries for OS/390, with IMS acting as the syncpoint coordinator.

The IMS adapter

The adapter also provides a trigger monitor program that can start programs automatically when certain trigger conditions on a queue are met. For more information, see “Chapter 14. Starting MQSeries applications using triggers” on page 191.

If you are writing batch DL/I programs, follow the guidance given in this book for OS/390 batch programs.

Writing OS/390 OpenEdition applications

The batch adapter supports queue manager connections from Batch and TSO address spaces:

If we consider a Batch address space, the adapter supports connections from multiple TCBS within that address space as follows:

- Each TCB can connect to multiple queue managers via the MQCONN or MQCONNX call (but a TCB can only have one instance of a connection to a particular queue manager at any one time).
- Multiple TCBS can connect to the same queue manager (but the queue manager handle returned on any MQCONN or MQCONNX call is bound to the issuing TCB and cannot be used by any other TCB).

OS/390 OpenEdition supports two types of pthread_create call:

1. Heavyweight threads, run one per TCB, that are ATTACHED and DETACHED at thread start and end by OS/390.
2. Mediumweight threads, run one per TCB, but the TCB can be one of a pool of long-running TCBS. The onus is on the application to perform all necessary application clean up, since, if it is connected to a server, the default thread termination that may be provided by the server at Task (TCB) termination, will *not* always be driven.

Lightweight threads are not supported. (If an application creates permanent threads which do their own dispatching of work requests, then the *application* is responsible for cleaning up any resources before starting the next work request.)

MQSeries for OS/390 supports OS/390 OpenEdition threads via the Batch Adapter as follows:

1. Heavyweight threads are fully supported as Batch connections. Each thread runs in its own TCB which is ATTACHED and DETACHED at thread start and end. Should the thread end before issuing an MQDISC call, then MQSeries for OS/390 performs its standard task clean up which includes committing any outstanding unit of work if the thread terminated normally, or backing it out if the thread terminated abnormally.
2. Mediumweight threads are fully supported but if the TCB is going to be reused by another thread, then the application must ensure that an MQDISC call, preceded by either MQCMIT or MQBACK, is issued prior to the next thread start. This implies that if the application has established a Program Interrupt Handler, and the application then abends, then the Interrupt Handler should issue MQCMIT and MQDISC calls before reusing the TCB for another thread.

Again, lightweight threads are not supported.

Note: Threading models do *not* support access to common MQSeries resources from multiple threads.

The API-crossing exit for OS/390

This section contains product-sensitive programming interface information.

An exit is a point in IBM-supplied code where you can run your own code. MQSeries for OS/390 provides an *API-crossing exit* that you can use to intercept calls to the MQI, and to monitor or modify the function of the MQI calls. This section describes how to use the API-crossing exit, and describes the sample exit program that is supplied with MQSeries for OS/390.

Note

The API-crossing exit is invoked only by the CICS adapter of MQSeries for OS/390. The exit program runs in the CICS address space.

Using the API-crossing exit

You could use the API-crossing exit to:

- Operate additional security checks by examining the contents of each message before and after each MQI call
- Replace the queue name supplied in the message with another queue name
- Cancel the call and either issue a return code of 0 to simulate a successful call, or another value to indicate that the call was not performed
- Monitor the use of MQI calls in an application
- Gather statistics
- Modify input parameters on specific calls
- Modify the results of specific calls

Defining the exit program

Before the exit can be used, an exit program load module must be available when the CICS adapter connects to MQSeries for OS/390. The exit program is a CICS program that must be named CSQCAPX and reside in a library in the DFHRPL concatenation. CSQCAPX must be defined in the CICS system definition file (CSD), and the program must be enabled.

When CSQCAPX is loaded, a confirmation message is written to the CKQC adapter control panel or to the console. If the program cannot be loaded, a diagnostic message is displayed.

How the exit is invoked

When enabled, the API-crossing exit is invoked:

- By *all* applications that use the CICS adapter of MQSeries for OS/390
- For the following MQI calls:
 - MQCLOSE
 - MQGET
 - MQINQ
 - MQOPEN
 - MQPUT
 - MQPUT1
 - MQSET
- Every time one of these MQI calls is made
- Both before *and* after a call

Using the API-crossing exit

This means that using the API-crossing exit degrades the performance of MQSeries for OS/390, so plan your use of it carefully.

The exit program can be invoked once *before* a call is executed, and once *after* the call is executed. On the before type of exit call, the exit program can modify any of the parameters on the MQI call, suppress the call completely, or allow the call to be processed. If the call is processed, the exit is invoked again after the call has completed.

Note: The exit program is not recursive. Any MQI calls made inside the exit do not invoke the exit program for a second time.

Communicating with the exit program

After it has been invoked, the exit program is passed a parameter list in the CICS communication area pointed to by a field called DFHEICAP. The CICS Exec Interface Block field EIBCALEN shows the length of this area. The structure of this communication area is defined in the CMQXPA assembler-language macro that is supplied with MQSeries for OS/390 :

```
*
MQXP_COPYPLIST      DSECT
                    DS  0D          Force doubleword alignment
MQXP_PXPB           DS  AL4        Pointer to exit parameter block
MQXP_PCOPYPARM      DS  11AL4      Copy of original plist
*
                    ORG  MQXP_PCOPYPARM
MQXP_PCOPYPARM1     DS  AL4        Copy of 1st parameter
MQXP_PCOPYPARM2     DS  AL4        Copy of 2nd parameter
MQXP_PCOPYPARM3     DS  AL4        Copy of 3rd parameter
MQXP_PCOPYPARM4     DS  AL4        Copy of 4th parameter
MQXP_PCOPYPARM5     DS  AL4        Copy of 5th parameter
MQXP_PCOPYPARM6     DS  AL4        Copy of 6th parameter
MQXP_PCOPYPARM7     DS  AL4        Copy of 7th parameter
MQXP_PCOPYPARM8     DS  AL4        Copy of 8th parameter
MQXP_PCOPYPARM9     DS  AL4        Copy of 9th parameter
MQXP_PCOPYPARM10    DS  AL4        Copy of 10th parameter
MQXP_PCOPYPARM11    DS  AL4        Copy of 11th parameter
*
MQXP_COPYPLIST_LENGTH EQU  *-MQXP_PXPB
                    ORG  MQXP_PXPB
MQXP_COPYPLIST_AREA  DS  CL(MQXP_COPYPLIST_LENGTH)
*
```

Field *MQXP_PXPB* points to the exit parameter block, MQXP.

Field *MQXP_PCOPYPARM* is an array of addresses of the call parameters. For example, if the application issues an MQI call with parameters P1,P2,or P3, the communication area contains:

```
PXPB,PP1,PP2,PP3
```

where *P* denotes a pointer (address) and *XPB* is the exit parameter block.

Writing your own exit program

You can use the sample API-crossing exit program (CSQCAPX) that is supplied with MQSeries for OS/390 as a framework for your own program. This is described on page 224.

When writing an exit program, to find the name of an MQI call issued by an application, examine the *ExitCommand* field of the MQXP structure. To find the number of parameters on the call, examine the *ExitParmCount* field. You can use

the 16-byte *ExitUserArea* field to store the address of any dynamic storage that the application obtains. This field is retained across invocations of the exit and has the same life time as a CICS task.

Your exit program can suppress execution of an MQI call by returning MQXCC_SUPPRESS_FUNCTION or MQXCC_SKIP_FUNCTION in the *ExitResponse* field. To allow the call to be executed (and the exit program to be reinvoked after the call has completed), your exit program must return MQXCC_OK.

When invoked after an MQI call, an exit program can inspect and modify the completion and reason codes set by the call.

Usage notes

Here are some general points you should bear in mind when writing your exit program:

- For performance reasons, you should write your program in assembler language. If you write it in any of the other languages supported by MQSeries for OS/390, you must provide your own data definition file.
- Link-edit your program as AMODE(31) and RMODE(ANY).
- To define the exit parameter block to your program, use the assembler-language macro, CMQXPA.
- If you are using the CICS Transaction Server for OS/390 storage protection feature, your program must run in CICS execution key. That is, you must specify EXECKEY(CICS) when defining both your exit program and any programs to which it passes control. For information about CICS exit programs and the CICS storage protection facility, see the *CICS Customization Guide*.
- Your program can use all the APIs (for example, IMS, DB2, and CICS) that a CICS task-related user exit program can use. It can also use any of the MQI calls except MQCONN, MQCONNX, and MQDISC. However, any MQI calls within the exit program do not invoke the exit program a second time.
- Your program can issue EXEC CICS SYNCPOINT or EXEC CICS SYNCPOINT ROLLBACK commands. However, these commands commit or roll back *all* the updates done by the task up to the point that the exit was used, and so their use is not recommended.
- Your program must end by issuing an EXEC CICS RETURN command. It must not transfer control with an XCTL command.
- Exits are written as extensions to the MQSeries for OS/390 code. You must take great care that your exit does not disrupt any MQSeries for OS/390 programs or transactions that use the MQI. These are usually indicated with a prefix of "CSQ" or "CK".
- If CSQCAPX is defined to CICS, the CICS system will attempt to load the exit program when CICS connects to MQSeries for OS/390. If this attempt is successful, message CSQC301I is sent to the CKQC panel or to the system console. If the load is unsuccessful (for example, if the load module does not exist in any of the libraries in the DFHRPL concatenation), message CSQC315 is sent to the CKQC panel or to the system console.
- Because the parameters in the communication area are addresses, the exit program must be defined as local to the CICS system (that is, not as a remote program).

The CSQCAPX sample

The sample API-crossing exit program, CSQCAPX

The sample exit program is supplied as an assembler-language program. The source file (CSQCAPX) is supplied in the library **thlqual.SCSQASMS** (where **thlqual** is the high-level qualifier used by your installation). This source file includes pseudocode that describes the program logic.

The sample program contains initialization code and a layout that you can use when writing your own exit programs.

The sample shows how to:

- Set up the exit parameter block
- Address the call and exit parameter blocks
- Determine for which MQI call the exit is being invoked
- Determine whether the exit is being invoked before or after processing of the MQI call
- Put a message on a CICS temporary storage queue
- Use the macro DFHEIENT for dynamic storage acquisition to maintain reentrancy
- Use DFHEIBLK for the CICS exec interface control block
- Trap error conditions
- Return control to the caller

Design of the sample exit program

The sample exit program writes messages to a CICS temporary storage queue (CSQ1EXIT) to show the operation of the exit. The messages show whether the exit is being invoked before or after the MQI call. If the exit is invoked after the call, the message contains the completion code and reason code returned by the call. The sample uses named constants from the CMQXPA macro to check on the type of entry (that is, before or after the call).

The sample does not perform any monitoring function, but simply places time-stamped messages into a CICS queue indicating the type of call it is processing. This provides an indication of the performance of the MQI, as well as the proper functioning of the exit program.

Note: The sample exit program issues six EXEC CICS calls for each MQI call that is made while the program is running. If you use this exit program, MQSeries for OS/390 performance is degraded.

Preparing and using the API-crossing exit

The sample exit is supplied in source form only. To use the sample exit, or an exit program that you have written, you must create a load library, as you would for any other CICS program, as described on page 313.

- For CICS Transaction Server for OS/390 and CICS for MVS/ESA, when you update the CICS system definition (CSD) data set, the definitions you need are in the member **thlqual.SCSQPROC(CSQ4B100)**.

Note: The definitions use a suffix of MQ. If this suffix is already used in your enterprise, this must be changed before the assembly stage.

If you use the default CICS program definitions supplied, the exit program CSQCAPX is installed in a *disabled* state. This is because using the exit program can produce a significant reduction in performance.

Preparing the CSQCAPX sample

To activate the API-crossing exit temporarily:

1. Issue the command CEMT S PROGRAM(CSQCAPX) ENABLED from the CICS master terminal.
2. Run the CKQC transaction, and use option 3 in the Connection pull-down to alter the status of the API-crossing exit to 'Enabled'.

If you want to run MQSeries for OS/390 with the API-crossing exit permanently enabled, do one of the following:

- For CICS Transaction Server for OS/390 and CICS for MVS/ESA do one of the following:
 - Alter the CSQCAPX definition in member CSQ4B100, changing STATUS(DISABLED) to STATUS(ENABLED). You can update the CICS CSD definition using the CICS-supplied batch program DFHCSDUP.
 - Alter the CSQCAPX definition in the CSQCAT1 group by changing the status from DISABLED to ENABLED.

In both cases you must reinstall the group. You can do this by cold-starting your CICS system or by using the CICS CEDA transaction to reinstall the group while CICS is running.

Note: Using CEDA may cause an error if any of the entries in the group are currently in use.

End of product-sensitive programming interface information.

MQSeries Workflow

MQSeries Workflow on OS/390 is a tool which helps companies improve their business processes. OS/390 workload manager (WLM) addresses the need for:

- Managing workload distribution
- Load balancing
- Distribution of computing resources to competing workloads

MQSeries support for OS/390 workload manager introduces a new type of local queue: a WLM-managed queue. It is recognized by a new value of the INDXTYPE attribute called MSGTOKEN. The initiation queue associated with a WLM-managed queue should have TRIGTYPE defined as NONE and no ordinary local queues should be associated with this initiation queue.

If an MQSeries Workflow server application has the initiation queue open for input, MQSeries updates a WLM worklist as part of commit processing of MQPUTs to the WLM-managed queue. The setting of TRIGGER or NOTRIGGER on the WLM-managed queue has no effect on the updating of this WLM worklist.

The PROCESS definition is used to provide the name of the application_environment associated with a WLM-managed queue. This is passed in the APPLICID attribute. You should ensure that a WLM-managed queue uniquely references an associated process and that two processes do not specify the same APPLICID value.

Messages are retrieved from a WLM-managed queue using a unique message_token which must be passed to MQGET. To do this, a new message_token value (MQGMO_MSGTOKEN) and a new get message match option (MQMO_MATCH_MSG_TOKEN) are used. Workflow does not normally issue

MQSeries Workflow

MQGET calls until the message is placed successfully on the queue. If the application needs to wait for the arrival of a message, it must set the match option to MQMO_NONE.

There are new MQRC values for MQGET (MQRC_MSG_TOKEN_ERROR) and MQPUT (MQRC_MISSING_WIH and MQRC_WIH_ERROR).

MQRC_MISSING_WIH is returned if a message, MQPUT to a WLM-managed queue, does not include the new work information header (MQWIH).

MQRC_WIH_ERROR is returned if the message data does not conform to an MQWIH. MQGET does not remove this header from the message.

Note: You may experience excessive CPU usage if your OS/390 system is at Version 2.5 or earlier and the number of messages on WLM-managed queues exceeds 500.

For further information see *IBM MQSeries Workflow: Concepts and Architecture*, GH12-6285 and *IBM MQSeries Workflow for OS/390: Customization and Administration*, SC33-7030.

Application programming with shared queues

This section discusses some of the factors you need to take into account when designing new applications that will use shared queues, and when migrating existing applications to the shared-queue environment.

Serializing your applications

Certain types of applications might have to ensure that messages are retrieved from a queue in exactly the same order as they arrived on the queue. For example, if MQSeries is being used to shadow database updates on to a remote system, a message describing the update to a record must be processed after a message describing the insert of that record. In a local queuing environment, this is often achieved by the application that is getting the messages opening the queue with the MQOO_INPUT_EXCLUSIVE option, thus preventing any other getting application from processing the queue at the same time.

MQSeries allows applications to open shared queues exclusively in the same way. However, if the application is working from a partition of a queue (for example, all database updates are on the same queue, but those for table A have a correlation identifier of A, and those for table B a correlation identifier of B), and applications want to get messages for table A updates and table B updates concurrently, the simple mechanism of opening the queue exclusively is not possible.

If this type of application is to take advantage of the high availability of shared queues, you might decide that another instance of the application that accesses the same shared queues, running on a secondary queue manager, will take over if the primary getting application or queue manager fails.

If the primary queue manager fails, two things happen:

- Shared queue peer recovery ensures that any incomplete updates from the primary application are completed or backed out.
- The secondary application takes over processing the queue.

The secondary application might start before all the incomplete units of work have been dealt with, which could lead to the secondary application retrieving the messages out of sequence. To solve this type of problem, the application can choose to be a *serialized application*.

A serialized application uses the **MQCONN** call to connect to the queue manager, specifying a connection tag when it connects that is unique to that application. Any units of work performed by the application are marked with the connection tag. MQSeries ensures that units of work within the queue-sharing group with the same connection tag are serialized (according to the serialization options on the **MQCONN** call).

This means that, if the primary application uses the **MQCONN** call with a connection tag of Database shadow retriever, and the secondary takeover application attempts to use the **MQCONN** call with an identical connection tag, the secondary application will not be able to connect to the second MQSeries until any outstanding primary units of work have been completed, in this case by peer recovery.

You should consider using the serialized application technique for applications that depend on the exact sequence of messages on a queue. In particular:

- Applications that must not restart after an application or queue manager failure until all commit and back-out operations for the previous execution of the application are complete.

In this case, the serialized application technique is only applicable if the application works in syncpoint.

- Applications that must not start while another instance of the same application is already running.

In this case, the serialized application technique is only required if the application cannot open the queue for exclusive input.

Note: MQSeries only guarantees to preserve the sequence of messages when certain criteria are met. These are described in the description of the **MQGET** call in the *MQSeries Application Programming Reference*.

Applications that are not suitable for use with shared queues

Some features of MQSeries are not supported when you are using shared queues, so applications that use these features are not suitable for the shared queue environment. You should consider the following points when designing your shared-queue applications:

- Persistent messages are not supported. However, the nonpersistent messages on shared queues are stored in the coupling facility, so they are not lost if an individual queue manager in the queue-sharing group fails. The nonpersistent messages on shared queues are lost only if the Coupling Facility fails.
- Messages on shared queues cannot be greater than 63 KB in size. Because Coupling Facility storage is limited, you must also consider the number of messages to be generated by the application to ensure that the messages will not fill the queue. However, remember that you can monitor the queue and start more versions of the application on different queue managers to service it if this is a problem.
- Queue indexing is limited for shared queues. If you want to use the message identifier or correlation identifier to select the message you want to get from the queue, the queue must have the correct index defined. If you do not define a queue index, applications can only get the next available message.

Programming with shared queues

- You cannot use temporary dynamic queues as shared queues. You can use permanent dynamic queues however. The models for shared dynamic queues have a DEFTYPE of SHAREDYN (shared dynamic) although they are created and destroyed in the same way as PERMDYN (permanent dynamic) queues.

Deciding whether to share non-application queues

There are queues other than application queues that you might want to consider sharing:

Initiation queues

If you define a shared initiation queue, you do not need to have a trigger monitor running on every queue manager in the queue-sharing group, as long as there is at least one trigger monitor running. (You can also use a shared initiation queue even if there is a trigger monitor running on each queue manager in the queue-sharing group.)

If you have a shared application queue and use the trigger type of EVERY (or a trigger type of FIRST with a small trigger interval, which behaves like a trigger type of EVERY) your initiation queue should always be a shared queue. For more information about when to use a shared initiation queue, see Table 11 on page 229.

Dead-letter queue

You should not define your dead-letter queue as a shared queue. This is because shared queues cannot hold persistent messages, or messages with a size greater than 63 KB. If you use a shared dead-letter queue, any persistent messages (or nonpersistent messages longer than 63 KB) that cannot be delivered to their destination queues will be discarded.

SYSTEM.* queues

You can define the SYSTEM.ADMIN.* queues used to hold event messages as shared queues. This can be useful to check load balancing if an exception occurs. Each event message created by MQSeries contains a correlation identifier indicating which queue manager produced it.

You must define the SYSTEM.QSG.* queues used for shared channels and intra-group queuing as shared queues.

You can also change the definitions of the SYSTEM.DEFAULT.LOCAL.QUEUE to be shared, or define your own default shared queue definition. This is described in the section **Defining system objects** in the *MQSeries for OS/390 Concepts and Planning Guide*.

You cannot define any of the other SYSTEM.* queues as shared queues.

Migrating your existing applications to use shared queues

Migrating your existing queues to shared queues is described in the *MQSeries for OS/390 System Administration Guide*.

When you are migrating your existing applications, you should consider the following things, which might work in a slightly differently way in the shared queue environment.

Reason Codes

When you are migrating your existing applications to use shared queues, remember to check for the new reason codes that can be issued.

Triggering

If you are using a shared application queue, triggering works on committed messages only (on a non-shared application queue, triggering works on all messages).

If you use triggering to start applications, you might want to use a shared initiation queue. Table 11 describes what you need to consider when deciding which type of initiation queue to use.

Table 11. When to use a shared-initiation queue

	Non-shared application queue	Shared application queue
Non-shared initiation queue	As for previous releases.	<p>If you are using trigger type of FIRST or DEPTH, you can use a non-shared initiation queue with a shared application queue. There is the possibility of extra trigger messages being generated, but this setup is good for triggering long-running applications (like the CICS bridge) and provides high availability.</p> <p>For trigger type FIRST or DEPTH, a trigger message will trigger an instance of the application on every queue manager that is running a trigger monitor and that does not already have the application queue open for input.</p>
Shared initiation queue	You should not use a shared initiation queue with a non-shared application queue.	<p>If you have a shared application queue that has a trigger type of EVERY, you should use a shared initiation queue or you will lose trigger messages.</p> <p>For trigger type FIRST or DEPTH, one trigger message will be generated by each queue manager that has the named initiation queue open for input. If the initiation queue is defined as a local queue, one trigger message is available to any trigger monitors running on that queue manager against the queue.</p>

MQINQ

When you use the **MQINQ** call to display information about a shared queue, the values of the number of **MQOPEN** calls that have the queue open for input and output relate only to the queue manager that issued the call. No information is produced about other queue managers in the queue-sharing group that have the queue open.

Programming with shared queues

Chapter 16. Using and writing MQSeries-CICS bridge applications for OS/390

Throughout this chapter the terms 'CICS bridge' and 'bridge' mean 'MQSeries-CICS bridge'.

Most CICS applications were developed when a 3270 terminal was the main way of interacting with users. To use these applications with new transport mechanisms, such as MQSeries, CICS recommends that applications are written with the business logic of the application separated from the presentation logic. The business logic can be accessed by a CICS Distributed Program Link (DPL) request to run the program. However, not all applications can be restructured in this way, for example when the customer does not own the source code of the application. The CICS bridge protects the investment in these legacy applications by allowing them to be accessed from other platforms. This makes CICS resources readily available to programs not running under CICS. This can be done for DPL programs and for 3270 transactions.

An MQSeries application can start a CICS application by sending a structured message to the CICS bridge request queue. Any data required by the CICS application can be included in the request message:

- For DPL programs, the data required is the CICS communication area (COMMAREA) data used by the application.
- For 3270 transactions, the data required is vectors describing the Application Data Structures (ADSs) used by the application.

Similarly, the CICS application can send data back to the MQSeries application in a message that is sent to a reply queue:

- For DPL programs, the data sent back is the COMMAREA data output by the application.
- For 3270 transactions, the data sent back is vectors describing the Application Data Structures (ADSs) output by the application.

The MQSeries application can run on any platform, but the bridge request queue must reside on the local OS/390 queue manager that is connected to the CICS adapter.

This chapter describes how to use and design MQSeries-CICS bridge applications, under these headings:

- "Using CICS DPL programs with the bridge" on page 232
- "Using CICS transactions with the bridge" on page 234
- "Programming for the distributed environment" on page 249
- "From 3270 legacy to 3270 bridge - an example" on page 250
- "Setting fields in the MQMD and MQCIH structures" on page 253
- "Setting the open options and put message options for the bridge request queue" on page 256
- "Managing MsgId and CorrelId in a unit of work" on page 256
- "Error handling by the CICS bridge" on page 258
- "Debugging CICS bridge applications" on page 260
- "Application data structure terminology" on page 262

Using CICS DPL programs with the bridge

To link to another program that has been defined to CICS, a CICS application can issue a command like this:

```
EXEC CICS LINK PROGRAM(name) COMMAREA(data-area)
```

For the complete syntax of this command, see the *CICS Application Programming Reference* book.

If you want an MQSeries application to run a CICS application that invokes a CICS DPL program, the MQSeries application must send a structured message to the bridge request queue. In the simplest case, the message data consists only of the name of a DPL program to be run. Follow this by COMMAREA data if you want to make data available to the DPL program when it starts.

If you want to run more than one DPL program within a unit of work, or you prefer a specific transaction code (overwriting the default CKBP), or you require certain levels of authorization to run the DPL program, you must supply information in an MQCIH. The MQCIH must precede the program name and any COMMAREA data that you send.

CICS DPL bridge message structure

These examples show the different structures you can use for messages that run DPL programs through the bridge.

- Use this structure for an application that runs a single DPL program using default processing options, and does not send or receive COMMAREA data:

```
-----  
| MQMD | ProgName |  
-----
```

The program specified by *ProgName* is invoked by CICS as a DPL program.

- Use this structure for an application that runs a single DPL program using default processing options, and sends and receives COMMAREA data:

```
-----  
| MQMD | ProgName | CommareaData |  
-----
```

- Use this structure for an application that runs one or more DPL programs within a unit of work, or needs specific authorization to run the program, but does not send or receive COMMAREA data:

```
-----  
| MQMD | MQCIH | ProgName |  
-----
```

- Use this structure for an application that invokes one or more DPL programs within a unit of work, or needs specific authorization to run the program, and sends and receives COMMAREA data:

```
-----  
| MQMD | MQCIH | ProgName | CommareaData |  
-----
```

If a bridge task running a DPL program ends abnormally, a message is returned to the reply queue with the following structure, whether or not the inbound message preceding the failure contained an MQCIH:

```
-----  
| MQMD | MQCIH | CSQC* message |  
-----
```

*CSQC** message represents an error message that indicates the error type. The value of field *MQCIH.Format* is set to *MQFMT_STRING*, so that the message can be properly converted if the final destination uses a different CCSID and encoding. The *MQCIH* also contains other fields that you can use to diagnose the problem.

Notes:

1. The *MQMD* is shown in the examples to help you visualize the overall structure of the message. This is the structure that you see if you use the general queue browser utility of MQSeries SupportPac™ MA10 "MQSeries for MVS/ESA - ISPF utilities".
2. When you want to send only a program name, and no *COMMAREA* data, to the bridge, the program name must be 8 characters long. It must not be a name that is padded to the right with spaces, or the bridge will report a *COMMAREA* negative length error.
3. When you want to send *COMMAREA* data, you must pad the program name with spaces to the right, to give a total length of eight characters.
4. You can include control data in the message to specify unit of work management, and to provide data for security checking, when you include an MQSeries CICS information header (*CIH*) in the message.

Application programming for the CICS DPL bridge

This C-language code fragment shows how you can construct a message buffer when you want to invoke a DPL program with *COMMAREA* data, and include an MQSeries CICS Information Header (*MQCIH*).

```

/* #defines */
#define PGMNAME "DPLPGM" /* DPL program name */
#define PGMNAMELEN 8
#define CALEN 100 /* Commarea length */
:
/* Data declarations */
MQMD mqmd ; /* Message descriptor */
MQCIH mqcih ; /* CICS information header */
MQCHAR * Commarea ; /* Commarea pointer */
MQCHAR * MsgBuffer ; /* Message buffer pointer */
:
/* allocate storage for the buffers */

Commarea = malloc(CALEN * sizeof(MQCHAR)) ;
MsgBuffer = malloc(sizeof(MQCIH) + PGMNAMELEN + CALEN) ;
:
/* Initialize commarea with data */
:
/* Initialize fields in the MQMD as required, including: */

mqmd.MsgId = MQMI_NONE ;
mqmd.CorrelId = MQCI_NEW_SESSION ;

/* Initialize fields in the MQCIH as required */
:
/* Copy the MQCIH to the start of the message buffer */

memcpy(MsgBuffer, &mqcih, sizeof(MQCIH)) ;

/* Set 8 bytes after the MQCIH to spaces */

memset(MsgBuffer + sizeof(MQCIH), ' ', PGMNAMELEN) ;

/* Append the program name to the MQCIH. If it is less than */
/* 8 characters, it is now padded to the right with spaces. */

```

Programming for the CICS DPL bridge

```
strncpy(MsgBuffer + sizeof(MQCIH), PGMNAME, PGMNAMELEN) ;  
/* Append the commarea after the program name */  
memcpy(MsgBuffer + sizeof(MQCIH) + PGMNAMELEN, &Commarea  
        CALEN ) ;  
/* The message buffer is now ready for the MQPUT */  
/* to the Bridge Request Queue. */  
:
```

The DPL program that is invoked must be defined to CICS with *EXECUTIONSET=DPLSUBSET*, and it must conform to the DPL subset rules. See the *CICS Application Programming Guide* for further details.

Managing units of work for the CICS DPL bridge

If your Bridge application is running just a single DPL program, set the value of *MQCIH.UOWControl* to *MQCUOWC_ONLY*. However, if your application is sending and receiving multiple messages, you must handle units of work correctly for the CICS DPL bridge. If you want to run multiple user programs within a unit of work, you should

- set the value of *MQCIH.UOWControl* to *MQCUOWC_FIRST* in the first request
- set *MQCUOWC_MIDDLE* in any intermediate requests
- set *MQCUOWC_LAST* in the last request

Your application can send multiple request messages within a unit of work before receiving any response messages. At any time after the first message, you can terminate the unit of work by sending a message with *MQCIH.UOWControl* set to *MQCUOWC_COMMIT* or *MQCUOWC_BACKOUT*.

Using CICS transactions with the bridge

Without using MQSeries, a CICS transaction can be started in several ways, including:

- A terminal user can enter the transaction name, followed (optionally) by data. The transaction can obtain any data that follows its identifier by issuing EXEC CICS RECEIVE when it starts.
- A preceding transaction at the terminal terminates with EXEC CICS RETURN TRANSID(*transid*), causing 3270 data stream to be sent by the terminal and start a new transaction. A transaction that is started in this way obtains the data in the 3270 data stream by issuing EXEC CICS RECEIVE MAP or EXEC CICS RECEIVE, depending on whether it uses BMS (Basic Mapping Support) mapping or terminal control.
- An application issues an EXEC CICS START command. The started transaction issues EXEC CICS RETRIEVE to retrieve any data that has been specified on the START command.

A transaction that has been invoked at a terminal can subsequently issue commands such as EXEC CICS CONVERSE, EXEC CICS SEND MAP and EXEC CICS RECEIVE MAP in a conversation or pseudoconversation with a terminal user.

The CICS bridge can emulate any of the these ways of starting CICS transactions. It can also emulate a terminal user sending and receiving screens of data from the transaction. These emulations are achieved by using CICS bridge vectors, which represent the EXEC CICS command being emulated and provide any data that is

needed. The data needed by a CICS transaction accompanies inbound messages, and the data needed by a CICS bridge application accompanies outbound messages.

Using CICS bridge vectors

Use vectors to represent EXEC CICS commands in request and reply messages. Vectors are represented in bridge messages by strings of numeric characters known as vector descriptors, for example 0402. Each vector descriptor is the CICS EIBFN value of the EXEC CICS command that it represents. For example, 0402 is the EIBFN value for EXEC CICS RECEIVE, and also the vector descriptor of the RECEIVE vector. Vectors are further qualified by the letters I and O to show whether they are inbound (to the bridge) or outbound (from the bridge).

An outbound message can contain a request vector or a reply vector. These descriptions do not mean that they go to the request queue or the reply queue; all outbound messages go to the reply queue. A CICS transaction uses a request vector to request data from the MQSeries bridge application that is acting as the virtual terminal. A CICS transaction uses a reply vector when it does not expect any data back. No such distinction is made for inbound vectors.

The following vectors are available. To get the CICS command that each represents, prefix the vector name with EXEC CICS.

Outbound reply vectors (no further data is required in the next inbound message):

- SEND
- SEND CONTROL
- SEND MAP
- SEND TEXT
- ISSUE ERASEAUP

Outbound request vectors (further data is required in the next inbound message):

- RECEIVE
- RECEIVE MAP
- CONVERSE

Inbound vectors:

- RECEIVE
- RECEIVE MAP
- CONVERSE
- RETRIEVE

Each of these vectors is an architected structure followed by variable length data. For details of the structures, refer to the *CICS Internet and External Interfaces Guide* for CICS V1.2, or *CICS External Interfaces Guide* for CICS V1.3.

CICS 3270 bridge message structure

The term 'CICS 3270 bridge' is used here to mean all non-DPL CICS transactions.

Inbound messages

These examples show the possible structures of CICS 3270 bridge inbound messages.

CICS 3270 bridge message structure

- Use this structure for an application that invokes a CICS transaction without any data:

```
-----  
| MQMD | MQCIH |  
-----
```

Set the field MQCIH.TransactionId to the name of the transaction you want to start. Set the other fields in the CIH to values that are appropriate for the application.

- Use this structure for inbound messages that have zero length data:

```
-----  
| MQMD | MQCIH | BRMQ structure |  
-----
```

For example, an inbound RECEIVE MAP vector can represent an action where the user has only pressed a PF key. In this case, a field within the *BRMQ structure* specifies which AID key had been used, but no data follows the *BRMQ structure*.

- Use this structure for an application that invokes a transaction that will issue an EXEC CICS command that expects data to be available:

```
-----  
| MQMD | MQCIH | BRMQ structure | data |  
-----
```

BRMQ structure represents any of the inbound vector structures RECEIVE, RECEIVE MAP, CONVERSE or RETRIEVE.

Outbound messages

Outbound messages from the bridge have one of three structures, depending on whether an error occurred. Although only a single vector is shown in each of these examples, messages can contain several concatenated vectors, except when an error occurs.

- This structure is used when bridge processing concludes normally, no errors were detected, and no data is to be returned to the bridge application:

```
-----  
| MQMD | MQCIH | BRMQ structure |  
-----
```

Even if an application abends, this is still regarded as normal completion by the bridge. The abend code issued by the application is given in the MQCIH.

- This structure is used when bridge processing concludes normally, no errors were detected, and data is to be returned to the bridge application:

```
-----  
| MQMD | MQCIH | BRMQ structure | data |  
-----
```

BRMQ structure represents any of the architected outbound reply or request vector structures.

- This structure is used when bridge processing concludes abnormally, an error having been detected by the bridge monitor:

```
-----  
| MQMD | MQCIH | CSQC* message |  
-----
```

CSQC message* represents an error message that indicates the error type. The value of field MQCIH.Format is set to MQFMT_STRING, to ensure that the

CICS 3270 bridge message structure

message can be properly converted if the final destination uses a different CCSID and encoding. The MQCIH also contains other fields that you can use to diagnose the problem.

Notes:

1. The MQMD is shown in the examples to help you visualize the overall structure of the message. This is the structure that you will see if you use the general queue browser utility of MQSeries SupportPac MA10 "MQSeries for MVS/ESA - ISPF utilities".
2. Only a single vector is shown associated with any message. In practice, a message might contain several vectors concatenated:
 - Inbound messages can contain several RECEIVE MAP vectors in anticipation of future RECEIVE MAP requests from the CICS transaction. The application needs to know the flow of control in the transaction in order to construct the message.
 - Outbound messages can contain several vectors, for example as a result of successive EXEC CICS SEND MAP commands being issued by a transaction. The CICS transaction does not have control over whether the outbound message contains a single vector or multiple vectors.
If the transaction issues a command that causes a request vector to be generated, the request vector will always be the last one in the message.

Application programming for the CICS 3270 bridge

Application programming for the CICS 3270 bridge is usually more complex than application programming for the DPL bridge for these reasons:

- The bridge emulates all of the functions of the CICS terminal API, including minimum function BMS
- The bridge application needs to be aware of the internal logic and flow of control in the CICS transaction that is being run, and it must interpret and respond to vectors that it receives in outbound messages
- If a transaction uses BMS maps, the bridge application might not have access to the copybooks created during map assembly to help interpret data in the vectors. In this case, the data must be analyzed indirectly through the use of an application data structure (ADS) descriptor.

If you are unfamiliar with the terminology used for describing application data structures, you should now refer to the section "Application data structure terminology" on page 262.

You must make sure every inbound message that is sent to the CICS 3270 Bridge includes a vector structure after the CIH, except when you start a transaction with no data.

The vector structure definitions are available in C-language header file `dfhbrmqh.h` and COBOL copybook `DFHBRMQO`. You should include these in any application that uses the bridge. These members are only provided with CICS Transaction Server on OS/390. If you wish to create your application on another platform, you must copy them to that environment.

All the vectors have a common header, but their structures differ. Details of the structures are given in the *CICS Internet and External Interfaces Guide* for CICS V1.2, or the *CICS External Interfaces Guide* for CICS V1.3. You will need to refer to these books when you are developing your bridge applications.

Programming for the CICS 3270 bridge

You are recommended also to obtain a copy of CICS SupportPac CA1E "CICS Bridge Passthrough" as an aid to analyzing the logic of your existing CICS transactions, and to help plan your MQSeries CICS 3270 Bridge applications. You can use the SupportPac to test whether CICS transactions work in a Bridge environment without having to write any application code. It also enables you to display and capture the inbound and outbound data flows, to study how messages must be structured, and what values need to be inserted into fields in the MQCIH and the vectors.

This simple example illustrates how you might write applications for the CICS 3270 Bridge; it shows how to invoke a transaction that would normally be started by entering its identifier and some command line arguments at a CICS terminal:

Example: Invoking CEMT I TASK from an application

This example shows how an application can start a transaction, in this case CEMT, that expects to receive command line arguments when it is invoked.

When the CEMT task starts, it issues EXEC CICS RECEIVE to receive any command line arguments that follow its identifier. The application that emulates the command line invocation must therefore start CEMT with a RECEIVE vector that contains appropriate values in the vector structure, and also include the command line values. The following C-language code fragment shows how the inbound message can be constructed.

```
/* #includes */
#include cmqc.h          /* MQSeries header */
#include dfhbrmqh.h     /* Vector structures */
:
:
/* #defines */
#define CMDSTRING "CEMT I TASK" /* Command string */
#define RCV_VECTOR "0402"      /* Vector descriptor */
#define INBOUND "I "           /* Inbound type */
#define VERSION "0000"         /* Vector version */
#define YES "Y "               /* YES indicator */
#define NO "N "                /* NO indicator */
:
:
/* Data declarations */
/* AID indicator value */
const char AID[ 4 ] = { 0x7d, ' ', ' ', ' ' } ;
MQMD      mqmd ;           /* Message descriptor */
MQCIH     mqcih ;         /* CICS information header */
brmq_vector_header brvh ; /* Standard vector header */
brmq_receive brrcv ;      /* RECEIVE vector structure */
MQCHAR    *MsgBuffer ;    /* Message buffer pointer */
:
:
/* allocate storage for the message buffer. Note that the
/* RECEIVE vector structure includes space for the standard
/* vector header. */

MsgBuffer = malloc(sizeof(MQCIH) + sizeof(brrcv)
                   + strlen(CMDSTRING) ) ;
:
:
/* Initialize fields in the MQMD as required */
/* Initialize fields in the MQCIH as required, including: */

strncpy(MQCIH.TransactionId, "CEMT", strlen("CEMT"));
:
:
/* Initialize fields in the RECEIVE vector header: */

brvh.brmq_vector_length = sizeof(brrcv) + strlen(CMDSTRING) ;
strncpy(brvh.brmq_vector_descriptor, RCV_VECTOR,
        strlen(RCV_VECTOR)) ;
strncpy(brvh.brmq_vector_type, INBOUND, strlen(INBOUND)) ;
```

```

strncpy(brvh.brmq_vector_version, VERSION, strlen(VERSION)) ;

/* Initialize fields in the RECEIVE vector structure:      */

strncpy(brrcv.brmq_re_transmit_send_areas, YES, strlen(YES)) ;
strncpy(brrcv.brmq_re_buffer_indicator, NO, strlen(NO)) ;
strncpy(brrcv.brmq_re_aid, AID, sizeof(brrcv.brmq_re_aid)) ;
brrcv.brmq_re_data_len = strlen(CMDSTRING) ;
:
/* Copy the MQCIH to the start of the message buffer      */

memcpy(MsgBuffer, &mqcih, sizeof(MQCIH)) ;

/* Append the RECEIVE vector to the CIH                    */

memcpy(MsgBuffer + sizeof(MQCIH), brrcv, sizeof(brrcv) ) ;

/* Overlay the standard vector header on the RECEIVE vector */

memcpy(MsgBuffer + sizeof(MQCIH), brvh, sizeof(brvh) ) ;

/* Append the command string to the vector structure      */

strncpy(MsgBuffer + sizeof(MQCIH) + sizeof(brrcv),
        CMDSTRING, strlen(CMDSTRING)) ;

/* The message buffer is now ready for the MQPUT          */
/* to the Bridge Request Queue.                          */
:

```

The outbound message that is returned to the reply queue contains a SEND reply vector with data in terminal control format – your application needs to know this when it analyzes the data that it receives.

Managing units of work for the CICS 3270 bridge

If your bridge application is running just a single transaction, you should set the value of MQCIH.UOWControl to MQCUOWC_ONLY. However, if your application is sending and receiving multiple messages, you must handle unit of work management differently for the CICS 3270 bridge

A transaction can split itself into multiple units of work by issuing EXEC CICS SYNCPOINT, but you cannot group transactions into a single unit of work. Always set the value of MQCIH.UOWControl to MQCUOWC_ONLY in the first message, even when many messages are sent by your application. You should set MQCUOWC_CONTINUE in messages supplying additional data to the transaction. If you wish to terminate the transaction, set the value of MQCIH.CancelCode to a four-character abend code.

Writing applications using CICS Basic Mapping Support

CICS Basic Mapping Support (BMS) provides a way for CICS applications to support a number of different terminal types. When the application issues EXEC CICS SEND MAP, BMS merges terminal-specific control data with the application data to produce a 3270 data stream that can be displayed at the terminal. When the application issues EXEC CICS RECEIVE MAP, application data is extracted from an inbound 3270 data stream and returned to the application.

A BMS map for a CICS application is created by assembling a set of BMS macros that define the characteristics of fields that are required for the display. One of the outputs from map assembly is a copybook that maps the display fields to an ADS. The CICS application must include the copybook in its data definitions so that it

Writing CICS BMS applications

can address the fields in the map symbolically. The application data in a SEND MAP, and expected by a RECEIVE MAP, is mapped directly to the ADS in the copybook.

When the transaction runs under the CICS bridge, EXEC CICS SEND MAP and EXEC CICS RECEIVE MAP commands cause SEND MAP and RECEIVE MAP vectors to be generated in outbound messages. Instead of a 3270 data stream, these vectors contain ADSs equivalent to those used by the CICS application to address fields in the map.

The format of the ADS is unique for each map. It is described by a copybook created as part of map generation. Without this copybook it is difficult to interpret the data. Usually MQSeries applications include the BMS copybooks so that they can create RECEIVE MAP data, and interpret SEND MAP data. However, you can write an application without the specific BMS copybooks. The format of the data is described by a structure known as the ADS Descriptor (ADSD). The ADSD is added to the end of a SEND MAP vector, and it describes the format of the ADS in the vector. The ADSD contents include the names, positions and lengths of the fields in the ADS. An ADSD can also be sent on a RECEIVE MAP Request. You can use this in conversational applications to tell the MQSeries application the structure of the ADS requested by the CICS application. The MQSeries application can then build a RECEIVE MAP vector with this ADS, and send it as a new request.

As an application programmer, you can choose whether you want to interpret vector data in messages using the ADS, the ADSD, or the ADSDL. In order to interpret the ADS directly, you need to include the copybook from map assembly in your MQSeries Bridge application. If you want to do this, but you do not have access to the copybook or map, you should recreate the map with the CICS utility DFHBMSP – this requires CICS Transaction Server 1.2 or later.

If you want to interpret the ADS indirectly through the ADSD or ADSDL, for example if you are creating a generic application that will handle all maps, you do not need to include the copybook in your bridge application. Instead you need to send control information to the bridge that tells it to include the ADSD or ADSDL in outbound SEND MAP and RECEIVE MAP request vectors as required.

If your application must run in the distributed environment, you need to include an ADSDL in outbound SEND MAP vectors. MQSeries can then perform data conversion on the outbound message.

You can specify any of the following options by setting appropriate values in field MQCIH.ADSDDescriptor in inbound messages:

- To include an ADSD (short form of the application data structure descriptor) with the SEND MAP vector, set:

```
MQCIH.ADSDDescriptor = MQCADSD_SEND
```

If you specify this alone, you will also get the short form of the ADS (application data structure) included in the SEND MAP vector.

- To include an ADSD with the RECEIVE MAP vector, set:

```
MQCIH.ADSDDescriptor = MQCADSD_RECV
```

The ADS is never present in an outbound RECEIVE MAP request vector.

- To include an ADSDL (long form of the application data structure descriptor) in the SEND MAP or RECEIVE MAP vector, set:

```
MQCIH.ADSDescriptor = MQCADSD_MSGFORMAT
```

If you specify this, you will also get the long form of the ADS included in the SEND MAP vector.

- To not include an ADS descriptor in the SEND MAP or RECEIVE MAP vector set:

```
MQCIH.ADSDescriptor = MQCADSD_NONE
```

This is the default. If you specify this, you will get the short form of the ADS included in the SEND MAP vector.

You can add MQCADSD_* values together, so do this to cause the long form of the application data structure descriptor to be included in both SEND MAP and RECEIVE MAP vectors:

```
MQCIH.ADSDescriptor = MQCADSD_SEND + MQCADSD_RECV + MQCADSD_MSGFORMAT
```

The SEND MAP vector would also in this case include an ADS in long form.

Interpreting outbound SEND MAP and RECEIVE MAP vectors

Your bridge application should contain logic to interpret outbound SEND MAP and RECEIVE MAP request vectors, and it should build and send an inbound RECEIVE MAP vector in response to the corresponding outbound RECEIVE MAP request.

SEND MAP vectors:

An outbound SEND MAP vector can contain an application data structure (ADS) and an application data structure descriptor in short form (ADSD) or long form (ADSDL).

To interpret a SEND MAP vector, do this (assuming that the message contains both an ADS and an ADSD or ADSDL):

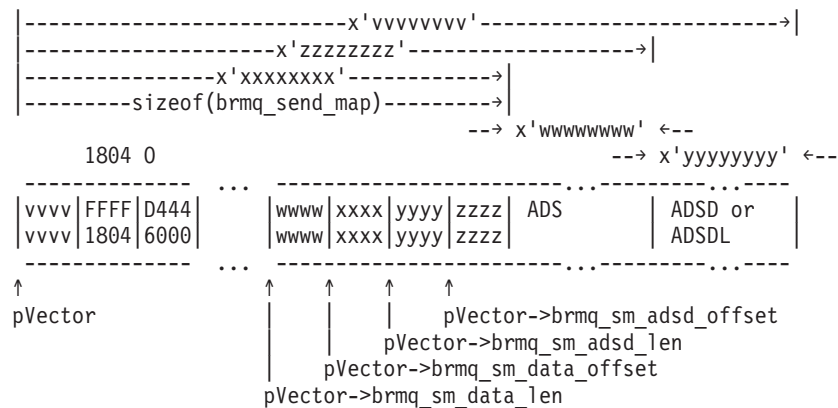
1. GET the message containing the SEND MAP vector from the bridge reply queue into a message buffer.
2. Locate the start of the outbound SEND MAP vector in the message buffer. This is appended to the CIH, and so is at an offset equal to the length of the CIH from the start of the message buffer. You can use the following code fragment as a model.

```
/* #includes */
#include cmqc.h          /* MQSeries header */
#include dfhbrmqh.h     /* Vector structures */
:
:
/* #defines */
:
:
MQCHAR      * MsgBuffer ; /* Message buffer pointer */
brmq_send_map * pVector  ; /* Vector pointer */
:
:
/* Get message from reply queue */
:
:
/* Set the vector pointer to the start of the vector */
pVector = MsgBuffer + ((MQCIH *) MsgBuffer)->StrucLength ;
```

3. Identify the starting addresses of the application data structure (ADS) and the application data structure descriptor (ADSD or ADSDL) from the SEND MAP vector.

SEND MAP vectors

This diagram shows the structure of an outbound SEND MAP vector (assuming that you have set a pointer called *pVector* to address the start of the *brmq_send_map* vector, as in the code fragment above).



Values in the diagram shown like this:

```
ABCD
1234
```

show hexadecimal values as you would see them in an ISPF editor with 'hex on'. This is equivalent to the hexadecimal value `x'A1B2C3D4'`.

Fields *pVector->brmq_sm_data_offset* and *pVector->brmq_sm_data_len* give the offset and length, respectively, of the ADS, and fields *pVector->brmq_sm_adsd_offset* and *pVector->brmq_sm_adsd_len* give the offset and length, respectively, of the ADSD or ADSDL.

Fields *brmq_sm_adsd_offset* and *brmq_sm_adsd_len* will both be set to zero if no ADSD or ADSDL is included in the message.

4. Identify the fields in the ADSD or ADSDL.

The ADSD and ADSDL are both mapped to structures that are defined in header file *dfhbrarh.h*, which is distributed in library `<hlq>.SDFHC370` for CICS Transaction Server for OS/390 Version 1.2 or later. You can examine the structure definitions there to see how the fields are laid out. The fields of the ADSD are also described in the *CICS Internet and External Interfaces Guide* for CICS V1.2, or the *CICS External Interfaces Guide* for CICS V1.3..

If your bridge application is to be compiled on a workstation, you will need to copy file *dfhbrarh.h* to that environment.

Both the ADSD and the ADSDL are represented by two types of structure. The first structure is the descriptor, which occurs only once at the start of the ADSD or ADSDL. These types are defined:

ads_descriptor

Descriptor for the ADSD (short form)

ads_long_descriptor

Descriptor for the ADSDL (long form)

The second structure is the field descriptor, which is repeated once for each field in the map. These types are defined:

ads_field_descriptor

Field descriptor for the ADSD (short form)

SEND MAP vectors

```

:
pADSDL_FD += sizeof(ads_long_field_descriptor) ;
}
while (pADSDL_FD < pADSDL_D->adsdl_length ) ;
}

else          /* Short form          */
{
    pADSD_D = pVector->brmq_sm_adsd_offset; /* Short form */
    pADSD_FD = pADSD_D + sizeof(ads_descriptor) ;
/* Enter a loop where we process all field descriptors */
/* in the ADSD sequentially */

    do
    {
/* Perform some processing */

:
    pADSD_FD += sizeof(ads_field_descriptor) ;
    }
    while (pADSD_FD < pADSD_D->adsd_length ) ;
}
:

```

5. Identify the fields in the ADS.

The ADS itself is mapped to a structure that is generated when you assemble your map. If you include a keyword=parameter value of *DSECT=ADSDL* in your mapset definition macro, you will get the long form of the ADS. The output from map assembly is a union of two structures: an input structure and an output structure. This example shows part of such a union (only the first field definition is shown for each structure, and the comments have been added following map assembly).

```

union
{
struct {
    char    dfhms1[12]; /* 12 reserved bytes */
    int     dfhms2; /* Offset to next field */
    int     tranidl; /* Data length of this field */
    int     tranidf; /* Flag or attribute value */
    int     dfhms3[7]; /* Extended attributes array */
    char    tranidi[4]; /* Data value of field */
    ...
} bmstmpli; /* Input structure */

struct {
    char    dfhms56[12]; /* 12 reserved bytes */
    int     dfhms57; /* Offset to next field */
    int     dfhms58; /* Data length of this field */
    int     tranida; /* Flag or attribute value */
    int     tranidc; /* Extended attribute */
    int     tranidp; /* Extended attribute */
    int     tranidh; /* Extended attribute */
    int     tranidv; /* Extended attribute */
    int     tranidu; /* Extended attribute */
    int     tranidm; /* Extended attribute */
    int     tranidt; /* Extended attribute */
    char    tranido[4]; /* Data value of field */
    ...
} bmstmplo; /* Output structure */

} bmstmp1; /* Union */

```

The two structures are functionally identical, except that the input structure includes the extended attribute values in a seven-element array, and the output structure provides individually named fields.

SEND MAP vectors

You can use the following code fragment to set pointers to the start of the ADS. The structure names shown in the example DSECT above are used for illustration. Two pointers are set, the first to address inbound data and the second to address outbound data. It is assumed that *pVector* already addresses the start of the *brmq_send_map* vector.

```
/* #includes */
#include cmqc.h /* MQSeries header */
#include dfhbrmqh.h /* Vector structures */
#include dfhbrarh.h .. /* ADSD structures */
#include mydsect.h /* DSECT from map assembly */
:
:
bmstmpli * pADSI ; /* Pointer to the inbound ADS */
bmstmplo * pADSO ; /* Pointer to the outbound ADS */
bmstmpli * pADSI_An ; /* Inbound ADS Anchor */
bmstmplo * pADSO_An ; /* Outbound ADS Anchor */
:
/* We are dealing with an outbound vector, so we will */
/* initialize the outbound pointer to address the ADS */

pADSO = pVector->brmq_sm_adsd_offset ;

/* Save initial value as anchor */

pADSO_An = pADSO ;

/* Move to the start of the first field */

pADSO += pADSDL_FD->adddl_field_offset ;

/* Enter a loop where we process all fields in the ADS */
/* sequentially. It is assumed that the value of pADSDL_FD */
/* is being augmented to the next field descriptor in the */
/* ADSDL with every loop. A model for this is shown in a code*/
/* fragment above. Note that adddl_field_offset contains */
/* the absolute offset of the field from the start of the */
/* ADS. */

do
{
/* Perform some processing */

:

/* Add offset of next field to ADS Anchor value */
/* to address the next field */

pADSO = pADSO_An + pADSDL_FD->adddl_field_offset ;
}
while (pADSDL_FD < pADSDL_D->adddl_length) ;
:
```

The general structures of the long and short forms of the ADS are given in the IBM Redbook *CICS Transaction Server for OS/390 Version 1 Release 3: Web Support and 3270 Bridge*.

RECEIVE MAP vectors:

A RECEIVE MAP request is a request for the client to provide a RECEIVE MAP on the next input message. Unlike a SEND MAP vector, an outbound RECEIVE MAP request vector never contains an ADS. It contains an ADSD or ADSDL that describes the ADS data that it requires in the next inbound RECEIVE MAP vector, provided that MQCADSD_RECV has been specified in MQCIH.ADSDDescriptor.

RECEIVE MAP vectors

The RECEIVE MAP vector structure differs from that of the SEND MAP vector. The main difference is that there are no fields giving the offset and length of the ADS.

Do this to interpret a RECEIVE MAP vector (assuming that the message contains an ADSD or ADSDL):

1. GET the message containing the RECEIVE MAP request vector from the bridge reply queue into a message buffer.
2. Locate the start of the outbound RECEIVE MAP vector in the message buffer. This is appended to the CIH and so is at an offset equal to the length of the CIH from the start of the message buffer. You can use this code fragment as a model.

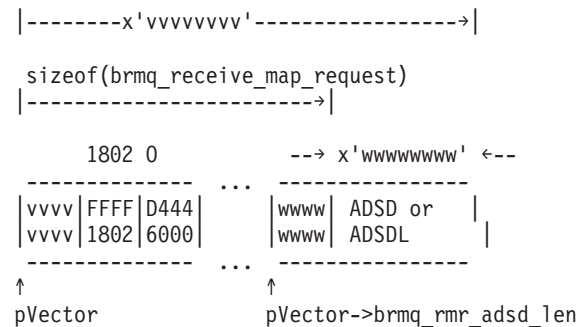
```

/* #includes */
#include cmqc.h /* MQSeries header */
#include dfhbrmqh.h /* Vector structures */
:
/* #defines */
:
MQCHAR * MsgBuffer ; /* Message buffer pointer */
brmq_receive_map_request * pVector ; /* Vector pointer */
:
/* Get message from reply queue */
:
/* Set the vector pointer to the start of the vector */
pVector = MsgBuffer + ((MQCIH *) MsgBuffer)->StrucLength ;
:

```

3. Identify the starting address ADSD or ADSDL from the RECEIVE MAP vector.

This following diagram shows the structure of an outbound RECEIVE MAP request vector (the diagram assumes that you have set a pointer called *pVector* to address the start of the *brmq_receive_map_request* vector, as in the code fragment above).



Values in the diagram shown like this:

```

ABCD
1234

```

show hexadecimal values as you would see them in an ISPF editor with 'hex on'. This is equivalent to the hexadecimal value x'A1B2C3D4'.

Field *pVector->brmq_rmr_adsd_len* gives the length of the ADSD or ADSDL. No offset is given since the ADSDL is appended directly to the *brmq_receive_map_request* vector.

4. Identify the fields in the ADSD or ADSDL. To do this, proceed in general as for the SEND MAP vector described above. Use the following code fragment, however, to set pointers to the start of the ADSD or ADSDL.

```

      :
if (mqcih.ADSDescriptor && MQCADSD_MSGFORMAT)
{
    pADSDL_D = pVector + sizeof(brmq_receive_map_request) ;
      :
}

else                                  /* Short form          */
{
    pADSD_D = pVector + sizeof(brmq_receive_map_request) ;
      :
}
      :

```

The ADSD or ADSDL has exactly the same structure in the RECEIVE MAP vector as in the SEND MAP vector, so once you have identified its start address you can proceed as described for the SEND MAP vector.

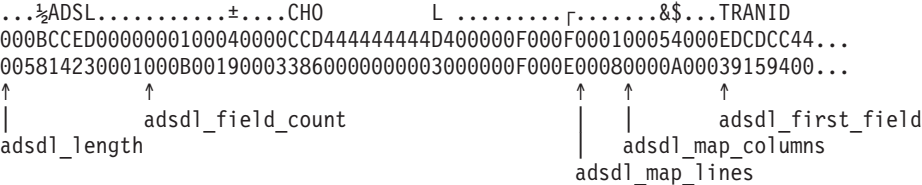
Example of an ADSDL and an ADS:

An example showing parts of an ADSDL and an ADS is given here. For full details of all the fields, see the references already cited. Values in the diagrams shown like this:

ABCD
1234

show hexadecimal values as you would see them in an ISPF editor with 'hex on'. This is equivalent to the hexadecimal value x'A1B2C3D4'.

This diagram shows the start of the ADSDL (even though the eyecatcher shows ADSL):



The fields named in this example show the following:

- adsdl_length**
This shows that this ADSDL is 0x05B8 bytes long
- adsdl_field_count**
There are 0x1B (27) named fields in the ADS
- adsdl_map_lines**
The map has 0x18 (24) lines
- adsdl_map_columns**
The map has 0x50 (80) columns
- adsdl_first_field**
This is the start of the first field description in the ADSDL.

The next diagram shows the ADSDL first field descriptor and part of the next field descriptor.

A number of attribute and extended attribute values for the field, not identified here, appear between the fullword giving the offset to the next field and the field value itself.

Transactions with start data

An application that starts a transaction that will issue an EXEC CICS RETRIEVE ... QUEUE(data-area) to retrieve its start data would send a message to the bridge with a RETRIEVE vector structure, which is defined in C as *brmq_retrieve*. The structure contains a character field of length eight bytes in which the application program must specify the name of the temporary storage queue that contains the data to be retrieved. A message containing a RETRIEVE vector is always the first in an exchange representing a conversation or pseudoconversation.

Programming for the distributed environment

CICS DPL programs and transactions can be driven through the CICS bridge when the client application resides on a workstation.

The main consideration when programming for the distributed environment is data conversion between the different encoding schemes and CCSID values of the workstation and OS/390. Conversion is carried out by two different routines, one for the MQCIH structure and another for the vector.

You can ensure that conversion of the MQCIH is achieved by specifying MQFMT_CICS in the MQMD.Format field. Vector conversion, however, requires a little more consideration.

CICS transactions in the distributed environment

Conversion is only supported by the CICS bridge for the outbound SEND MAP and RECEIVE MAP request vectors, and for the inbound RECEIVE MAP vector.

To achieve conversion of the SEND MAP and RECEIVE MAP vectors, do this:

- Make sure that you assemble your maps specifying *DSECT=ADSDL* in your DFHMSD macro. Your map must be assembled under CICS Transaction Server for OS/390 Version 1.2 or greater for the ADSD or ADSDL to be made available. If you do not have the original mapset definition, recreate the map using the CICS DFHBMSUP utility.
- Specify a value of MQCADSD_SEND+MQCADSD_MSGFORMAT in field MQCIH.ADSDDescriptor. If you are using an ADSD or ADSDL to build your RECEIVE MAP ADS, you must also add in the value MQCADSD_RECV for this field.
- Specify a value of CSQCBDCI in field MQCIH.Format on every inbound message.

If you want to use vectors other than SEND MAP and RECEIVE MAP to drive transactions in the distributed environment, you must either write your own data conversion routines or create and interpret the data streams in the format required by OS/390.

The MQCIH.Format is always set to CSQCBDCO in outbound messages. If you want to specify another format type for outbound conversion, you must intercept the message by writing to a local reply queue. Change the value of MQCIH.Format to specify your own routine before sending it on to the remote environment.

Distributed programming

No support is provided for conversion between workstation and mainframe formats of vectors other than SEND MAP (outbound) and RECEIVE MAP (both inbound and outbound).

CICS DPL programs in the distributed environment

If you are driving a DPL program that neither receives nor returns COMMAREA data, or if the COMMAREA data is purely character data, you can achieve data conversion by specifying MQFMT_STRING in field MQCIH.Format. If your COMMAREA data is not purely character data, you must write your own conversion routine.

From 3270 legacy to 3270 bridge - an example

This Section illustrates the differences in the data flows that take place when a CICS 3270 transaction interacts with a 3270 terminal, and a CICS bridge application.

In this example, the transaction has an identifier of BAAA. It uses BMS maps, which allow the transaction to be adapted easily to run under the CICS bridge.

In the legacy environment, the transaction is started by entering its name at the CICS 3270 terminal and pressing ENTER. Logic in the transaction causes it to issue EXEC CICS SEND MAP the first time it is invoked in a pseudoconversation, and then to terminate by issuing EXEC CICS RETURN TRANSID(BAAA).

The user enters values into fields in the map that is displayed at the terminal, and then presses an AID key. Logic in the transaction the second time it is invoked causes it to issue EXEC CICS RECEIVE MAP to receive the map. It updates certain fields in the map by changing values in its own application data structure, and then issues EXEC CICS SEND MAP to re-display the map at the user's terminal.

The user can then update fields in the re-displayed map, and start the RECEIVE MAP - SEND MAP cycle again. The logic can be illustrated like this (where EC represent EXEC CICS):

Terminal user		3270 Transaction
BAAA <ENTER>	----->	<Initial start>
	<-----	<business logic>
		EC SEND MAP FROM(ads)
		EC RETURN TRANSID(BAAA)
Update fields	----->	EC RECEIVE MAP INTO(ads)
<ENTER>	<-----	<business logic>
		EC SEND MAP
		EC RETURN TRANSID(BAAA)
Update fields	----->	EC RECEIVE MAP
<ENTER>		
:		

When the transaction runs in the bridge environment, the physical terminal is replaced by an application. The logic of the 3270 transaction is unchanged, and the application data that it receives is the same, but the data that flows, and the means by which it is transmitted, are different. Instead of a 3270 data stream, an MQSeries message is used that contains an MQCIH structure (a CICS Information Header), a bridge vector structure, and optionally a representation of the application data structure.

Inclusion of these objects in the message depends on the direction in which the message flows (inbound to the bridge or outbound from the bridge), the sequence of the message in the exchange, and whether an application data structure descriptor has been requested by setting the appropriate value in a field in the MQCIH.

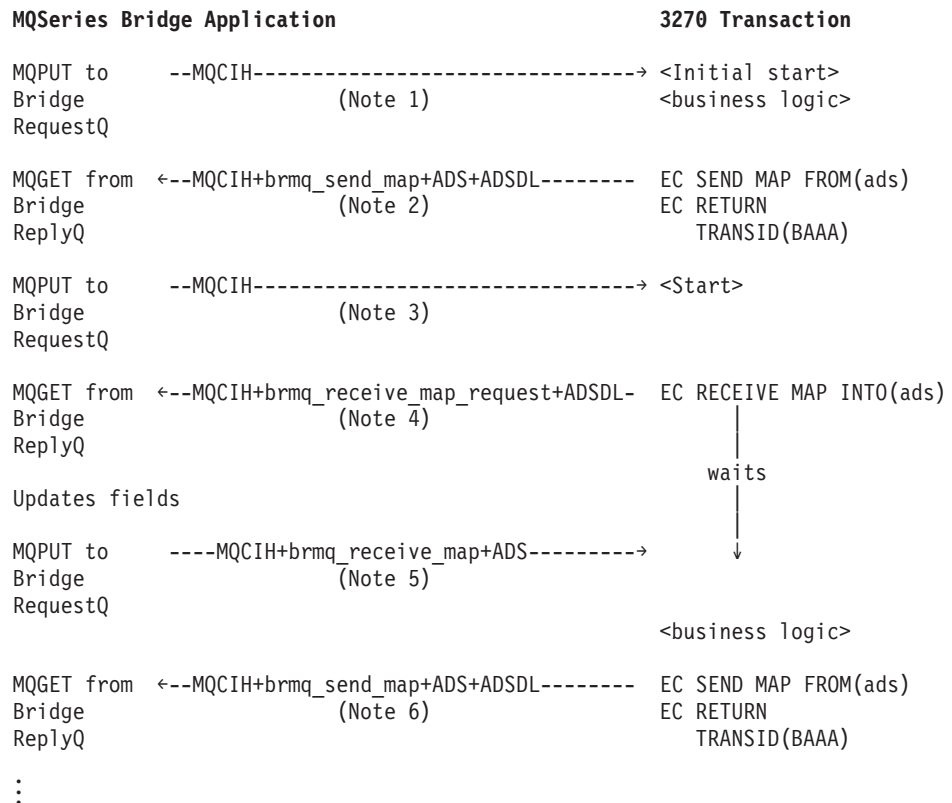
The section “Exact emulation - no optimization” shows the flows that take place when the above scheme is emulated exactly. There is scope for optimization by including more than one vector in inbound messages, as shown in the section “Improved emulation, with optimization” on page 252.

It is assumed that MQCIH.ADSDescriptor is set to MQCADSD_SEND + MQCADSD_RECV + MQCADSD_MSGFORMAT

so application data structure descriptors in long form are appended to both outbound and inbound application data structures during the exchange of messages.

For clarity, the details of messaging are omitted here. For a description of the queuing model used by the CICS bridge, see the *MQSeries for OS/390 Concepts and Planning Guide*.

Exact emulation - no optimization



Notes:

1. The initial flow from the application contains just an MQCIH. The MQCIH includes control information specifying which transaction is to be started.
2. The return flow from the 3270 transaction contains an MQCIH, which has important information including a facility token to be used for all subsequent flows, and diagnostic information if an error has occurred. It also contains a

Emulation without optimization

SEND MAP vector structure containing control information relating to the map itself, and data that represents the map. If the initiating application has requested it, an application data structure descriptor is also be included.

3. The bridge application sends a message back containing only an MQCIH. This contains control information to start the transaction once more.
4. The 3270 transaction issues EC RECEIVE MAP, as it did in the legacy environment. However, in the bridge environment the map data is not immediately available. The call is converted to a message containing an outbound RECEIVE MAP request vector. The application data structure descriptor is also included in the message. In this example, the transaction waits while the message is turned around by the bridge application. The model here is a little different to that in the legacy environment. Note, however, that the bridge architecture allows messages to contain more than one vector, so a number of requests could be satisfied by a single inbound message.
5. Having updated any fields it needs to in the application data structure, the bridge application sends an inbound RECEIVE MAP reply vector to satisfy the outbound request.
6. The 3270 transaction issues EC SEND MAP, which converts to a SEND MAP vector, and the cycle repeats.

If you examine the flows that are captured when such a transaction is run using the Passthrough tool (CICS SupportPac CA1E "CICS 3270 Bridge Passthrough"), you will be able to identify the structures with the help of the available online documentation.

Improved emulation, with optimization

MQSeries Bridge Application	3270 Transaction
MQPUT to Bridge RequestQ	--MQCIH-----> <Initial start> <business logic>
MQGET from Bridge ReplyQ	<--MQCIH+brmq_send_map+ADS+ADSDL----- EC SEND MAP FROM(ads) EC RETURN TRANSID(BAAA)
MQPUT to Bridge RequestQ	----MQCIH+brmq_receive_map+ADS-----> <Start> EC RECEIVE MAP INTO(ads) <business logic>
MQGET from Bridge ReplyQ	<--MQCIH+brmq_send_map+ADS+ADSDL----- EC SEND MAP FROM(ads) EC RETURN TRANSID(BAAA)
:	

If you compare this sequence with the un-optimized flows shown at the section "Exact emulation - no optimization" on page 251, you can see that the CICS transaction does not have to send a RECEIVE MAP request vector, because the inbound RECEIVE MAP vector has already anticipated the requirement and the inbound map is already available to the transaction.

Setting fields in the MQMD and MQCIH structures

Your CICS bridge application must set a number of fields in the MQMD and the MQCIH in order to use the bridge successfully. You need to consider the open options and the put message options that you use for the bridge request queue if the bridge monitor is started with authorization levels of VERIFY_UOW or VERIFY_ALL.

Setting the MQMD fields

Fields in the MQMD that can affect the operation of the CICS bridge need to be initialized in your application program:

MQMD.Format

Set the value to MQCICS if you include an MQCIH in the message you send to the bridge request queue, otherwise set it to MQFMT_NONE.

MQMD.MsgId

For MQPUTs to the request queue, set the value to MQMI_NONE in every message.

For MQGETs from the reply queue, use the value of MQMD.Msgid that MQSeries set in your message descriptor when you put your first message to the request queue.

MQMD.CorrelId

For MQPUTs to the request queue, set the value to MQCI_NEW_SESSION in the first or only message in a unit of work. On subsequent messages in the unit of work, set the value to MQMD.Msgid that MQSeries set in your message descriptor when you put your first message to the request queue.

For MQGETs from the reply queue, use the value of MQMD.Msgid that MQSeries set in your message descriptor when you put your most recent message to the request queue, or specify MQCI_NONE.

MQMD.ReplyToQ

Set the value to the name of the queue where you want the bridge to send reply messages.

MQMD.UserIdentifier

This field is only used when the bridge monitor is running with authorization levels of IDENTIFY, VERIFY_UOW or VERIFY_ALL. If you use any of these, set the value to the user ID that will be checked for access to the CICS DPL program.

You must add the value MQOO_SET_IDENTITY_CONTEXT to the open options when you open the bridge request queue, and also add the value MQPMO_SET_IDENTITY_CONTEXT to the put message options when you send a message to the queue.

If you use this field with one of the VERIFY_* options, the MQCIH.Authenticator field must also be initialized. Set it to the value of the password or passticket associated with the User ID.

Setting the MQCIH fields

The MQCIH contains both input and output fields. The input fields that you need to initialize in your application program when you use the CICS bridge are as follows.

MQCIH.Format

Set the value to "CSQCBDCI". This informs the bridge that any data

Setting MQCIH fields

following the MQCIH is inbound to the bridge, and may need to be converted. The bridge will set the value of MQCIH.Format in the outbound message, which is returned to the reply queue, to "CSQCBDCO".

MQCIH.UOWControl

This field controls the unit of work processing performed by the bridge. Allowed values are described in *MQSeries Application Programming Reference*. Select the required value from the ones that are available, noting that most are valid for the DPL bridge only, and not for the 3270 bridge. See also "Managing units of work for the CICS DPL bridge" on page 234 and "Managing units of work for the CICS 3270 bridge" on page 239.

MQCIH.GetWaitInterval

If you allow this to default, the bridge task GET WAIT interval for messages within a unit of work will be the value specified on the WAIT parameter when the bridge monitor was started. If you also allow the WAIT parameter to default, the GET WAIT interval will be unlimited.

MQCIH.LinkType

You must specify MQCLT_PROGRAM if you are using the DPL bridge, or MQCLT_TRANSACTION if you are using the 3270 bridge.

MQCIH.OutputDataLength

This applies only to the DPL bridge. See the *MQSeries Application Programming Reference* for details.

MQCIH.FacilityKeepTime

This applies to the 3270 bridge only. If you are sending more than a single message in a pseudoconversation, you must set this to a non zero value in the first message for the bridge to return a facility token. Successive transactions in a pseudoconversation can use the same facility token once it has been set in this way, ensuring that associated terminal areas - for example the TCTUA - are preserved for the period of the pseudoconversation.

Note also that the value of MQCIH.Facility must be set to MQCFAC_NONE in the first message in order to receive a facility token from the bridge.

MQCIH.ADSDDescriptor

This field applies only to the 3270 bridge for transactions that use BMS SEND MAP and RECEIVE MAP calls. If this is the case, and the application that is sending bridge request messages is on a workstation, you must set this value to MQCADSD_SEND + MQCADSD_RECV + MQCADSD_MSGFORMAT. This will ensure that the vectors in the bridge request and reply messages are correctly converted between the different CCSID and encoding schemes of the workstation and the mainframe.

MQCIH.ConversationalTask

This applies to the 3270 bridge only. See the *MQSeries Application Programming Reference* for details.

MQCIH.Facility

This applies to the 3270 bridge only. Ensure that this is set to MQCFAC_NONE in the first message in a pseudoconversation, and also ensure that the MQCIH.FacilityKeepTime is set to a non zero value. The bridge will return a facility token in the first message, and this value must be used in all subsequent inbound messages in the pseudoconversation.

MQCIH.Authenticator

This field only applies if you are using an authorization level of

Setting MQCIH fields

VERIFY_UOW or VERIFY_ALL. VERIFY_UOW is available to both the 3270 bridge and the DPL bridge, but VERIFY_ALL is available only to the DPL bridge.

Set the value to the password or passticket that is to be associated with the user ID in the MQMD.UserIdentifier field. Together, the values will be used by the external security manager to determine whether the user is authorized to start the 3270 transaction or link to the DPL program.

MQCIH.ReplyToFormat

Set this to MQFMT_NONE (the default value) if your application and the bridge are running in the same CCSID and encoding environment. Otherwise, set the value to MQFMT_CICS.

MQCIH.TransactionId

If you are using the DPL bridge, use the default value (four spaces) unless you want the bridge to run the DPL program under a transaction code other than the default value of CKBP. See the *MQSeries Application Programming Reference* for details of how to specify a different transaction code.

If you are using the 3270 bridge, this is the transaction identifier of the user transaction that run as the bridge task. The first message must specify the first transaction to be started. You should set this field in subsequent messages to the value of MQCIH.NextTransactionId that is returned in the preceding reply message.

MQCIH.FacilityLike

You can either use the default value of four spaces, or you can specify the name of an installed terminal. You can find the names of installed terminals by entering the CICS command CEMT I TASK or a CEMT I TERM at a CICS terminal.

MQCIH.AttentionId

Set this field to a value representing the AID key expected by the transaction, if any, otherwise accept the default value of four spaces, which will appear to the CICS transaction as the ENTER AID key.

The inbound RECEIVE, RECEIVE MAP and CONVERSE vectors also have fields in which you can specify AID values. The value in the MQCIH is the value to which EIBAID is set to when the application is started. It represents the PF key used to start the transaction. The value in the inbound vector is the value used when the data is entered. For example, this would be the value of EIBAID after the EXEC CICS RECEIVE MAP instruction.

Notes:

1. For conversational transactions there would be separate values for the initial MQCIH value and the value on the vector.
2. If the MQSeries application is sending a message in response to a request vector, the value in the MQCIH is ignored.
3. In the case of pseudoconversational transactions, the same value should be entered in the MQCIH and the first vector.

The first byte of this field is set to the value in the CICS copybook DFHAID.

MQCIH.StartCode

Change the value of this field from the default value of MQCSC_NONE only if you are starting a 3270 transaction. The value you use depends on

Setting MQCIH fields

the nature of the transaction. Use a value of MQCSC_START if the transaction is started by an EXEC CICS START command without data, and it does not issue EXEC CICS RETRIEVE. Use a value of MQCSC_STARTDATA if the transaction is started by an EXEC CICS START command with data, and it issues EXEC CICS RETRIEVE. Otherwise, use a value of MQCSC_TERMINPUT.

Setting the open options and put message options for the bridge request queue

If you start the bridge monitor with authorization levels of IDENTIFY, VERIFY_UOW or VERIFY_ALL, you must open the bridge request queue with open options that include MQOO_SET_IDENTITY_CONTEXT. You must also include a value of MQPMO_SET_IDENTITY_CONTEXT in your put message options.

Managing MsgId and CorrelId in a unit of work

The following diagrams summarize the values that you should use and expect in key fields in the MQMD and MQCIH in typical CICS DPL bridge and CICS 3270 bridge applications.

In Figure 16 on page 257, running more than one user program using the DPL bridge, the *MsgId* of the request message is set by the queue manager (to M1), and subsequently copied to the *CorrelId*. Figure 17 on page 258 shows a conversational 3270 transaction.

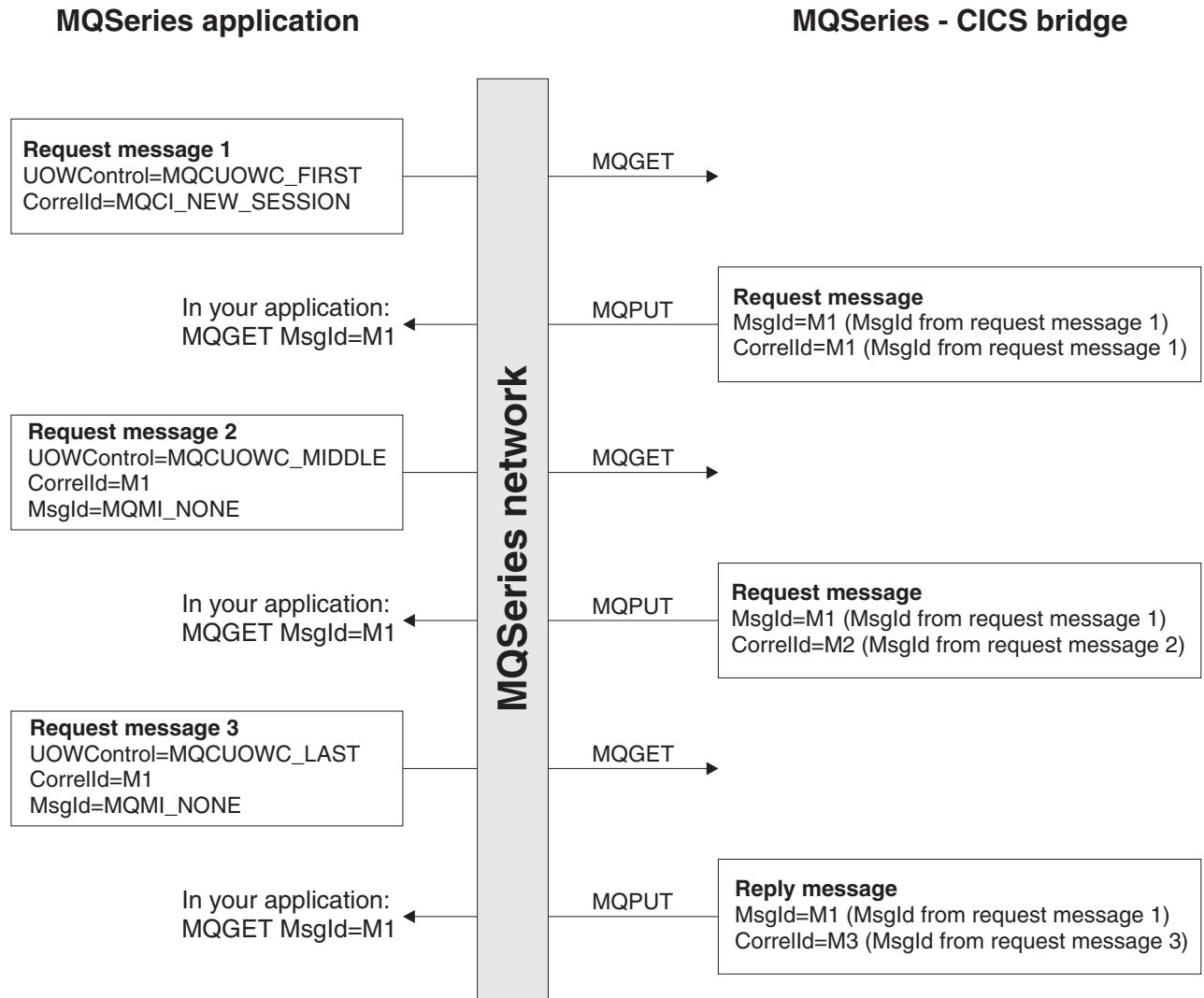


Figure 16. Setting of key fields for many CICS user programs in a unit of work

1

Bridge error handling

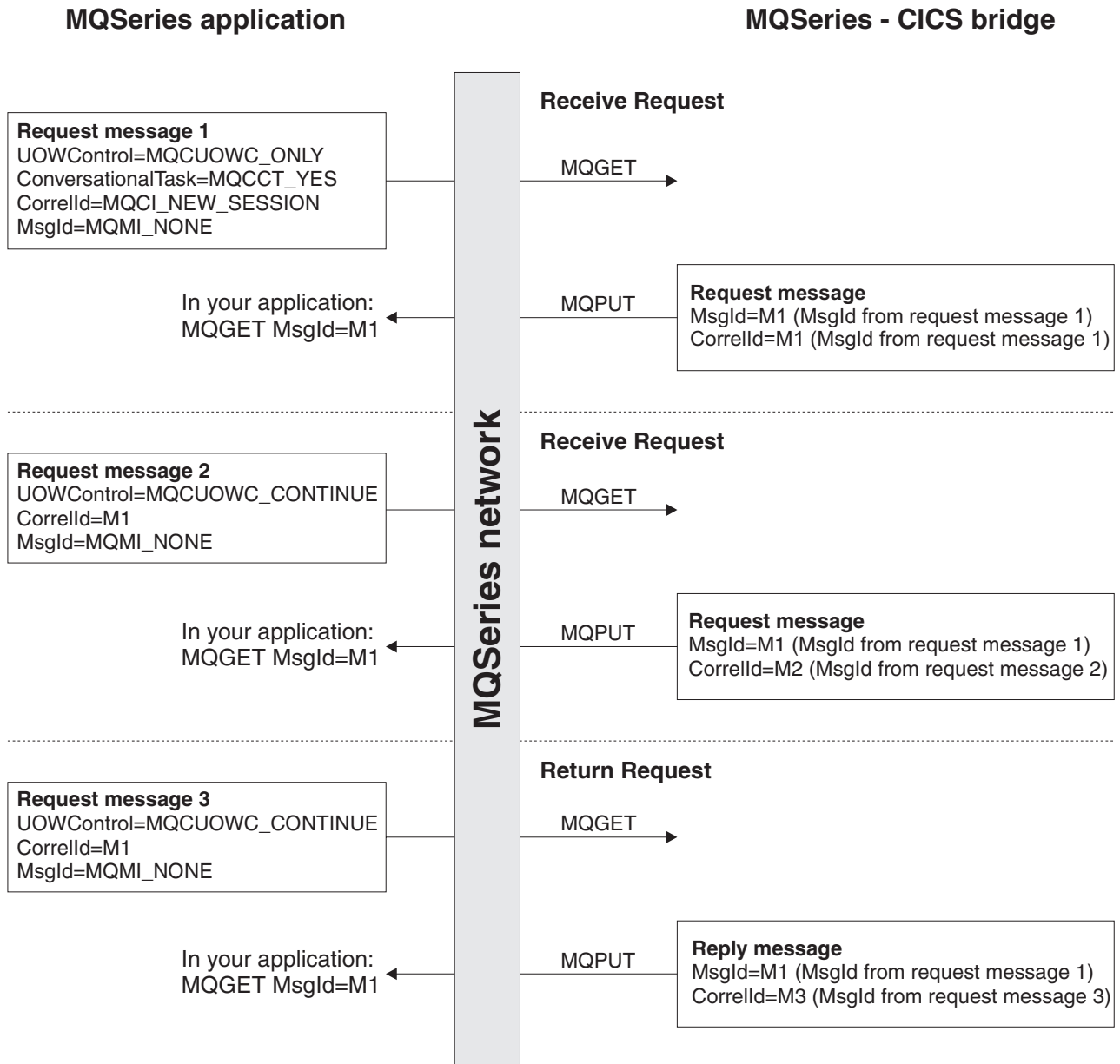


Figure 17. Setting of key fields: MQSeries - conversational 3270 transaction

Error handling by the CICS bridge

Errors detected by the CICS bridge cause the bridge to:

- Back out the unit of work
- Copy request messages to the dead-letter queue
- Send an error reply message back to the client if a reply-to queue is available
- Write a CSQC7nn message to the CICS CSMQ transient data queue or issue a transaction abend. Where it is possible to put a message on the reply-to queue, the message will contain this abend code

Any further request messages in the same unit of work are removed from the request queue and copied to the dead-letter queue, either during error processing for this unit of work or at the next initialization of the monitor. No further error messages are generated.

Bridge error handling

Unexpected messages are removed from the request queue during monitor initialization and put on the dead-letter queue, passing all context. No error reply messages are generated.

If sending a reply message fails, the CICS Bridge puts the reply on the dead-letter queue, passing identity context from the CICS bridge request queue. A unit of work is not backed out if the reply message is successfully put on the dead-letter queue. Failure to put a reply message on the dead-letter queue is treated as a request error, and the unit of work is backed out.

If the CICS bridge fails to put a request message on the dead-letter queue, the CICS bridge task abends and leaves the CICS bridge monitor to process the error. If the monitor fails to copy the request to the dead-letter queue, the monitor abends.

Failure to put an error reply is ignored by the CICS bridge. The request message has already been copied to the dead-letter queue and the unit of work has been backed out by MQSeries.

Figure 18 shows what happens when an error occurs in a unit of work.

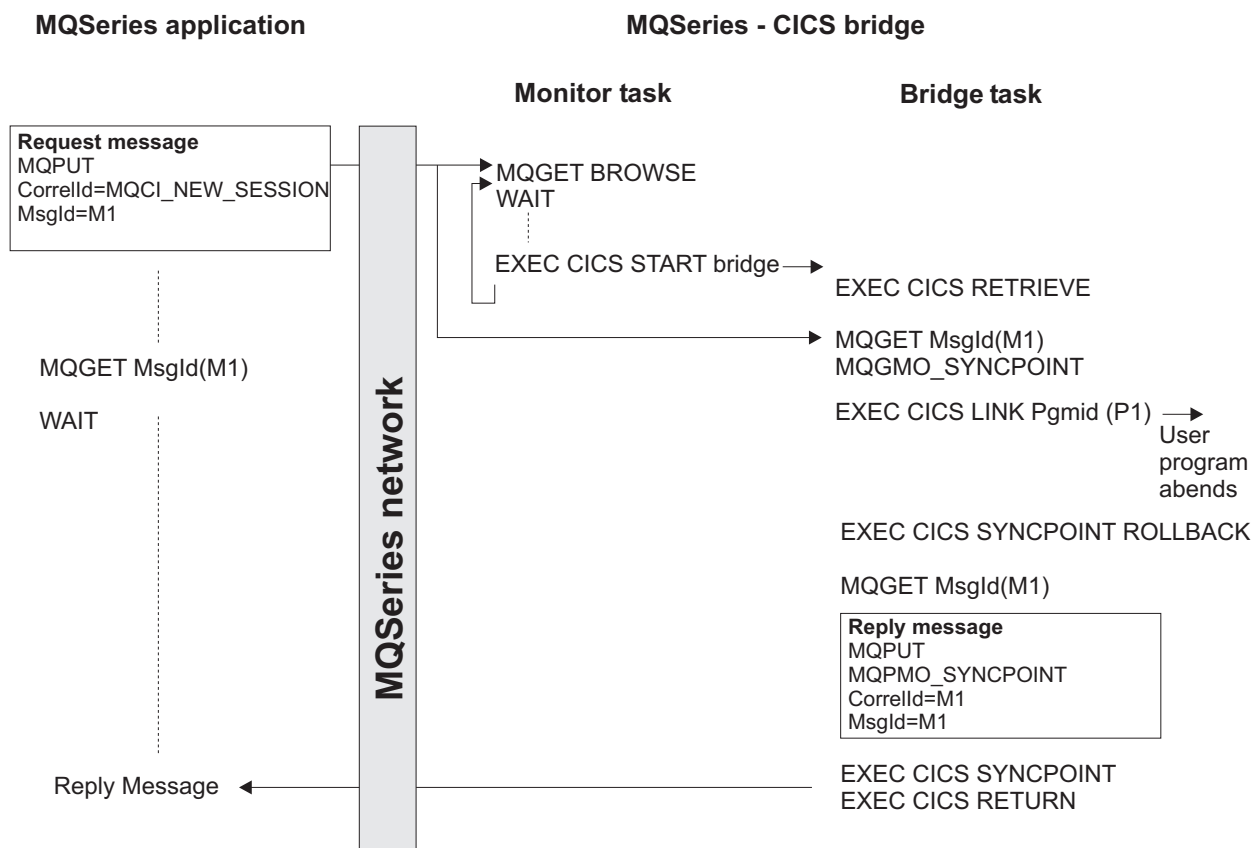


Figure 18. User program abends (only program in the unit of work)

In this figure:

- The client application sends a request message to run a CICS program named P1. The queue manager used by the client receives the message.

Bridge error handling

The monitor task browses the request queue awaiting the arrival of a message. When a message arrives, it:

- Gets the request message with browse
- Checks for any problems with the request message
- Starts a CICS bridge task
- Continues browsing the request queue

The CICS bridge task

- Gets the request message from the request queue under syncpoint control
- Takes the information in the request message and builds a COMMAREA for program P1
- Issues an EXEC CICS LINK call to program P1
- Waits for program P1 to complete

When these tasks are complete, the user program abends.

The CICS bridge task abend handler is driven, which:

- Issues an EXEC CICS SYNCPOINT ROLLBACK which:
 - Backs out all the changes made by P1
 - Reinstates the request message on the request queue
- Gets the request message a second time from the request queue, again under syncpoint control
- Copies the request to the dead-letter queue
- Puts an error reply to the reply-to queue

If the request message includes the name of a reply-to queue:

- Writes a CSQC7nn message to the CICS CSMT transient data queue

Debugging CICS bridge applications

This section describes some common symptoms when your bridge application appears not to work as you might expect, and suggests procedures for diagnosing the problem.

Message is PUT to the bridge request queue, but is not processed by the bridge monitor

1. Check that the bridge monitor is running. Issue CEMT I TASK and look for CKBR, or whatever other transaction identifier you are using for the bridge monitor.

If it is not running and you are expecting it to be triggered, make sure that the triggering options on the bridge request queue are correct. You should use a queue attribute of TRIGTYPE(FIRST).

If the bridge monitor was running but is no longer running, check the output in the CICS MSGUSR log to see if there has been an error that has caused the bridge monitor to terminate.

2. If the bridge request queue is defined with QSGDISP(SHARED), check that it also specifies INDXTYPE(CORRELID).
3. Browse the inbound message that is not being processed and check that the values of *MQMD.MsgId* and *MQMD.CorrelId* are correct. If this is the first message

in a unit of work or a pseudoconversation, *MQMD.CorrrelId* must be set to a value of *MQCI_NEW_SESSION* and *MQMD.MsgId* must be set to *MQMI_NONE* (binary zeros).

4. If this is not the first message in a unit of work or pseudoconversation, ensure that your application has checked previous reply messages adequately for possible errors. As a minimum, it should check the following fields in the *MQCIH*:
 - *MQCIH.ReturnCode*
 - *MQCIH.CompCode*
 - *MQCIH.TaskEndStatus*
 - *MQCIH.AbendCode*
 - *MQCIH.ErrorOffset*

Inbound message is taken from the request queue by the bridge monitor, but the CICS DPL program or CICS transaction fails to run

1. Check the output in the CICS MSGUSR log. This will almost always report the reason why the DPL program or transaction failed to run. The common reasons for this are:
 - Program or transaction not defined to CICS. Use CEDA to define the program or transaction and run your bridge application again.
 - Insufficient authority to run the program or transaction. Details of how to control the level of authentication used by the CICS bridge are given in the *MQSeries for OS/390 System Setup Guide*.
2. Check the message that is sent to the reply queue by the bridge monitor. If an error has occurred, it is likely that *MQCIH.Format* is set to *MQFMT_STRING* and an error message is appended to the *MQCIH* in place of a vector.
3. Check the dead letter queue to see if a reply message has been sent there by the bridge monitor. If it has, and the values of *MQMD.MsgId* and *MQMD.CorrrelId* are correct, check the value of *MQDLH.Reason*. This should be set to a feedback code that indicates the reason for the failure.

For information on feedback codes, including those specific to the CICS Bridge, see "MQMD - Message descriptor" in the *MQSeries Application Programming Reference* book.

Bridge task abends

Abend codes are set in outbound messages in field *MQCIH.AbendCode*. In addition, the output in the CICS MSGUSR log reports abend codes for failing bridge tasks.

Abends *ABR**, *ABS** and *ABX** are CICS bridge abends, and are documented in the *CICS Messages and Codes* book. Abends *MBR** and *MQB** are MQSeries bridge abends, and are documented in the *MQSeries for OS/390 Messages and Codes* book.

Some common abend codes can be dealt with as follows:

- ABRG** An invalid bridge facility token was specified in an inbound message. Your first inbound message must always specify a value of *MQCFAC_NONE* in field *MQCIH.Facility*, and a non zero value in *MQCIH.FacilityKeepTime*. CICS will return a facility token in field *MQCIH.Facility*, and you can use this value in all subsequent inbound messages in the pseudoconversation.
- MBRJ** The *MQCIH* has invalid data. Check the values in the *MQCIH* field by field to find the one that is out of range.

Debugging bridge applications

MBRN

The message is shorter than expected. There are one or two data length fields in every vector structure. The first is the very first fullword field in the standard header for all vectors, and it should be equal to the overall length of the vector including the variable length data. Some vectors also contain another fullword length field which gives just the length of the variable length data. If these values indicate more data than there actually is, the bridge task will abend MBRN.

MBRO and MBRP

There is an error in the vector structure (not the variable length data). The MQCIH field ERROROFFSET give the offset of the field in error. Check the values of the fields in the vector against the permitted values, which are described in the *CICS Internet and External Interfaces Guide* for CICS V1.2, or the *CICS External Interfaces Guide* for CICS V1.3..

Bridge monitor errors

Some errors can cause the bridge monitor transaction, CKBR, to terminate unexpectedly. If you are using triggered queues to start the monitor, and there are still messages on the bridge request queue, the CKTI transaction may attempt to restart CKBR. If the original error persists, this can lead to a loop of CKBR failures. To halt the loop you can set off the *TriggerControl* attribute of the request queue while you diagnose and fix the underlying problem.

The bridge monitor can fail if it does not have sufficient authority to access the queues or CICS transactions, if it is unable to write to the dead letter queue or it encounters problems when executing CICS or MQ services.

Application data structure terminology

Application data structure

The application data structure is the copybook generated during assembly of a BMS map. It has the mnemonic ADS, and it can be created in short form or long form. You will sometimes see the long form referred to as ADSL.

The short form of the ADS has fields that are not fullword aligned, whereas the long form of the ADS has all its fields fullword aligned.

The short form of the ADS is generated by default by map assembly. In order to obtain the long form of the ADS, you must assemble your BMS maps with the following parameters specified in the DFHMSD macro:

```
MSETNAM  DFHMSD
          ⋮
DSECT=ADSL,      *
LANG=C,          *
          ⋮
```

If you examine the DSECT that is produced on map assembly, you will see that all of the fields are fullword aligned. Be aware that this significantly increases the size of the application data structure and any message which includes it.

This option is only available for programs written in the C-language. However, It is possible to create COBOL data structures from such a

DSECT by manually creating them. The redbook *CICS Transaction Server for OS/390 Version 1 Release 3: Web Support and 3270 Bridge* gives examples of this.

Application data structure descriptor

The application data structure descriptor is an architected structure that allows an application to interpret the application data structure in a vector without having access to the copybook generated during map assembly.

There are two forms of the application data structure descriptor:

- The application data structure descriptor in short form contains fields that are not fullword aligned. It has the mnemonic ADSD.
- The application data structure descriptor in long form contains fields that are all fullword aligned. It has the mnemonic ADSDL.

ADSL – an ambiguous mnemonic

The mnemonic ADSL is ambiguous, and is sometimes used to refer to the application data *structure* in long form, and sometimes the application data structure *descriptor* in long form. The correct mnemonic for the application data structure descriptor, long form, is ADSDL. However, if you look at the eye-catcher in the ADSDL you will see that it is ADSL, which leads to the ambiguity. The correct use of ADSL is to describe the application data structure, not its descriptor, in long form.

ADS terminology

Chapter 17. IMS and IMS Bridge applications on MQSeries for OS/390

This chapter helps you to write IMS applications using MQSeries:

- To use syncpoints and MQI calls in IMS applications, see “Writing IMS applications using MQSeries”.
- To write applications that exploit the MQSeries-IMS Bridge, see “Writing MQSeries-IMS bridge applications” on page 269.

Writing IMS applications using MQSeries

This section discusses the following subjects that you should consider when using MQSeries in IMS applications:

- “Syncpoints in IMS applications”
- “MQI calls in IMS applications”

Syncpoints in IMS applications

In an IMS application, you establish a syncpoint by using IMS calls such as GU (get unique) to the IOPCB and CHKP (checkpoint). To back out all changes since the previous checkpoint, you can use the IMS ROLB (rollback) call. For more information, see the following books:

- *IMS/ESA Application Programming: Transaction Manager*
- *IMS/ESA Application Programming: Design Guide*

The queue manager is a participant in a two-phase commit protocol; the IMS syncpoint manager is the coordinator.

All open handles are closed by the IMS adapter at a syncpoint (except in a batch-oriented BMP). This is because a different user could initiate the next unit of work and MQSeries security checking is performed when the MQCONN, MQCONNX, and MQOPEN calls are made, not when the MQPUT or MQGET calls are made.

Handles are also closed after a ROLB call unless you are running IMS Version 3 or are running a batch-oriented BMP.

If an IMS application (either a BMP or an MPP) issues the MQDISC call, open queues are closed but no implicit syncpoint is taken. If the application closes down normally, any open queues are closed and an implicit commit occurs. If the application closes down abnormally, any open queues are closed and an implicit backout occurs.

MQI calls in IMS applications

This section covers the use of MQI calls in the following types of IMS applications:

- “Server applications” on page 266
- “Enquiry applications” on page 268

MQI calls in IMS applications

Server applications

Here is an outline of the MQI server application model:

```
Initialize/Connect
.
Open queue for input shared
.
Get message from MQSeries queue
.
Do while Get does not fail
.
  If expected message received
    Process the message
  Else
    Process unexpected message
  End if
.
  Commit
.
  Get next message from MQSeries queue
.
End do
.
Close queue/Disconnect
.
END
```

Sample program CSQ4ICB3 shows the implementation, in C/370™, of a BMP using this model. The program establishes communication with IMS first, and then with MQSeries:

```
main()
----
  Call InitIMS
  If IMS initialization successful
    Call InitMQM
    If MQSeries initialization successful
      Call ProcessRequests
      Call EndMQM
    End-if
  End-if

Return
```

The IMS initialization determines whether the program has been called as a message-driven or a batch-oriented BMP and controls MQSeries queue manager connection and queue handles accordingly:

```
InitIMS
-----
Get the IO, Alternate and Database PCBs
Set MessageOriented to true

Call ctdli to handle status codes rather than abend
If call is successful (status code is zero)
  While status code is zero
    Call ctdli to get next message from IMS message queue
    If message received
      Do nothing
    Else if no IOPBC
      Set MessageOriented to false
      Initialize error message
      Build 'Started as batch oriented BMP' message
      Call ReportCallError to output the message
    End-if
  Else if response is not 'no message available'
    Initialize error message
```

```

        Build 'GU failed' message
        Call ReportCallError to output the message
        Set return code to error
    End-if
End-if
End-while
Else
    Initialize error message
    Build 'INIT failed' message
    Call ReportCallError to output the message
    Set return code to error
End-if

Return to calling function

```

The MQSeries initialization performs queue manager connection opens the queues. In a Message-driven BMP this is called after each IMS syncpoint is taken; in a batch-oriented BMP, this is only called during program start-up:

```

InitMQM
-----
    Connect to the queue manager
    If connect is successful
        Initialize variables for the open call
        Open the request queue
        If open is not successful
            Initialize error message
            Build 'open failed' message
            Call ReportCallError to output the message
            Set return code to error
        End-if
    Else
        Initialize error message
        Build 'connect failed' message
        Call ReportCallError to output the message
        Set return code to error
    End-if

    Return to calling function

```

The implementation of the server model in an MPP is influenced by the fact that the MPP processes a single unit of work per invocation. This is because, when a syncpoint (GU) is taken, the connection and queue handles are closed and the next IMS message is delivered. This limitation can be partially overcome by one of the following:

- **Processing many messages within a single unit-of-work**

This involves:

- Reading a message
- Processing the required updates
- Putting the reply

in a loop until all messages have been processed or until a set maximum number of messages has been processed, at which time a syncpoint is taken.

Only certain types of application (for example, a simple database update or inquiry) can be approached in this way. Although the MQI reply messages can be put with the authority of the originator of the MQI message being handled, the security implications of any IMS resource updates need to be addressed carefully.

- **Processing one message per invocation of the MPP and ensuring multiple scheduling of the MPP to process all available messages.**

MQI calls in IMS applications

Use the MQSeries IMS trigger monitor program (CSQQTRMN) to schedule the MPP transaction when there are messages on the MQSeries queue and no applications serving it.

If the MPP is started by the trigger monitor the queue manager name and queue name are be passed to the program, as shown in the following COBOL code extract:

```
* Data definition extract
01 WS-INPUT-MSG.
   05 IN-LL1                PIC S9(3) COMP.
   05 IN-ZZ1                PIC S9(3) COMP.
   05 WS-STRINGPARM        PIC X(1000).
01 TRIGGER-MESSAGE.
   COPY CMQTM2L.

*
* Code extract
GU-IOPCB SECTION.
  MOVE SPACES TO WS-STRINGPARM.
  CALL 'CBLTDLI' USING GU,
                                IOPCB,
                                WS-INPUT-MSG.
  IF IOPCB-STATUS = SPACES
    MOVE WS-STRINGPARM TO MQTMC.
*   ELSE handle error
*
* Now use the queue manager and queue names passed
  DISPLAY 'MQTMC-QMGRNAME      ='
          MQTMC-QMGRNAME OF MQTMC '='.
  DISPLAY 'MQTMC-QNAME        ='
          MQTMC-QNAME   OF MQTMC '='.
```

The server model, which is expected to be a long running task, is better supported in a batch processing region, although the BMP cannot be triggered using CSQQTRMN.

Enquiry applications

A typical MQSeries application initiating an inquiry or update works as follows:

- Gather data from the user
- Put one or more MQSeries messages
- Get the reply messages (you might have to wait for them)
- Provide a response to the user

Because messages put on to MQSeries queues do not become available to other MQSeries applications until they are committed, they must either be put out of syncpoint, or the IMS application must be split into two transactions.

If the inquiry involves putting a single message, it is acceptable to use the “no syncpoint” option; however, if the inquiry is more complex, or resource updates are involved, you might get consistency problems if failure occurs and you don’t use syncpointing.

To overcome this, IMS MPP transactions using MQI calls could be split using a program-to-program message switch; see the *IMS/ESA Application Programming: Data Communication* manual for information about this. This would allow an inquiry program to be implemented in an MPP:

```
Initialize first program/Connect
.
Open queue for output
.
Put inquiry to MQSeries queue
.
Switch to second MQSeries program, passing necessary data in save
```



```

    pack area (this commits the put)
    .
END
    .
    .
    Initialize second program/Connect
    .
    Open queue for input shared
    .
    Get results of inquiry from MQSeries queue
    .
    Return results to originator
    .
END

```

Writing MQSeries-IMS bridge applications

This section discusses writing applications to exploit the MQSeries-IMS bridge. The following topics are discussed:

- “How the MQSeries-IMS bridge deals with messages”
- “Writing your program” on page 275
- “Triggering” on page 276

For information about the MQSeries-IMS bridge, see the *MQSeries for OS/390 Concepts and Planning Guide*.

How the MQSeries-IMS bridge deals with messages

When you use the MQSeries-IMS bridge to send messages to an IMS application, you need to construct your messages in a special format. You must also put your messages on MQSeries queues that have been defined with a storage class that specifies the XCF group and member name of the target IMS system. These are known as MQ-IMS bridge queues, or simply **bridge** queues.

A user does not need to sign on to IMS before sending messages to an IMS application. The user ID in the *UserIdentifier* field of the MQMD structure is used for security checking. The level of checking is determined when MQSeries connects to IMS, and is described in the security section of the *MQSeries for OS/390 System Setup Guide*. This enables a pseudo sign-on to be implemented.

The MQSeries-IMS bridge accepts the following types of message:

- Messages containing IMS transaction data and an MQIIH structure (described in the *MQSeries Application Programming Reference* manual):

```
MQIIH LLZZ<trancode><data>[LLZZ<data>][LLZZ<data>]
```

Notes:

1. The square brackets, [], represent optional multi-segments.
 2. The *Format* field of the MQMD structure must be set to MQFMT_IMS to use the MQIIH structure.
- Messages containing IMS transaction data but no MQIIH structure:

```
LLZZ<trancode><data> \
[LLZZ<data>][LLZZ<data>]
```

MQSeries validates the message data to ensure that the sum of the LL bytes plus the length of the MQIIH (if it is present) is equal to the message length.

How the IMS bridge deals with messages

When the MQSeries-IMS bridge gets messages from the bridge queues, it processes them as follows:

- If the message contains an MQIIH structure the bridge verifies the MQIIH (see the *MQSeries Application Programming Reference* manual) builds the OTMA headers and sends the message to IMS. The transaction code is specified in the input message. If this is an LTERM, IMS replies with a DFS1288E message. If the transaction code represents a command, IMS executes the command, otherwise the message is queued in IMS for the transaction.
- If the message contains IMS transaction data, but no MQIIH structure, the IMS bridge makes the following assumptions:
 - The transaction code is in bytes 5 through 12 of the user data
 - The transaction is in non-conversational mode
 - The transaction is in commit mode 0 (commit-then-send)
 - The *Format* in the MQMD is used as the *MFSMapName* (on input)
 - The security mode is MQISS_CHECK

The reply message is also built without an MQIIH structure, taking the *Format* for the MQMD from the *MFSMapName* of the IMS output.

The MQSeries-IMS bridge uses one or two Tpipes for each MQSeries queue:

- A synchronized Tpipe is used for all messages using Commit mode 0 (COMMIT_THEN_SEND) (these show with SYN in the status field of the IMS /DIS TMEMBER client TPIPE xxxx command)
- A non-synchronized Tpipe is used for all messages using Commit mode 1 (SEND_THEN_COMMIT)

The Tpipes are created by MQSeries when they are first used. A non-synchronized Tpipe exists until IMS is restarted. Synchronized Tpipes exist until IMS is cold started. You cannot delete these Tpipes yourself.

Mapping MQSeries messages to IMS transaction types

Table 12. Mapping MQSeries messages to IMS transaction types

MQSeries message type	Commit-then-send (mode 0) - uses synchronized IMS Tpipes	Send-then-commit (mode 1) - uses non-synchronized IMS Tpipes
Persistent MQSeries messages	<ul style="list-style-type: none"> • Recoverable full function transactions • Irrecoverable transactions are rejected by IMS 	<ul style="list-style-type: none"> • Fastpath transactions • Conversational transactions • Full function transactions
Nonpersistent MQSeries messages	<ul style="list-style-type: none"> • Irrecoverable full function transactions • Recoverable transactions are rejected by IMS 	<ul style="list-style-type: none"> • Fastpath transactions • Conversational transactions • Full function transactions
<p>Note: IMS commands cannot use persistent MQSeries messages with commit mode 0. See the <i>IMS/ESA Open Transaction Manager Access User's Guide</i> for more information.</p>		

If the message cannot be put to the IMS queue

If the message cannot be put to the IMS queue, the following action is taken by MQSeries:

- If a message cannot be put to IMS because the message is invalid, the message is put to the dead-letter queue and a message is sent to the system console.
- If the message is valid, but is rejected by IMS, MQSeries sends an error message to the system console, the message includes the IMS sense code, and the

How the IMS bridge deals with messages

MQSeries message is put to the dead-letter queue. If the IMS sense code is 001A, IMS will send an MQSeries message containing the reason for the failure to the reply-to queue.

Note: In the circumstances listed above, if MQSeries is unable to put the message to the dead-letter queue for any reason, the message is returned to the originating MQSeries queue. An error message is sent to the system console, and no further messages are sent from that queue.

To resend the messages, do *one* of the following:

- Stop and restart the Tpipes in IMS corresponding to the queue
 - Alter the queue to GET(DISABLED), and again to GET(ENABLED)
 - Stop and restart IMS or the OTMA
 - Stop and restart your MQSeries subsystem
- If the message is rejected by IMS for anything other than a message error, the MQSeries message is returned to the originating queue, MQSeries stops processing the queue, and an error message is sent to the system console. If an exception report message is required, the bridge puts it to the reply-to queue with the authority of the originator. If the message cannot be put to the queue, the report message is put to the dead-letter queue with the authority of the bridge. If it cannot be put to the DLQ, it is discarded.

IMS bridge feedback codes

IMS sense codes are normally output in hexadecimal format in MQSeries console messages such as CSQ2001I (for example, sense code 001A). MQSeries feedback codes as seen in the dead-letter header of messages put to the dead-letter queue are decimal numbers.

The IMS bridge feedback codes are in the range 301 through 399. They are mapped from the IMS-OTMA sense codes as follows:

1. The IMS-OTMA sense code is converted from a hexadecimal number to a decimal number.
2. 300 is added to the number resulting from the calculation in 1, giving the MQSeries *Feedback* code.

Refer to the *IMS/ESA Open Transaction Manager Access Guide* for information about IMS-OTMA sense codes.

The MQMD fields in messages from the IMS bridge

The MQMD of the originating message is carried by IMS in the User Data section of the OTMA headers. If the message originates in IMS, this is built by the IMS Destination Resolution Exit. The MQMD of a message received from IMS is built as follows:

StrucID	"MD "
Version	MQMD_VERSION_1
Report	MQRO_NONE
MsgType	MQMT_REPLY
Expiry	MQWI_UNLIMITED
Feedback	MQFB_NONE
Encoding	MQENC.Native (the encoding of the OS/390 system)

How the IMS bridge deals with messages

CodedCharSetId	MQCCSI_Q_MGR (the CodedCharSetID of the OS/390 system)
Format	MQFMT_IMS if the MQMD.Format of the input message is MQFMT_IMS, otherwise IOPCB.MODNAME
Priority	MQMD.Priority of the input message
Persistence	Depends on commit mode: MQMD.Persistence of the input message if CM-1; persistence will match recoverability of the IMS message if CM-0
MsgId	MQMD.MsgId if MQRO_PASS_MSG_ID, otherwise New MsgId (the default)
CorrelId	MQMD.CorrelId from the input message if MQRO_PASS_CORREL_ID, otherwise MQMD.MsgId from the input message (the default)
BackoutCount	0
ReplyToQ	Blanks
ReplyToQMgr	Blanks (set to local qmgr name by the queue manager during the MQPUT)
UserIdentifier	MQMD.UserIdentifier of the input message
AccountingToken	MQMD.AccountingToken of the input message
ApplIdentityData	MQMD.ApplIdentityData of the input message
PutApplType	MQAT_XCF if no error, otherwise MQAT_BRIDGE
PutAppName	<XCFgroupName><XCFmemberName> if no error, otherwise QMGR name
PutDate	Date when message was put
PutTime	Time when message was put
ApplOriginData	Blanks

The MQIIH fields in messages from the IMS bridge

The MQIIH of a message received from IMS is built as follows:

StrucId	"IIH "
Version	1
StrucLength	84
Encoding	MQENC_NATIVE
CodedCharSetId	MQCCSI_Q_MGR
Format	MQIIH.ReplyToFormat of the input message if MQIIH.ReplyToFormat is not blank, otherwise IOPCB.MODNAME
Flags	0
LTermOverride	LTERM name (Tpipe) from OTMA header

How the IMS bridge deals with messages

MFSMapName	Map name from OTMA header
ReplyToFormat	Blanks
Authenticator	Blanks
TranInstanceId	Conversation ID / Server Token from OTMA header if in conversation, otherwise nulls
TranState	"C" if in conversation, otherwise blank
CommitMode	Commit mode from OTMA header ("0" or "1")
SecurityScope	Blank
Reserved	Blank

Reply messages from IMS

When an IMS transaction ISRTs to its IOPCB, the message is routed back to the originating LTERM or TPIPE. These are seen in MQSeries as reply messages. Reply messages from IMS are put onto the reply-to queue specified in the original message. If the message cannot be put onto the reply-to queue, it is put onto the dead-letter queue using the authority of the bridge. If the message cannot be put onto the dead-letter queue, a negative acknowledgement is sent to IMS to say that the message cannot be received. Responsibility for the message is then returned to IMS. If you are using commit mode 0, messages from that Tpipe are not sent to the bridge, and remain on the IMS queue; that is, no further messages are sent until restart. If you are using commit mode 1, other work can continue.

If the reply has an MQIIH structure, its format type is MQFMT_IMS; if not, its format type is specified by the IMS MOD name used when inserting the message.

Using alternate response PCBs in IMS transactions

When an IMS transaction uses alternate response PCBs (ISRTs to the ALTPCB, or issues a CHNG call to a modifiable PCB), the Pre-routing Exit (DFSYPRX0) is invoked to determine if the message should be re-routed. If the message is to be re-routed, the Destination Resolution Exit (DFSYDRU0) is invoked to confirm the destination and prepare the header information. See the *MQSeries for OS/390 System Setup Guide* for information about these exit programs.

Unless action is taken in the exits, all output from IMS transactions initiated from an MQSeries queue manager, whether to the IOPCB or to an ALTPCB, will be returned to the same queue manager.

Sending unsolicited messages from IMS

To send messages from IMS to an MQSeries queue, you need to invoke an IMS transaction that ISRTs to an ALTPCB. You need to write pre-routing and destination resolution exits to route unsolicited messages from IMS and build the OTMA user data, so that the MQMD of the message can be built correctly. See the *MQSeries for OS/390 System Setup Guide* for information about these exit programs.

Note: The MQSeries-IMS bridge does not know whether a message it receives is a reply or an unsolicited message. It handles the message the same way in each case, building the MQMD and MQIIH of the reply based on the OTMA UserData that arrived with the message

How the IMS bridge deals with messages

Unsolicited messages can create new Tpipes. For example if an existing IMS transaction switched to a new LTERM (for example PRINT01) but the implementation required that the output be delivered through OTMA; a new Tpipe (called PRINT01 in this example) would be created. By default this will be a non-synchronized Tpipe. If the implementation requires the message to be recoverable the destination resolution exit Output flag must be set. See the *IMS Customization Guide* for more information.

Message segmentation

IMS transactions may be defined as expecting single- or multi-segment input. The originating MQSeries application must construct the user input following the MQIIH structure as one or more LLZZ-data segments. All segments of an IMS message must be contained in a single MQSeries message sent with a single MQPUT.

The maximum length of any one LLZZ-data segment is defined by IMS/OTMA (32764 bytes). The total MQSeries message length is the sum of the LL bytes, plus the length of the MQIIH structure.

All the segments of the reply are contained in a single MQSeries message.

There is a further restriction on the 32 KB limitation on messages with format MQFMT_IMS_VAR_STRING. When the data in an ASCII mixed CCSID message is converted to an EBCDIC mixed CCSID message, a shift-in byte or a shift-out byte is added every time there is a transition between SBCS and DBCS characters. The 32 KB restriction applies to the maximum size of the message. That is, because the LL field in the message cannot exceed 32 KB, the message must not exceed 32 KB including all shift-in and shift-out characters. The application building the message must allow for this.

Data conversion

The data conversion (including the calling of any necessary exits) is performed by the distributed queuing facility when it puts a message to a destination queue that has XCF information defined for its storage class. Any exits needed must be available to the distributed queuing facility in the data set referenced by the CSQXLIB DD statement. This means that you can send messages to an IMS application using the MQSeries-IMS bridge from any MQSeries platform.

Note: Because the MQSeries-IMS bridge does not convert messages when it gets a message, messages arriving through the CICS distributed queuing facility are not converted.

If there are conversion errors, the message is put to the queue unconverted; this results eventually in it being treated as an error by the MQSeries-IMS bridge, because the bridge cannot recognize the header format. If a conversion error occurs, an error message is sent to the OS/390 console.

See “Chapter 11. Writing data-conversion exits” on page 155 for detailed information about data conversion in general.

Sending messages to the MQSeries-IMS bridge: To ensure that conversion is performed correctly, you must tell the queue manager what the format of the message is. If the message has an MQIIH structure, the *Format* in the MQMD must be set to the built-in format MQFMT_IMS, and the *Format* in the MQIIH must be set to the name of the format that describes your message data. If there is no MQIIH, set the *Format* in the MQMD to your format name.

How the IMS bridge deals with messages

If your data (other than the LLZZs) is all character data (MQCHAR), use as your format name (in the MQIIH or MQMD, as appropriate) the built-in format MQFMT_IMS_VAR_STRING. Otherwise, use your own format name, in which case you must also provide a data-conversion exit for your format. The exit must handle the conversion of the LLZZs in your message, in addition to the data itself (but it does not have to handle any MQIIH at the start of the message).

If your application makes use of *MFSMapName*, you are recommended to use messages with the MQFMT_IMS instead, and define the map name passed to the IMS transaction in the MFSMapName field of the MQIIH.

Receiving messages from the MQSeries-IMS bridge: If an MQIIH structure is present on the original message that you are sending to IMS, one is also present on the reply message.

To ensure your reply is converted correctly, follow these steps:

- If you have an MQIIH structure on your original message, specify the format you want for your reply message in the MQIIH *ReplytoFormat* field of the original message. This value is placed in the MQIIH *Format* field of the reply message. This is particularly useful if all your output data is of the form LLZZ<character data>.
- If you do not have an MQIIH structure on your original message, specify the format you want for the reply message as the MFS MOD name in the IMS application's ISRT to the IOPCB.

Writing your program

The coding required to handle IMS transactions through MQSeries is dependent on the message format required by the IMS transaction and the range of responses it may return. However, there are several points to be borne in mind when your application handles IMS screen formatting information.

When an IMS transaction is started from a 3270 screen, the message passes through IMS Message Format Services. This can remove all terminal dependency from the data stream seen by the transaction. When a transaction is started through OTMA, MFS is not involved. If application logic is implemented in MFS, this will have to be recreated in the new application.

In some IMS transactions, the end-user application can modify certain 3270 screen behavior, for example, highlighting a field which has had invalid data entered. This type of information is communicated by adding a two byte attribute field to the IMS message for each screen field needing to be modified by the program.

Thus, if you are coding an application to mimic a 3270, you need to take account of these fields when building or receiving messages.

You may need to code information in your program to process:

- Which key is pressed (Enter, PF1, ...)
- Where the cursor is when the message is passed to your application
- Whether the attribute fields have been set by the IMS application
 - High/normal/zero intensity
 - Color
 - Whether IMS is expecting the field back the next time enter is pressed
- Whether the IMS application has used null characters (X'3F') in any fields.

Writing your IMS bridge application

If your IMS message contains only character data (apart from the LLZZ-data segment), and you are using an MQIIH structure, set the MQMD format to MQFMT_IMS and the MQIIH format to MQFMT_IMS_VAR_STRING.

If your IMS message contains only character data (apart from the LLZZ-data segment), and you are *not* using an MQIIH structure, set the MQMD format to MQFMT_IMS_VAR_STRING and ensure that your IMS application specifies MODname MQFMT_IMS_VAR_STRING when replying. If a problem occurs (for example, user not authorised to use the transaction) and IMS sends an error message, this will have a MODname of the form DFSMOx, where 'x' is a number between 1 and 5. This will be put in the MQMD.Format.

If your IMS message contains binary, packed, or floating point data (apart from the LLZZ-data segment), you will need to code your own data-conversion routines. Refer to the *IMS/ESA Application Programming: Transaction Manager* manual for information about IMS screen formatting.

Writing MQSeries applications to invoke IMS conversational transactions

When you write an application which will invoke an IMS conversation, you should bear the following in mind:

- You must include an MQIIH structure with your application message.
- You must set the *CommitMode* in MQIIH to MQICM_SEND_THEN_COMMIT.
- To invoke a new conversation, set *TranState* in MQIIH to MQITS_NOT_IN_CONVERSATION.
- To invoke second and subsequent steps of a conversation, set *TranState* to MQITS_IN_CONVERSATION, and set *TranInstanceId* to the value of that field returned in the previous step of the conversation.
- There is no easy way in IMS to find the value of a *TranInstanceId*, should you lose the original message sent from IMS.
- The application must check the *TranState* of messages from IMS to check whether the IMS transaction has terminated the conversation.
- You can use /EXIT to end a conversation. You must also quote the *TranInstanceId*, set *TranState* to MQITS_IN_CONVERSATION, and use the MQSeries queue on which the conversation is being carried out.
- You cannot use /HOLD or /REL to hold or release a conversation.
- Conversations invoked through the MQSeries-IMS bridge are terminated if IMS is restarted.

Triggering

The MQSeries-IMS bridge does not support trigger messages.

If you define an initiation queue that uses a storage class with XCF parameters, messages put to that queue are rejected when they get to the bridge.

Writing programs containing IMS commands

An application programme can build an MQSeries message of the form LLZZ<command>, instead of a transaction, where <command> is of the form /DIS TRAN PART or /DIS POOL ALL. Most IMS commands can be issued in this way, see the *IMS/ESA V6 OTMA Guide and Reference* for details. The command output will be received in the MQSeries reply message in the text form as would be sent to a 3270 terminal for display.

Writing your IMS bridge application

| OTMA has implemented a special form of the IMS display transaction command,
| which will return an architected form of the output. The exact format is defined in
| the *IMS/ESA V6 OTMA Guide and Reference*. To invoke this form from an MQSeries
| message, build the message data as before, for example /DIS TRAN PART, and set
| the TranState field in the MQOOH to MQITS_ARCHITECTED. The command will
| be processed by IMS, and the reply returned in the architected form. An
| architected response contains all the information that could be found in the text
| form of the output, and one additional piece of information: whether the
| transaction is defined as recoverable or non-recoverable.

Writing your IMS bridge application

Chapter 18. Object-oriented programming with MQSeries

The preceding chapters have described the procedural Message Queue Interface (MQI), which may be used from programming languages such as COBOL, PL/I, C, and C++. The MQI comprises calls, structures, and elementary data types to allow an application programmer to create MQSeries applications.

MQSeries provides an alternative way of programming MQSeries applications, that can be used from object-oriented programming languages. It is called the *MQSeries Object Model*. Instead of calls and structures, the MQSeries Object Model provides *classes* that provide the same functionality, but which are a more natural way of programming in an object-oriented environment.

What is in the MQSeries Object Model?

The MQSeries Object Model comprises the following:

- *Classes* representing familiar MQSeries concepts such as queue managers, queues, and messages.
- *Methods* on each class corresponding to MQI calls.
- *Properties* on each class corresponding to attributes of MQSeries objects.

When creating an MQSeries application using the MQSeries Object Model, you create instances of these classes in the program. An instance of a class in object-oriented programming is called an *object*. When an object has been created, you interact with the object by examining or setting the values of the object's properties (the equivalent of issuing an MQINQ or MQSET call), and by making method calls against the object (the equivalent of issuing the other MQI calls).

Classes

The MQSeries Object Model provides the following base set of classes. Note that the actual implementation of the model varies slightly between the different supported object-oriented environments.

MQQueueManager

An object of the MQQueueManager class represents a connection to a queue manager. It has methods to Connect(), Disconnect(), Commit(), and Backout() (the equivalent of MQCONN or MQCONNX, MQDISC, MQCMIT, and MQBACK). It has properties corresponding to the attributes of a queue manager. Note that accessing a queue manager attribute property implicitly connects to the queue manager if not already connected. Destroying an MQQueueManager object implicitly disconnects from the queue manager.

MQQueue

An object of the MQQueue class represents a queue. It has methods to Put() and Get() messages to and from the queue (the equivalent of MQPUT and MQGET). It has properties corresponding to the attributes of a queue. Note that accessing a queue attribute property, or issuing a Put() or Get() method call, implicitly opens the queue (the equivalent of MQOPEN). Destroying an MQQueue object implicitly closes the queue (the equivalent of MQCLOSE).

Object-oriented programming

MQMessage

An object of the MQMessage class represents a message to be put on a queue or got from a queue. It comprises a buffer, and encapsulates both application data and MQMD. It has properties corresponding to MQMD fields, and methods that allow you to write and read user data of different types (for example, strings, long integers, short integers, single bytes) to and from the buffer.

MQPutMessageOptions

An object of the MQPutMessageOptions class represents the MQPMO structure. It has properties corresponding to MQPMO fields.

MQGetMessageOptions

An object of the MQGetMessageOptions class represents the MQGMO structure. It has properties corresponding to MQGMO fields.

MQProcess

An object of the MQProcess class represents a process definition (used with triggering). It has properties that represent the attributes of a process definition.

MQDistributionList

MQSeries Version 5 products only, except MQSeries for OS/390. An object of the MQDistributionList class represents a distribution list (used to send multiple messages with a single MQPUT). It comprises a list of MQDistributionListItem objects.

MQDistributionListItem

MQSeries Version 5 products only except MQSeries for OS/390. An object of the MQDistributionListItem class represents a single distribution list destination. It encapsulates the MQOR, MQRR, and MQPMR structures, and has properties corresponding to the fields of these structures.

Object references

In an MQSeries program that uses the MQI, MQSeries returns connection handles and object handles to the program. These handles must be passed as parameters on subsequent MQSeries calls. With the MQSeries Object Model, these handles are hidden from the application program. Instead, the creation of an object from a class results in an object reference being returned to the application program. It is this object reference that is used when making method calls and property accesses against the object.

Return codes

Issuing a method call or setting a property value results in return codes being set. These return codes are a completion code and a reason code, and are themselves properties of the object. The values of completion code and reason code are exactly the same as those defined for the MQI, with some extra values specific to the object-oriented environment.

Programming language considerations

The MQSeries Object Model is implemented in C++, Java, LotusScript, and ActiveX[®].

Coding in C++

Refer to the *MQSeries Using C++* book for information about coding programs using the MQSeries Object Model in C++.

Coding in Java

Refer to the *MQSeries Using Java* book for information about coding programs using the MQSeries Object Model in Java.

Coding in LotusScript

Refer to the *MQSeries LotusScript Extension* book for information about coding programs using the MQSeries Object Model in LotusScript.

The MQSeries link LotusScript Extension is commonly known as the MQLSX. For Windows NT the MQLSX is included as part of MQSeries for Windows NT, V5.1. For other platforms, or for earlier releases of MQSeries, the MQLSX and its documentation may be downloaded from the MQSeries Web site as a SupportPac.

Coding in ActiveX

Refer to the *MQSeries for Windows NT Using the Component Object Model Interface* for information about coding programs using the MQSeries Object Model in ActiveX.

The MQSeries ActiveX is commonly known as the MQAX. The MQAX is included as part of MQSeries for Windows NT, V5.1. For earlier releases of MQSeries for Windows NT, the MQAX and its documentation may be downloaded from the MQSeries Web site as a SupportPac.

Part 3. Building an MQSeries application

Chapter 19. Building your application on AIX	285	Building applications in C++ on Compaq Tru64	
Preparing C programs	285	UNIX Version 4.0	301
Linking libraries	285	C++ language include files	301
Preparing COBOL programs	286	Preparing C++ programs	302
Preparing COBOL programs using IBM COBOL		Linking libraries	302
SET for AIX	287	Building applications in C++ on Compaq Tru64	
Preparing COBOL programs using Micro Focus		UNIX Version 5.0	302
COBOL	287	C++ language include files	302
Preparing PL/I programs	287	Preparing C++ programs	303
Preparing CICS programs	287	Linking libraries	303
CICS on Open Systems support	288	Building applications in COBOL	303
Preparing CICS COBOL programs using IBM		Preparing COBOL programs	303
COBOL SET for AIX	288	Linking libraries	304
Preparing CICS COBOL programs using		Building applications in Java	304
Micro Focus COBOL	288	Preparing Java programs	304
Preparing CICS C programs	289		
Chapter 20. Building your application on AS/400	291	Chapter 24. Building your application on HP-UX	305
Preparing C programs	291	Preparing C programs	305
Preparing COBOL programs	292	Preparing C programs on HP-UX V10.20	305
Preparing CICS programs	293	Preparing C programs on HP-UX V11.00	306
Preparing RPG programs	293	Linking libraries	306
SQL programming considerations	294	Preparing COBOL programs	307
AS/400 programming considerations	294	Programs to run in the MQSeries client	
QMQM activation group	294	environment.	307
		Preparing CICS programs	308
Chapter 21. Building your application on AT&T		CICS on Open Systems support	308
GIS UNIX	295	CICS C sample transaction	308
Preparing C programs	295	Preparing CICS COBOL programs using	
C compiler flags	295	Micro Focus COBOL	308
Linking libraries	295		
		Chapter 25. Building your application on OS/390	311
Chapter 22. Building your application on Digital		Preparing your program to run	311
OpenVMS	297	Building OS/390 batch applications	312
Preparing C programs	297	Building CICS applications	313
C compiler version	297	Building IMS (BMP or MPP) applications	314
C compiler flags	297	Dynamically calling the MQSeries stub	315
Linking libraries	297	Debugging your programs	319
Preparing COBOL programs	297	Debugging CICS programs	320
COBOL compiler flags	298	CICS trace	320
Linking libraries	298	Debugging TSO programs	322
Chapter 23. Building your application on		Chapter 26. Building your application on OS/2	
Compaq Tru64 UNIX	299	Warp	323
Building applications in C on Compaq Tru64 UNIX		Preparing C programs	323
Version 4.0	299	Preparing CICS and Transaction Server	
C language include files	299	programs	324
Preparing C programs	299	Preparing COBOL programs	325
Linking libraries	300	Preparing Transaction Server programs	326
Building applications in C on Compaq Tru64 UNIX		Preparing PL/I programs	326
Version 5.0	300		
C language include files	300	Chapter 27. Building your application on SINIX	
Preparing C programs	301	or DC/OSx	327
Linking libraries	301	Preparing C programs	327
		C compiler flags	327
		Preparing COBOL programs	328

Compiling COBOL programs	328
Preparing CICS programs	329
CICS on Open Systems support	329
CICS sample transaction.	329
Linking libraries	330

Running the sample program	353
Program design	354

Chapter 28. Building your application on Sun

Solaris	331
Preparing C programs	331
Linking libraries	332
Preparing COBOL programs	332
Preparing CICS programs	333
CICS on Open Systems support	333
Preparing CICS COBOL programs using Micro Focus COBOL	334
Preparing CICS C programs	334

Chapter 29. Building your application on

Tandem NSK	335
Unit of work (transaction) management	335
General design considerations	335
MQGMO_BROWSE_* with MQGMO_LOCK	335
Triggered applications	335
Compiling and binding applications.	336
Running applications.	337

Chapter 30. Building your application on

VSE/ESA	339
Linking library	339
Using the batch interface	339
Preparing C programs	339
Preparing COBOL programs	339
Preparing PL/I programs	339

Chapter 31. Building your application on

Windows	341
Linking libraries	341
Preparing Visual Basic programs	341

Chapter 32. Building your application on

Windows NT	343
Preparing C programs	343
Preparing CICS and Transaction Server programs.	344
Preparing COBOL programs	345
Preparing CICS and Transaction Server programs.	346
Preparing PL/I programs	347
Preparing Visual Basic programs	347

**Chapter 33. Using lightweight directory access
protocol services with MQSeries for Windows**

NT	349
What is a directory service?	349
What is LDAP?.	349
Using LDAP with MQSeries	350
LDAP sample program	351
Building the sample program	351
Configuring the directory	351
Configuring the IBM eNetwork LDAP server	351
Configuring the Netscape directory server.	353

Chapter 19. Building your application on AIX

The AIX 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 AIX applications to run under AIX. C, C++, and COBOL are supported. For information about preparing your C++ programs, see the *MQSeries Using C++* book.

The tasks you must perform to create an executable application using MQSeries for AIX vary with the programming language your source code is written in. In addition to coding the MQI calls in your source code, you must add the appropriate language statements to include the MQSeries for AIX include files for the language you are using. You should make yourself familiar with the contents of these files. See “Appendix F. MQSeries data definition files” on page 541 for a full description.

EXTSHM environment variable

AIX Version 4.3.1 introduces the EXTSHM environment variable.

Do not set the EXTSHM environment variable before you issue the `strmqm` command. If you attempt to set this variable before you issue the `strmqm` command, the `strmqm` command will fail.

All other MQSeries operations work correctly with this variable set.

Preparing C programs

Precompiled C programs are supplied in the `/usr/mqm/samp/bin` directory. Use the ANSI compiler and run the following command:

```
$ cc -o <amqsput> <amqsput>.c -lmqm
```

where `amqsput` is a sample program.

If you want to use the programs on a machine which has only the MQSeries client for AIX installed, recompile the programs to link them with the client library (`-lmqic`) instead.

Linking libraries

You will need the following libraries:

- If your application is running in a DCE client environment you will need to copy the DCE library, `libxdsom.a`, on to your machine.
- You need to link your programs with the appropriate library provided by MQSeries.

In a non-threaded environment you must link to one of the following libraries:

Library file

Program/exit type

libmqm.a

Server for C

Preparing C programs

libmqic.a
Client for C

In a threaded environment, you must link to one of the following libraries:

Library file
Program/exit type

libmqm_r.a
Server for C

libmqic_r.a
Client for C

For example, to build a simple threaded MQSeries application from a single compilation unit on AIX 4.3 run the following command:

```
$ xlc_r7 -o myapp myapp.c -lmqm_r
```

where myapp is the name of your program.

Notes:

1. If you are writing an installable service (see the *MQSeries Programmable System Management* book for further information), you need to link to the libmqmzf.a library in a non-threaded application and to the libmqmzf_r.a library in a threaded application.
2. If you are producing an XA switch load file for external coordination by an XA-compliant transaction manager such as IBM CICS, Transarc Encina, or Novell Tuxedo, you need to link to the libmqmxa.a library in a non-threaded application and to the libmqmxa_r.a library in a threaded application.
3. You need to link trusted applications to the threaded MQSeries libraries. However, only one thread in an MQSeries on UNIX systems trusted application can be connected at a time.
4. To run the sample Encina program, link against the following libraries:
 - libmqmxa_r.a
 - libmqm_r.a

Also, link to the Encina and DCE libraries:

- libEncServer.a
- libEncina.a
- libdce.a

The sample must be compiled and linked using xlc_r4.

5. You must link MQSeries libraries before any other product libraries (in this case, DCE and Encina). For example:

```
cc -o put put.c -lmqm_r -ldce
```

This ensures that any operating system functions that have been redefined by DCE are also used by MQSeries.

Preparing COBOL programs

You need to link your program with one of the following:

libmqmcb.a
Server for COBOL

libmqicb.a
Client for COBOL

libmqmcb_r.a
Server for COBOL (in a threaded application)

Preparing COBOL programs

You can use the IBM COBOL Set compiler or Micro Focus COBOL compiler depending on the program:

- Programs beginning amqi are suitable for the IBM COBOL Set compiler,
- programs beginning amqm are suitable for the Micro Focus COBOL compiler,

and

- programs beginning amq0 are suitable for either compiler.

Preparing COBOL programs using IBM COBOL SET for AIX

Sample COBOL programs are supplied with MQSeries. To compile such a program, enter:

```
cob2 -o amq0put0 amq0put0.cb1
-L/usr/mqm/lib
-lmqmcb -qLIB
-I/usr/mqm/inc
```

Note: For threaded applications, cob2_r is used with the libmqmcb_r.a library.

Preparing COBOL programs using Micro Focus COBOL

Set environment variables before compiling your program as follows:

```
export COBCPY=/usr/mqm/inc
export LIB=/usr/mqm/lib;$LIB
```

To compile a COBOL program using Micro Focus COBOL, enter:

```
cob -xvP amq0put0.cb1 -lmqmcb
```

See the Micro Focus COBOL documentation for a description of the environment variables that need to be set up.

Preparing PL/I programs

Sample PL/I programs are supplied with MQSeries. PL/I include files are also provided so that the C entry points in the MQSeries libraries can be invoked directly.

To prepare a PL/I program:

1. Link your program with one of the libraries listed in "Linking libraries" on page 285.
2. Compile your program:

```
pli amqput0.pli -I/usr/mqm/inc /usr/mqm/lib/libmqm.a
```

Preparing CICS programs

XA switch modules are provided to enable you to link CICS with MQSeries:

Table 13. Essential Code for CICS applications (AIX)

Description	C (source)	C (exec) - add to your XAD.Stanza
XA initialization routine	amqzscix.c	amqzsc21 - CICS for AIX

Always link your transactions with the thread safe MQSeries library libmqm_r.a.

Preparing CICS programs

Note: On AIX Version 4, the libmqm_r.a library works with both native and DCE libraries.

Compile the program by typing:

```
xlc_r4 /usr/mqm/samp/amqzscix.c -I/usr/lpp/encina/include \  
-e amqzscix -o amqzscix /usr/lpp/cics/lib/regxa_swx.a \  
-L/usr/lpp/cics/lib -L/usr/lpp/encina/lib -lmqmcics_r -lmqmx_r -lmqm_r \  
-lcicsrt -lEncina -lEncServer -ldce
```

You can find more information about supporting CICS transactions in the *MQSeries System Administration* book.

CICS on Open Systems support

MQSeries on UNIX systems support CICS on Open Systems via the XA interface.

You must ensure that CICS COBOL applications are linked to the threaded version of the library. CICS on Open Systems MQSeries transactions must link with libmqm_r, except on Sun Solaris, where you must link with lmqmcs_d.

You can run CICS programs using IBM COBOL SET for AIX or Micro Focus COBOL. The following sections describe the difference between these.

Preparing CICS COBOL programs using IBM COBOL SET for AIX

To use IBM COBOL, follow these steps:

1. Export the following environment variable:

```
export LD_FLAGS="-qLIB -bI:/usr/lpp/cics/lib/cicsprIBMCOB.exp \  
-I/usr/mqm/inc -I/usr/lpp/cics/include \  
-e iwz_cobol_main \  
-L/usr/lib/dce -lmqmc_b_r -ldcelibc_r -ldcepthreads"
```

where LIB is a compiler directive.

2. Translate, compile, and link the program by typing:

```
cicstcl -l IBMCOB <yourprog>.ccp
```

Preparing CICS COBOL programs using Micro Focus COBOL

To use Micro Focus COBOL, follow these steps:

1. Add the MQSeries COBOL run-time library module to the run-time library using the following command:

```
cicsmkcobol -L/usr/lib/dce -L/usr/mqm/lib \  
/usr/mqm/lib/libmqmcbrt.o -lmqm_r
```

This creates the Micro Focus COBOL language method file and enables the CICS run-time COBOL library to call MQSeries on UNIX systems.

Note: cicsmkcobol must be run only when one of the following is installed:

- New version or release of Micro Focus COBOL
- New version or release of CICS for AIX
- New version or release of any supported database product (for COBOL transactions only)
- CICS for AIX

2. Export the following environment variables:

```
COBCPY=/usr/mqm/inc export COBCPY  
LD_FLAGS="-L/usr/mqm/lib -lmqm_r" export LD_FLAGS
```

3. Translate, compile, and link the program by typing:

```
cicstcl -l COBOL -e <yourprog>.ccp
```

Preparing CICS C programs

You build CICS C programs using the standard CICS facilities:

1. Export *one* of the following environment variables:
 - LDFLAGS = "-L/usr/mqm/lib -lmqm_r" export LDFLAGS
 - USERLIB = "-L/usr/mqm/lib -lmqm_r" export USERLIB
2. Translate, compile, and link the program by typing:

```
cicstcl -l C amqscic0.ccs
```

CICS C sample transaction: Sample C source for a CICS MQSeries transaction is provided by AMQSCIC0.CCS. The transaction reads messages from the transmission queue SYSTEM.SAMPLE.CICS.WORKQUEUE on the default queue manager and places them onto the local queue whose name is contained in the transmission header of the message. Any failures will be sent to the queue SYSTEM.SAMPLE.CICS.DLQ. The sample MQSC script AMQSCIC0.TST may be used to create these queues and sample input queues.

Chapter 20. Building your application on AS/400

The AS/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 on AS/400 systems. COBOL, C, C++, and RPG programming languages are supported. For information about preparing your C++ programs, see the *MQSeries Using C++* book.

The tasks you must perform to create an executable MQSeries for AS/400 application depend on the programming language the source code is written in. 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 data definition files for the language you are using. You should make yourself familiar with the contents of these files. See “Appendix F. MQSeries data definition files” on page 541 for a full description.

MQSeries for AS/400, V5.1 supports messages up to 100 MB in size, whereas previous releases of MQSeries for AS/400 supported messages up to 4 MB in size. Application programs written in ILE C, supporting MQSeries messages greater than 16 MB, need to make use of the new Teraspace compiler option in order to allocate sufficient memory for these messages.

For further information on the new compiler options, see the *ILE C for AS/400 Programmer’s Guide*.

Preparing C programs

To compile a C module, you can use the OS/400 command, CRTCMOD. Make sure that the library containing the include files (QMQM) is in the library list when you perform the compilation.

You must then bind the output of the compiler with the service program using the CRTPGM command.

An example of the command for a nonthreaded environment is:

Table 14. Example of CRTPGM in the nonthreaded environment

Command	Program/exit type
CRTPGM PGM(<i>pgmname</i>) MODULE(<i>pgmname</i>) BNDSRVPGM(QMQM/LIBMQM)	Server for C

where *pgmname* is the name of your program.

Preparing C programs

An example of the command for a threaded environment is:

Table 15. Example of CRTPGM in the threaded environment

Command	Program/exit type
CRTPGM PGM(<i>pgmname</i>) MODULE(<i>pgmname</i>) BNDSRVPGM(QMQM/LIBMQM_R)	Server for C

where *pgmname* is the name of your program.

Preparing COBOL programs

MQSeries for AS/400 provides two methods for accessing the MQI from within COBOL programs:

1. A dynamic call interface to programs having the names of the MQI functions, such as MQCONN and MQOPEN. This interface is intended primarily for use with the OPM (Original Program Mode) COBOL compiler, but may also be used with the ILE (Integrated Language Environment) COBOL compiler. Some functions new to MQSeries for AS/400 V5.1, such as MQCMIT and MQBACK are not supported through this interface, which is provided for compatibility with previous releases.
2. A bound procedural call interface provided by service programs. This provides access to the new MQI functions in this release, support for threaded applications and possibly, better performance compared with the dynamic call interface. This interface can only be used with the ILE COBOL compiler.

In both cases the standard COBOL CALL syntax is used to access the MQI functions.

The COBOL copy files containing the named constants and structure definitions for use with the MQI are contained in the source physical files QMQM/QLBLSRC and QMQM/QCBLLESRC. The members in these two files are identical, but are packaged twice in this way to correspond with the defaults assumed by the OPM and ILE COBOL compilers respectively.

The COBOL copy files use the single quotation mark character (') as the string delimiter. The AS/400 COBOL compilers assume the delimiter will be the double quote("). To prevent the compilers generating warning messages, specify OPTION(*APOST) on the commands CRTCBLPGM, CRTBNDCBL, or CRTCBLMOD.

To make the compiler accept the single quotation mark character (') as the string delimiter in the COBOL copy files, use the compiler option \APOST.

Using the Dynamic Call Interface

- The QMQM library must be in your library list when you compile and when you run COBOL programs using the MQI dynamic call interface.
- Use the CRTCBLPGM command to invoke the OPM COBOL compiler.
- Use either the CRTBNDCBL command or the two separate commands CRTCBLMOD and CRTPGM to invoke the ILE COBOL compiler.

Using the Bound Procedure Call Interface

- First create a module using the CRTCBLMOD compiler specifying the parameter:
LINKLIT(*PRC)

Preparing COBOL programs

- Then use the CRTPGM command to create the program object specifying the parameter:

for non-threaded applications

```
BNDSRVPGM(QMQM/AMQ0STUB)
```

for threaded applications

```
BNDSRVPGM(QMQM/AMQ0STUB_R)
```

Note: Except for programs created using the V4R4 ILE COBOL compiler and containing the THREAD(SERIALIZE) option in the PROCESS statement, COBOL programs should not use the threaded MQSeries libraries. Even if a COBOL program has been made thread safe in this manner, careful consideration should be given to the overall application design, since THREAD(SERIALIZE) forces serialization of COBOL procedures at the module level and may have an impact on overall performance.

See the *ILE COBOL/400 Programmer's Guide* and *ILE COBOL/400 Reference* for further information.

For more information on compiling a CICS application, see the *CICS for AS/400 Application Programming Guide*, SC33-1386.

Preparing CICS programs

To create a program that includes EXEC CICS statements and MQI calls, perform these steps:

1. If necessary, prepare maps using the CRTCICSMAP command.
2. Translate the EXEC CICS commands into native language statements. Use the CRTCICSC command for a C program. Use the CRTCICSCBL command for a COBOL program.
Include CICSOPT(*NOGEN) in the CRTCICSC or CRTCICSCBL command. This halts processing to enable you to include the appropriate CICS and MQSeries service programs. This command puts the code, by default, into QTEMP/QACYCICS.
3. Compile the source code using the CRTCMOD command (for a C program) or the CRTCBMOD command (for a COBOL program).
4. Use CRTPGM to link the compiled code with the appropriate CICS and MQSeries service programs. This creates the executable program.

An example of such code follows (it compiles the shipped CICS sample program):

```
CRTCICSC OBJ(QTEMP/AMQSCIC0) SRCFILE(/MQSAMP/QCSRC) +  
        SRCMBR(AMQSCIC0) OUTPUT(*PRINT) +  
        CICSOPT(*SOURCE *NOGEN)  
CRTCMOD  MODULE(MQTEST/AMQSCIC0) +  
        SRCFILE(QTEMP/QACYCICS) OUTPUT(*PRINT)  
CRTPGM  PGM(MQTEST/AMQSCIC0) MODULE(MQTEST/AMQSCIC0) +  
        BNDSRVPGM(QMQM/LIBMQIC QCICS/AEGEIPGM)
```

Preparing RPG programs

If you are using MQSeries for AS/400, you can write your applications in RPG. For more information see "Coding in RPG" on page 82, and refer to the *MQSeries for AS/400 Application Programming Reference (ILE RPG)* manual.

SQL programming considerations

If your program contains EXEC SQL statements and MQI calls, perform these steps:

1. Translate the EXEC SQL commands into native language statements. Use the CRTSQLCI command for a C program. Use the CRTSQLCBLI command for a COBOL program.
Include OPTION(*NOGEN) in the CRTSQLCI or CRTSQLCBLI command. This halts processing to enable you to include the appropriate MQSeries service programs. This command puts the code, by default, into QTEMP/QSQLTEMP.
2. Compile the source code using the CRTCMOD command (for a C program) or the CRTCBMOD command (for a COBOL program).
3. Use CRTPGM to link the compiled code with the appropriate MQSeries service programs. This creates the executable program.

An example of such code follows (it compiles a program, SQLTEST, in library, SQLUSER):

```
CRTSQLCI OBJ(MQTEST/SQLTEST) SRCFILE(SQLUSER/QCSRC) +  
          SRCMBR(SQLTEST) OUTPUT(*PRINT) OPTION(*NOGEN)  
CRTCMOD  MODULE(MQTEST/SQLTEST) +  
          SRCFILE(QTEMP/QSQLTEMP) OUTPUT(*PRINT)  
CRTPGM  PGM(MQTEST/SQLTEST) +  
          BNDSRVPGM(QMQM/LIBMQIC)
```

AS/400 programming considerations

If you have compiled programs for releases of MQSeries for AS/400 earlier than V4R4, you will have linked to AMQZSTUB and, possibly, AMQVSTUB. These libraries are provided at this release for compatibility purposes; you do not need to recompile your applications.

These libraries provide support for the default connection handle (MQHC_DEF_HCONN). This is no longer provided by the standard V4R4 libraries. However, the libraries provided at this release for compatibility purposes do not support all new features (for example, MQCONNX, MQCMIT, and MQBACK).

QMOM activation group

When creating your program on AS/400, the QMOM activation group should not be used. The QMOM activation group is for the use of MQSeries only.

Chapter 21. Building your application on AT&T GIS UNIX

This chapter describes the additional tasks, and the changes to the standard tasks, you must perform when building MQSeries for AT&T GIS UNIX applications to run under AT&T GIS UNIX³. C and C++ programming languages are supported.

In addition to coding the MQI calls in your source code, you must add the appropriate include files. You should make yourself familiar with the contents of these files. See “Appendix F. MQSeries data definition files” on page 541 for a full description.

Preparing C programs

Precompiled C programs are found in the `/opt/mqm/samp/bin` directory. To build a sample from source code, use the C compiler in `/bin/cc`, for example:

```
/bin/cc -o <yourprog> <yourprog>.c -lmqm -lmqmcs -lmqmzse \  
-lnet -lnsl -lsocket -ldl
```

Note: The backslash (\) represents the continuation of the line.

C compiler flags

The order of the libraries specified is important. The following is an example of how to build the sample program `amqsput0`:

```
/bin/cc -o <amqsput0> <amqsput0>.c -lmqic -lmqmcs -lmqmzse \  
-lnet -lnsl -lsocket -ldl -lc
```

This links with the client library `-lmqic`, so allows you to use the programs on a machine which has only the MQSeries client for GIS installed.

If you use the other version of the compiler (`/usr/ucb/cc`), your application may compile and link successfully. However when you run it, it will fail when it attempts to connect to the queue manager.

Linking libraries

You need to link your programs with the appropriate library provided by MQSeries.

You must link to one or more of the following libraries:

Library file	Program/exit type
libmqm.so	Server for C
libmqmzse.so	For C
libmqic.so	Client for C
libmqmcs.so	Client for C

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

Preparing C programs

Notes:

1. If you are writing an installable service (see the *MQSeries Programmable System Management* book for further information), you need to link to the `libmqmzf.so` library.
2. If you are producing an XA switch load file for external coordination by an XA-compliant transaction manager such as IBM CICS, Transarc Encina, or Novell Tuxedo, you need to link to the `libmqmx.a` library.

Chapter 22. Building your application on Digital OpenVMS

This chapter describes the additional tasks, and the changes to the standard tasks, you must perform when building MQSeries for Compaq (DIGITAL) OpenVMS applications to run under Digital OpenVMS. C and COBOL are supported.

In addition to coding the MQI calls in your source code, you must add the appropriate include files. You should make yourself familiar with the contents of these files. See “Appendix F. MQSeries data definition files” on page 541 for a full description.

Preparing C programs

This section explains the compiler and libraries you need to prepare your C programs.

C compiler version

You must use the DEC C compiler. To invoke the compiler, enter:

```
$ CC/DECC
```

This is the default.

C compiler flags

The include files for MQSeries for Compaq (DIGITAL) OpenVMS are located in the MQS_INCLUDE directory. The following is an example of how to build the sample program AMQSPUT0:

```
$ CC/INCLUDE_DIRECTORY=MQS_INCLUDE AMQSPUT0  
$ LINK AMQSPUT0.OBJ,SYS$INPUT/OPTIONS  
SYS$SHARE: MQM/SHAREABLE  
Ctrl + Z
```

Linking libraries

You need to link your programs with the appropriate library provided by MQSeries. The libraries are found in SYS\$SHARE.

You must link to one or more of the following libraries:

Library file

Program/exit type

mqm.exe

Server for C

mqic.exe

Client for C

mqmzf.exe

Installable service exits for C

Preparing COBOL programs

This section explains the compiler and libraries you need to prepare your COBOL programs.

Preparing COBOL programs

COBOL compiler flags

You must compile the programs in ANSI mode using the /ANSI switch to the DEC COBOL compiler. The following is an example of how to build the sample program AMQ0PUT0:

```
$ COBOL/ANSI AMQ0PUT0.COB  
$ LINK AMQ0PUT0.OBJ,SYSS$INPUT/OPTIONS  
SYSS$SHARE: MQMCB/SHAREABLE  
Ctrl + Z
```

Linking libraries

You need to link your program with one of the following:

MQMCB.EXE

COBOL

MQICB.EXE

COBOL MQSeries client

Chapter 23. Building your application on Compaq Tru64 UNIX

This chapter describes how to build application programs to run under Compaq Tru64 UNIX.

It contains the following sections:

- “Building applications in C on Compaq Tru64 UNIX Version 4.0”
- “Building applications in C on Compaq Tru64 UNIX Version 5.0” on page 300
- “Building applications in C++ on Compaq Tru64 UNIX Version 4.0” on page 301
- “Building applications in C++ on Compaq Tru64 UNIX Version 5.0” on page 302
- “Building applications in COBOL” on page 303
- “Building applications in Java” on page 304

Throughout this chapter the \ character is used to split long commands over more than one line. Do not enter this character, enter each command as a single line.

Building applications in C on Compaq Tru64 UNIX Version 4.0

This section describes how to build application programs written in C to run under Compaq Tru64 UNIX Version 4.0.

C language include files

The MQSeries C include files are listed in Table 16. They are installed in the directory /opt/mqm/inc/. The include files are symbolically linked into /usr/include.

Table 16. C include files for MQSeries (Compaq Tru64 UNIX Version 4.0)

File name	Contents
<cmqc.h>	Call prototypes, data types, structures, return codes, and constants
<cmqfc.h>	Definitions for programmable commands
<cmqxc.h>	Definitions for channel exits and data-conversion exits
<cmqzc.h>	Definitions for installable services exits
Note: The files are protected against multiple declaration, therefore you can include them many times.	

Preparing C programs

Work in your usual environment. Precompiled C programs are supplied in the /opt/mqm/samp/bin directory. Here is an example instruction for building the sample program amqsput0.c in a nonthreaded environment:

```
$ cc -std1 -o amqsput amqsput0.c -lmqm
```

Here is an example instruction for building the sample program amqsput0.c in a threaded environment:

```
$ cc -std1 -pthread -o amqsput amqsput0.c -lmqm_r
```

Building applications in C

If you want to use the programs on a machine on which only the MQSeries client for Compaq Tru64 UNIX is installed, recompile the programs to link them with the client library. Here is an example instruction for building a nonthreaded client:

```
$ cc -std1 -o amqsput amqsput0.c -lmqmic
```

Here is an example instruction for building a threaded client:

```
$ cc -std1 -pthread -o amqsput amqsput0.c -lmqmic_r
```

Linking libraries

In a nonthreaded environment, you must link to one of the following libraries:

Library file	Program or exit type
libmqm.so	Server for C
libmqic.so	Client for C

In a threaded environment, you must link to one of the following libraries:

Library file	Program or exit type
libmqm_r.so	Server for C
libmqic_r.so	Client for C

Note: If you are writing an installable service (as described in the *MQSeries Programmable System Management* book), you need to link to the `libmqmzf.so` library.

Building applications in C on Compaq Tru64 UNIX Version 5.0

This section describes how to build application programs written in C to run under Compaq Tru64 UNIX Version 5.0.

Attention

Only one set of libraries is supplied with MQSeries for Compaq Tru64 UNIX, V5.1, on Compaq Tru64 UNIX Version 5.0. You can build threaded or nonthreaded applications by linking with these libraries.

C language include files

The MQSeries C include files are listed in Table 17. They are installed in the directory `/opt/mqm/inc/`. The include files are symbolically linked into `/usr/include`.

Table 17. C include files for MQSeries (Compaq Tru64 UNIX Version 5.0)

File name	Contents
<code><cmqc.h></code>	Call prototypes, data types, structures, return codes, and constants
<code><cmqfc.h></code>	Definitions for programmable commands
<code><cmqxc.h></code>	Definitions for channel exits and data-conversion exits
<code><cmqzc.h></code>	Definitions for installable services exits

Table 17. C include files for MQSeries (Compaq Tru64 UNIX Version 5.0) (continued)

File name	Contents
Note: The files are protected against multiple declaration, therefore you can include them many times.	

Preparing C programs

Work in your usual environment. Precompiled C programs are supplied in the /opt/mqm/samp/bin directory. Here is an example instruction for building the sample program amqsput0.c:

```
$ cc -std1 -o -pthread amqsput amqsput0.c -lmqm
```

If you want to use the programs on a machine on which only the MQSeries client for Compaq Tru64 UNIX is installed, recompile the programs to link them with the client library. Here is an example instruction for building a client:

```
$ cc -std1 -o -pthread amqsput amqsput0.c -lmqmic
```

Linking libraries

You must link to one of the following libraries:

Library file	Program or exit type
libmqm.so	Server for C
libmqic.so	Client for C

Building applications in C++ on Compaq Tru64 UNIX Version 4.0

This section describes how to build application programs written in C++ to run under Compaq Tru64 UNIX Version 4.0.

For more information on using MQSeries with C++, see the *MQSeries Using C++* book.

C++ language include files

The MQSeries C++ include files are listed in Table 18. They are installed in the directory /opt/mqm/inc/. The include files are symbolically linked into /usr/include.

Table 18. C++ include files for MQSeries (Compaq Tru64 UNIX Version 4.0)

File name	Contents
<cmqc.h>	MQI data structures and manifest constants
<imqi.hpp>	C++ MQI classes (includes cmqc.h and imqtype.h)
<imqtype.h>	Defines the ImqBoolean data type
Note: The files are protected against multiple declaration, therefore you can include them many times.	

Building applications in C++

Preparing C++ programs

Work in your usual environment. Precompiled C++ programs are supplied in the `/opt/mqm/samp/bin/ff` directory. Here is an example instruction for building the sample program `imqspu0.cpp` in a nonthreaded environment:

```
$ cxx -std1 -o imqspu0 imqspu0.cpp -limqs23ff -limqb23ff
```

Here is an example instruction for building the sample program `imqspu0.cpp` in a threaded environment:

```
$ cxx -std1 -pthread -o imqspu0 imqspu0.cpp -limqs23ff_r -limqb23ff_r
```

If you want to use the programs on a machine on which only the MQSeries client for Compaq Tru64 UNIX is installed, recompile the programs to link them with the client library. Here is an example instruction for building the sample program `imqspu0.ccp` as a nonthreaded client:

```
$ cxx -std1 -o imqspu0 imqspu0.cpp -limqc23ff -limqb23ff
```

Here is an example instruction for building the sample program `imqspu0.cpp` as a threaded client:

```
$ cxx -std1 -pthread -o imqspu0 imqspu0.cpp -limqc23ff_r -limqb23ff_r
```

Linking libraries

In a nonthreaded environment, you must link to one of the following libraries:

Library file	Program or exit type
<code>libimqs23ff.so</code>	Server for C++
<code>libimqc23ff.so</code>	Client for C++

In a threaded environment, you must link to one of the following libraries:

Library file	Program or exit type
<code>libimqs23ff_r.so</code>	Server for C++
<code>libimqc23ff_r.so</code>	Client for C++

Note: If you are writing an installable service (as described in the *MQSeries Programmable System Management* book), you need to link to the `libmqmzf.so` library.

Building applications in C++ on Compaq Tru64 UNIX Version 5.0

This section describes how to build application programs written in C++ to run under Compaq Tru64 UNIX Version 5.0.

For more information on using MQSeries with C++, see the *MQSeries Using C++* book.

C++ language include files

The MQSeries C++ include files are listed in Table 19 on page 303. They are installed in the directory `/opt/mqm/inc/`. The include files are symbolically linked into `/usr/include`.

Table 19. C++ include files for MQSeries (Compaq Tru64 UNIX Version 5.0)

File name	Contents
<cmqc.h>	MQI data structures and manifest constants
<imqi.hpp>	C++ MQI classes (includes cmqc.h and imqtype.h)
<imqtype.h>	Defines the ImqBoolean data type
Note: The files are protected against multiple declaration, therefore you can include them many times.	

Preparing C++ programs

Work in your usual environment. Precompiled C++ programs are supplied in the /opt/mqm/samp/bin/ff directory. Here is an example instruction for building the sample program imqspu0.cpp:

```
$ cxx -pthread -o imqspu0 imqspu0.cpp -limqs23ff -limqb23ff
```

If you want to use the programs on a machine on which only the MQSeries client for Compaq Tru64 UNIX is installed, recompile the programs to link them with the client library. Here is an example instruction for building the sample program imqspu0.cpp as a client:

```
$ cxx -pthread -o imqspu0 imqspu0.cpp -limqc23ff -limqb23ff
```

Linking libraries

You must link to one of the following libraries:

Library file	Program or exit type
libimqs23ff.so	Server for C++
libimqc23ff.so	Client for C++

Note: If you are writing an installable service (as described in the *MQSeries Programmable System Management* book), you need to link to the libmqmzf.so library.

Building applications in COBOL

This section describes how to build application programs written in COBOL to run under Compaq Tru64 UNIX.

For information on the MQSeries COBOL copy files, see "COBOL copy files" on page 543.

Preparing COBOL programs

Use the Micro Focus COBOL compiler to compile your programs. The copy files that declare the structures are in the /opt/mqm/inc directory.

See the Micro Focus COBOL compiler documentation for a description of the environment variables that you must set up.

```
$ export COBDIR=/opt/cobol
$ export COBCPY=/opt/mqm/inc
$ export LD_LIBRARY_PATH=/opt/mqm/lib:$LD_LIBRARY_PATH
```

Building applications in COBOL

Here is an example instruction for building the sample program amq0put0.cbl in a nonthreaded environment:

```
$ cob -vxP amq0put0.cbl -lmqmcb
```

Here is an example instruction for building the sample program amq0put0.cbl in a threaded environment:

```
$ cob -vxP amq0put0.cbl -lmqmcb_r
```

If you want to use the programs on a machine on which only the MQSeries client for Compaq Tru64 UNIX is installed, recompile the programs to link them with the client library. Here is an example instruction for building a nonthreaded client:

```
$ cob -vxP amq0put0.cbl -lmqmicb
```

Here is an example instruction for building a threaded client:

```
$ cob -vxP amq0put0.cbl -lmqmicb_r
```

Linking libraries

You must link to one of the following libraries:

Library file	Program or exit type
libmqmcb.so	Server for COBOL
libmqmicb.so	Client for COBOL
libmqmcb_r.so	Threaded applications

Building applications in Java

This section describes how to build application programs written in Java to run under Compaq Tru64 UNIX.

Preparing Java programs

Make sure that your MQSeries Client for Java or MQSeries Bindings for Java installation directory is in your CLASSPATH environment variable. For example:

```
CLASSPATH=/usr/opt/jdk118/lib/classes.zip:/opt/mqm/java/lib: \  
/opt/mqm/samp/mqbind/En_US:.
```

The following environment variable is required:

```
LD_LIBRARY_PATH=/opt/mqm/lib
```

To compile the class MyClass.java, for example, use the command:

```
$ javac MyClass.java
```

Note: If your MQSeries Java program handles large messages, you must increase the maximum Java heap size appropriately using the `-mx` option of the `java` command.

For more information on using Java with MQSeries, see the *MQSeries Using Java* book.

Chapter 24. Building your application on HP-UX

This chapter describes the additional tasks, and the changes to the standard tasks, you must perform when building MQSeries for HP-UX applications to run under HP-UX. C, C++, and COBOL are supported. For information about preparing your C++ programs, see the *MQSeries Using C++* book.

The tasks you must perform to create an executable application using MQSeries for HP-UX vary with the programming language your source code is written in. In addition to coding the MQI calls in your source code, you must add the appropriate language statements to include the MQSeries for HP-UX include files for the language you are using. You should make yourself familiar with the contents of these files. See “Appendix F. MQSeries data definition files” on page 541 for a full description.

Throughout this chapter the \ character is used to split long commands over more than one line. Do not enter this character, enter each command as a single line.

Preparing C programs

Work in your normal environment. Precompiled C programs are supplied in the /opt/mqm/samp/bin directory.

Preparing C programs on HP-UX V10.20

The following is an example of how to build the sample program amqsput0 in a non-threaded environment:

```
cc -Aa -D_HPUX_SOURCE -o amqsput0 amqsput0.c -lmqm
```

The following is an example of how to build the sample program amqsput0 in a threaded environment:

```
cc -Aa -D_HPUX_SOURCE -o amqsput0 amqsput0.c -lmqm_r -lcma
```

If you want to use the programs on a machine that has only the MQSeries client for HP-UX installed, recompile the programs to link them with the client library instead. The following is an example of how to build a non-threaded client:

```
cc -Aa -D_HPUX_SOURCE -o amqsput0 amqsput0.c -lmqic
```

The following is an example of how to build a threaded client:

```
cc -Aa -D_HPUX_SOURCE -o amqsput0 amqsput0.c -lmqic_r
```

Note: If you are building an application that uses the curses screen display library, you must explicitly link libC following libmqm and place libcurses at the end of the link order.

The following is an example of how to build a program that links with MQSeries curses:

```
cc -Aa -D_HPUX_SOURCE -o mqcourse mqcourse.c -lmqm -lc -lcurses
```

Preparing C programs

Preparing C programs on HP-UX V11.00

The following is an example of how to build the sample program `amqsput0` in a non-threaded environment:

```
cc -Aa -D_HPUX_SOURCE -o amqsput0 amqsput0.c -lmqm
```

The following is an example of how to build the sample program `amqsput0` in a POSIX draft 10 threaded environment:

```
cc -Aa -D_HPUX_SOURCE -o amqsput0 amqsput0.c -lmqm_r -lpthread
```

The following is an example of how to build the sample program `amqsput0` in a POSIX draft 4 (DCE) threaded environment:

```
cc -Aa -D_HPUX_SOURCE -D_PTHREADS_DRAFT -o amqsput0 \
amqsput0.c -lmqm_d -ld4r -lcma
```

If you want to use the programs on a machine that has only the MQSeries client for HP-UX installed, recompile the programs to link them with the client library instead. The following is an example of how to build a non-threaded client:

```
cc -Aa -D_HPUX_SOURCE -o amqsput0 amqsput0.c -lmqic
```

The following is an example of how to build a POSIX draft 10 threaded client:

```
cc -Aa -D_HPUX_SOURCE -o amqsput0 amqsput0.c -lmqic_r -lpthread
```

The following is an example of how to build a POSIX draft 4 (DCE) threaded client:

```
cc -Aa -D_HPUX_SOURCE -D_PTHREADS_DRAFT4 -o amqsput0 \
amqsput0.c -lmqic_d -ld4r -lcma
```

Linking libraries

The following lists the libraries you will need.

- You need to link your programs with the appropriate library provided by MQSeries.

In a non-threaded environment, you must link to one of the following libraries:

Library file

Program/exit type

libmqm.sl

Server for C

libmqic.sl

Client for C

In a threaded environment, you must link to one of the following libraries:

Library file

Program/exit type

libmqm_r.sl

Server for C

libmqic_r.sl

Client for C

In a POSIX draft 10 threaded environment on HP-UX V11.00, you must link to one of the following libraries:

Library file

Program/exit type

libmqm_r.sl

Server for C

libmqic_r.sl
Client for C

In a POSIX draft 4 (DCE) threaded environment on HP-UX V11.00, you must link to one of the following libraries:

Library file
Program/exit type

libmqm_d.sl
Server for C

libmqic_d.sl
Client for C

Notes:

1. If you are writing an installable service (see the *MQSeries Programmable System Management* book for further information), you need to link to the `libmqmzf.sl` library.
2. If you are producing an XA switch load file for external coordination by an XA-compliant transaction manager such as IBM CICS, Transarc Encina, or Novell Tuxedo, you need to link to the `libmqmxa.a` library in a non-threaded application and to the `libmqmxa_r.a` library in a threaded application.
3. You must link MQSeries libraries before any other product libraries (in this case, DCE and Encina). This ensures that any operating system functions that have been redefined by DCE are also used by MQSeries.

Preparing COBOL programs

Compile the programs using the Micro Focus compiler. The copy files which declare the structures are in `/opt/mqm/inc`:

```
$ export LIB=/usr/mqm/lib:$LIB
$ export COBCPY="/opt/mqm/inc"
$ cob -vxP <amqsput>.cbl -lmqmb
```

where `amqsput` is a sample program.

You must ensure that you have specified adequate run-time stack sizes; 16 KB is the recommended minimum.

You need to link your program with one of the following:

libmqmcb.sl
Server for COBOL

libmqicb.sl
Client for COBOL

amqmb_r.sl
Threaded applications

Programs to run in the MQSeries client environment

If you are using LU 6.2 to connect your MQI client to a server, you must link your application to `libsna.a`, part of the SNAPplusAPI product. Use the `-lv3` and `-lstr` options on your compile and link command.

- The `-lv3` option gives your program access to the AT&T signaling library (the SNAPplusAPI uses AT&T signals)
- The `-lstr` option links your program to the streams component

Note: The `-lstr` option is not needed on HP-UX Version 10.

Preparing COBOL programs

If you are not using LU 6.2, consider linking to `libsastubs.a` (in `/opt/mqm/lib`) to fully resolve function names. The need to link to this library varies with how you are using the `-B` flag during the linking stage.

Preparing CICS programs

To build the sample CICS transaction, `amqscic0.ccs`, run the following command:

```
$ export USERLIB="-lmqm_r"
$ cicstcl -l C amqscic0.ccs
```

An XA switch module is provided to enable you to link CICS with MQSeries:

Table 20. Essential Code for CICS applications (HP-UX)

Description	C (source)	C (exec)
XA initialization routine	<code>amqzscix.c</code>	<code>amqzsc</code>

You can find more information about supporting CICS transactions in the *MQSeries System Administration* book.

CICS on Open Systems support

MQSeries on UNIX systems supports CICS on Open Systems via the XA interface.

It is very important to ensure that CICS COBOL applications are linked to the threaded version of the library. CICS on Open Systems MQSeries transactions must link with `libmqm_r`, except on Sun Solaris, where you must link with `lmqmc_d`.

CICS C sample transaction

Sample C source for a CICS MQSeries transaction is provided by `AMQSCIC0.CCS`. The transaction reads messages from the transmission queue `SYSTEM.SAMPLE.CICS.WORKQUEUE` on the default queue manager and places them onto the local queue whose name is contained in the transmission header of the message. Any failures will be sent to the queue `SYSTEM.SAMPLE.CICS.DLQ`. The sample MQSC script `AMQSCIC0.TST` may be used to create these queues and sample input queues.

Preparing CICS COBOL programs using Micro Focus COBOL

To use Micro Focus COBOL, follow these steps:

1. Add the MQSeries COBOL run-time library module to the run-time library using the following command:

```
cicsmkcobol -L/usr/lib/dce -L/usr/mqm/lib \
            /usr/mqm/lib/ libmqmcbt.o -lmqm_r
```

This creates the Micro Focus COBOL language method file and enables the CICS run-time COBOL library to call MQSeries on UNIX systems.

Note: `cicsmkcobol` must be run only when one of the following is installed:

- New version or release of Micro Focus COBOL
 - New version or release of TXSeries for HP-UX
 - New version or release of any supported database product (for COBOL transactions only)
 - TXSeries for HP-UX
2. Export the following environment variables:

```
COBCPY=/usr/mqm/inc export COBCPY
LDLFLAGS="-L/usr/mqm/lib -lmqm_r" export LDLFLAGS
```


3. Translate, compile, and link the program by typing:
`cicstcl -l COBOL -e <yourprog>.ccp`

Preparing CICS programs

Chapter 25. Building your application on OS/390

The CICS, IMS, and OS/390 publications describe how to build applications that run in these environments. This chapter describes the additional tasks, and the changes to the standard tasks, you must perform when building MQSeries for OS/390 applications for these environments. COBOL, C, C++, Assembler, and PL/I programming languages are supported. (For information on building C++ applications see the *MQSeries Using C++* book.)

The tasks you must perform to create an executable MQSeries for OS/390 application depend on both the programming language the program is written in, and the environment in which the application will run.

In addition to coding the MQI calls in your program, you must add the appropriate language statements to include the MQSeries for OS/390 data definition file for the language you are using. You should make yourself familiar with the contents of these files. See “Appendix F. MQSeries data definition files” on page 541 for a full description.

Note

The name **thlqual** is the high-level qualifier of the installation library on OS/390.

This chapter introduces building OS/390 applications, under these headings:

- “Preparing your program to run”
- “Dynamically calling the MQSeries stub” on page 315
- “Debugging your programs” on page 319

Preparing your program to run

After you have written the program for your MQSeries application, to create an executable application you have to compile or assemble it, then link-edit the resulting object code with the stub program that MQSeries for OS/390 supplies for each environment it supports. How you prepare your program depends on both the environment (batch, CICS, or IMS(BMP or MPP)) in which the application will run, and the structure of the data sets on your OS/390 installation. The details are described in the following sections.

“Dynamically calling the MQSeries stub” on page 315 describes an alternative method of making MQI calls in your programs so that you do not need to link-edit an MQSeries stub. This method is not available for all languages and environments.

Do not link-edit a higher level of stub program than that of the version of MQSeries for OS/390 on which your program is running. For example, a program running on MQSeries for MVS/ESA, V1.2 must not be link-edited with a stub program supplied with MQSeries for OS/390, V5.2.

Building batch applications

Building OS/390 batch applications

To build an MQSeries for OS/390 application that runs under OS/390 batch, create job control language (JCL) that performs these tasks:

1. Compile (or assemble) the program to produce object code. The JCL for your compilation must include SYSLIB statements that make the product data definition files available to the compiler. The data definitions are supplied in the following MQSeries for OS/390 libraries:
 - For COBOL, **thlqual.SCSQCOBC**
 - For assembler language, **thlqual.SCSQMACS**
 - For C, **thlqual.SCSQC370**
 - For PL/I, **thlqual.SCSQPLIC**
2. For a C application, prelink the object code created in step 1.
3. Link-edit the object code created in step 1 (or step 2 for a C application) to produce a load module. When you link-edit the code, you must include one of the MQSeries for OS/390 batch stub programs (CSQBSTUB or one of the RRS stub programs: CSQBRRSI or CSQBRSTB).

CSQBSTUB

single-phase commit provided by MQSeries for OS/390

CSQBRRSI

two-phase commit provided by RRS via the MQI

CSQBRSTB

two-phase commit provided by RRS directly

Note: If you use CSQBRSTB then you must also link-edit your application with ATRSCSS from SYS1.CSSLIB. Figure 19 and Figure 20 on page 313 show fragments of JCL to do this. The stubs are language-independent and are supplied in library **thlqual.SCSQLOAD**.

4. Store the load module in an application load library.

```
      :
/*
/* MQSERIES FOR OS/390 LIBRARY CONTAINING BATCH STUB
/*
//CSQSTUB DD DSN=++HLQ.MQM100++.SCSQLOAD,DISP=SHR
/*
      :
//SYSIN DD *
INCLUDE CSQSTUB(CSQBSTUB)
      :
/*
```

Figure 19. Fragments of JCL to link-edit the object module in the batch environment, using single-phase commit

```

:
/*
/* MQSERIES FOR OS/390 LIBRARY CONTAINING BATCH STUB
/*
//CSQSTUB DD DSN=++HLQ.MQM100++.SCSQLOAD,DISP=SHR
//CSSLIB DD DSN=SYS1.CSSLIB,DISP=SHR
/*
:
//SYSIN DD *
INCLUDE CSQSTUB(CSQBRSTB)
INCLUDE CSSLIB(ATRSCSS)
:
/*

```

Figure 20. Fragments of JCL to link-edit the object module in the batch environment, using two-phase commit

To run a batch or RRS program, you must include the libraries **thlqual.SCSQAUTH** and **thlqual.SCSQLOAD** in the STEPLIB or JOBLIB data set concatenation.

To run a TSO program, you must include the libraries **thlqual.SCSQAUTH** and **thlqual.SCSQLOAD** in the STEPLIB used by the TSO session.

To run an OpenEdition® batch program from the OpenEdition shell, add the libraries **thlqual.SCSQAUTH** and **thlqual.SCSQLOAD** to the STEPLIB specification in your \$HOME/.profile like this:

```

STEPLIB=thlqual1.SCSQAUTH:thlqual1.SCSQLOAD
export STEPLIB

```

Building CICS applications

To build an MQSeries for OS/390 application that runs under CICS, you must:

- Translate the CICS commands in your program into the language in which the rest of your program is written
- Compile or assemble the output from the translator to produce object code
- Link-edit the object code to create a load module

CICS provides a procedure to execute these steps in sequence for each of the programming languages it supports.

- For CICS Transaction Server for OS/390, the *CICS Transaction Server for OS/390 System Definition Guide* describes how to use these procedures and the *CICS/ESA® Application Programming Guide* gives more information on the translation process.

You must include:

- In the SYSLIB statement of the compilation (or assembly) stage, statements that make the product data definition files available to the compiler. The data definitions are supplied in the following MQSeries for OS/390 libraries:
 - For COBOL, **thlqual.SCSQCOBC**
 - For assembler language, **thlqual.SCSQMACS**
 - For C, **thlqual.SCSQC370**
 - For PL/I, **thlqual.SCSQPLIC**
- In your link-edit JCL, the MQSeries for OS/390 CICS stub program (CSQCSTUB). Figure 21 on page 314 shows fragments of JCL code to do this. The stub is language-independent and is supplied in library **thlqual.SCSQLOAD**.

Building CICS applications

When you have completed these steps, store the load module in an application load library and define the program to CICS in the usual way.

```

:
/*
/* MQSERIES FOR OS/390 LIBRARY CONTAINING CICS STUB
/*
//CSQSTUB DD DSN=++HLQ.MQM100++.SCSQLOAD,DISP=SHR
/*
:
//LKED.SYSIN DD *
INCLUDE CSQSTUB(CSQSTUB)
:
/*
```

Figure 21. Fragments of JCL to link-edit the object module in the CICS environment

Before you run a CICS program, your system administrator must define it to CICS as an MQSeries program and transaction: you can then run it in the usual way.

Building IMS (BMP or MPP) applications

If you are building batch DL/I programs, see “Building OS/390 batch applications” on page 312. To build other applications that run under IMS (either as a BMP or an MPP), create JCL that performs these tasks:

1. Compile (or assemble) the program to produce object code. The JCL for your compilation must include SYSLIB statements that make the product data definition files available to the compiler. The data definitions are supplied in the following MQSeries for OS/390 libraries:
 - For COBOL, **thlqual.SCSQCOBC**
 - For assembler language, **thlqual.SCSQMACS**
 - For C, **thlqual.SCSQC370**
 - For PL/I, **thlqual.SCSQPLIC**
2. For a C application, prelink the object module created in step 1.
3. Link-edit the object code created in step 1 (or step 2 for a C/370 application) to produce a load module:
 - a. Include the IMS language interface module (DFSLI000).
 - b. Include the MQSeries for OS/390 IMS stub program (CSQQSTUB).
Figure 22 on page 315 shows fragments of JCL to do this. The stub is language independent and is supplied in library **thlqual.SCSQLOAD**.

Note: If you are using COBOL, you should select the NODYNAM compiler option to enable the linkage editor to resolve references to CSQQSTUB unless you intend to use dynamic linking as described in “Dynamically calling the MQSeries stub” on page 315.

4. Store the load module in an application load library.

```

:
/*
/* MQSERIES FOR OS/390 LIBRARY CONTAINING IMS STUB
/*
//CSQSTUB DD DSN=++HLQ.MQM100++.SCSQLOAD,DISP=SHR
/*
:
//LKED.SYSIN DD *
INCLUDE CSQSTUB(CSQSTUB)
:
/*

```

Figure 22. Fragments of JCL to link-edit the object module in the IMS environment

Before you run an IMS program, your system administrator must define it to IMS as an MQSeries program and transaction: you can then run it in the usual way.

Dynamically calling the MQSeries stub

Instead of link-editing the MQSeries stub program with your object code, you can dynamically call the stub from within your program. You can do this in the batch, IMS, and CICS environments. This facility is not supported by programs using PL/I in the CICS environment and it is not supported in the RRS environment.

However, this method:

- Increases the complexity of your programs
- Increases the storage required by your programs at execution time
- Reduces the performance of your programs
- Means that you cannot use the same programs in other environments

If you call the stub dynamically, the appropriate stub program and its aliases must be available at execution time. To ensure this, include the MQSeries for OS/390 data set SCSQLOAD:

For batch and IMS	In the STEPLIB concatenation of the JCL
For CICS	In the CICS DFHRPL concatenation

For IMS, you must ensure that the library containing the dynamic stub (built as described in the information about installing the IMS adapter in the *MQSeries for OS/390 System Setup Guide*) is ahead of the data set SCSQLOAD in the STEPLIB concatenation of the region JCL.

Use the names shown in Table 21 when you call the stub dynamically. In PL/I, only declare the call names used in your program.

Table 21. Call names for dynamic linking

MQI call	Dynamic call name		
	Batch (non-RRS)	CICS	IMS
MQBACK	CSQBBACK	not supported	not supported
MQCMIT	CSQBCOMM	not supported	not supported
MQCLOSE	CSQBCLOS	CSQCCLOS	MQCLOSE
MQCONN	CSQBCONN	CSQCCONN	MQCONN
MQCONNX	CSQBCONX	CSQCCONX	MQCONNX

Calling the MQSeries stub

Table 21. Call names for dynamic linking (continued)

MQI call	Dynamic call name		
MQDISC	CSQBDISC	CSQCDISC	MQDISC
MQGET	CSQBGET	CSQCGET	MQGET
MQINQ	CSQBINQ	CSQCINQ	MQINQ
MQOPEN	CSQBOPEN	CSQCOPEN	MQOPEN
MQPUT	CSQBPUT	CSQCPUT	MQPUT
MQPUT1	CSQBPUT1	CSQCPUT1	MQPUT1
MQSET	CSQBSET	CSQCSET	MQSET

For examples of how to use this technique, see the following figures:

Batch and COBOL	Figure 23
CICS and COBOL	Figure 24 on page 317
IMS and COBOL	Figure 25 on page 317
Batch and assembler	Figure 26 on page 317
CICS and assembler	Figure 27 on page 318
IMS and assembler	Figure 28 on page 318
Batch and C	Figure 29 on page 318
CICS and C	Figure 30 on page 318
IMS and C	Figure 31 on page 319
Batch and PL/I	Figure 32 on page 319
IMS and PL/I	Figure 33 on page 319

```

:
:   WORKING-STORAGE SECTION.
:
:       05 WS-MQOPEN                PIC X(8) VALUE 'CSQBOPEN'.
:
:   PROCEDURE DIVISION.
:
:       CALL WS-MQOPEN WS-HCONN
:           MQOD
:           WS-OPTIONS
:           WS-HOBJ
:           WS-COMPCODE
:           WS-REASON.
:
:

```

Figure 23. Dynamic linking using COBOL in the batch environment

Calling the MQSeries stub

```

:
: WORKING-STORAGE SECTION.
:
:   05 WS-MQOPEN                PIC X(8) VALUE 'CSQCOPEN'.
:
: PROCEDURE DIVISION.
:
:   CALL WS-MQOPEN WS-HCONN
:           MQOD
:           WS-OPTIONS
:           WS-HOBJ
:           WS-COMPCODE
:           WS-REASON.
:
:
```

Figure 24. Dynamic linking using COBOL in the CICS environment

```

:
: WORKING-STORAGE SECTION.
:
:   05 WS-MQOPEN                PIC X(8) VALUE 'MQOPEN'.
:
: PROCEDURE DIVISION.
:
:   CALL WS-MQOPEN WS-HCONN
:           MQOD
:           WS-OPTIONS
:           WS-HOBJ
:           WS-COMPCODE
:           WS-REASON.
:
:
: * ----- *
: *
: *   If the compile option 'DYNAM' is specified
: *   then you may code the MQ calls as follows
: *
: * ----- *
:
:   CALL 'MQOPEN' WS-HCONN
:           MQOD
:           WS-OPTIONS
:           WS-HOBJ
:           WS-COMPCODE
:           WS-REASON.
:
:
```

Figure 25. Dynamic linking using COBOL in the IMS environment

```

:
:   LOAD   EP=CSQBOPEN
:
:   CALL  (15), (HCONN, MQOD, OPTIONS, HOBJ, COMPCODE, REASON), VL
:
:   DELETE EP=CSQBOPEN
:
:
```

Figure 26. Dynamic linking using assembler language in the batch environment

Calling the MQSeries stub

```
⋮
EXEC CICS LOAD PROGRAM('CSQCOPEN') ENTRY(R15)
⋮
CALL (15), (HCONN, MQOD, OPTIONS, HOBJ, COMPCODE, REASON), VL
⋮
EXEC CICS RELEASE PROGRAM('CSQCOPEN')
⋮
```

Figure 27. Dynamic linking using assembler language in the CICS environment

```
⋮
LOAD EP=MQOPEN
⋮
CALL (15), (HCONN, MQOD, OPTIONS, HOBJ, COMPCODE, REASON), VL
⋮
DELETE EP=MQOPEN
⋮
```

Figure 28. Dynamic linking using assembler language in the IMS environment

```
⋮
typedef void CALL_ME();
#pragma linkage(CALL_ME, OS)
⋮
main()
{
CALL_ME * csqbopen;
⋮
csqbopen = (CALL_ME *) fetch("CSQBOPEN");
(*csqbopen)(Hconn, &ObjDesc, Options, &Hobj, &CompCode, &Reason);
⋮
}
```

Figure 29. Dynamic linking using C language in the batch environment

```
⋮
typedef void CALL_ME();
#pragma linkage(CALL_ME, OS)
⋮
main()
{
CALL_ME * csqcopen;
⋮
EXEC CICS LOAD PROGRAM("CSQCOPEN") ENTRY(csqcopen);
(*csqcopen)(Hconn, &ObjDesc, Options, &Hobj, &CompCode, &Reason);
⋮
}
```

Figure 30. Dynamic linking using C language in the CICS environment

```

:
typedef void CALL_ME();
#pragma linkage(CALL_ME, OS)
:
main()
{
CALL_ME * mqopen;
:
mqopen = (CALL_ME *) fetch("MQOPEN");
(*mqopen)(Hconn,&ObjDesc,Options,&Hobj,&CompCode,&Reason);
:
:

```

Figure 31. Dynamic linking using C language in the IMS environment

```

:
DCL CSQBOPEN ENTRY EXT OPTIONS(ASSEMBLER INTER);
:
FETCH CSQBOPEN;

CALL CSQBOPEN(HQM,
              MQOD,
              OPTIONS,
              HOBJ,
              COMPCODE,
              REASON);

RELEASE CSQBOPEN;

```

Figure 32. Dynamic linking using PL/I in the batch environment

```

:
DCL MQOPEN ENTRY EXT OPTIONS(ASSEMBLER INTER);
:
FETCH MQOPEN;

CALL MQOPEN(HQM,
            MQOD,
            OPTIONS,
            HOBJ,
            COMPCODE,
            REASON);

RELEASE MQOPEN;

```

Figure 33. Dynamic linking using PL/I in the IMS environment

Debugging your programs

The main aids to debugging MQSeries for OS/390 application programs are the reason codes returned by each API call. See the *MQSeries Application Programming Reference* manual for a list of these and for more information, including suggestions for corrective action.

This chapter also suggests other debugging tools that you may want to use in particular environments.

Debugging programs

Debugging CICS programs

You can use the CICS Execution Diagnostic Facility (CEDF) to test your CICS programs interactively without having to modify the program or program-preparation procedure. For more information about EDF, see the *CICS Transaction Server for OS/390 CICS Application Programming Guide*.

CICS trace

You will probably also find it helpful to use the CICS Trace Control transaction (CETR) to control CICS trace activity. For more information about CETR, see the *CICS Transaction Server for OS/390 CICS-Supplied Transactions* manual.

To determine whether CICS trace is active, display connection status using the CKQC panel. This panel also shows the trace number.

To interpret CICS trace entries, see Table 22.

The CICS trace entry for these values is AP0xxx (where xxx is the trace number specified when the CICS adapter was enabled). All trace entries except CSQCTEST are issued by CSQCTRUE. CSQCTEST is issued by CSQCRST and CSQCDSP.

Table 22. CICS adapter trace entries

Name	Description	Trace sequence	Trace data
CSQCABNT	Abnormal termination	Before issuing END_THREAD ABNORMAL to MQSeries. This is due to the end of the task and therefore an implicit backout could be performed by the application. A ROLLBACK request is included in the END_THREAD call in this case.	Unit of work information. You can use this information when finding out about the status of work. (For example, it can be verified against the output produced by the DISPLAY THREAD command, or the MQSeries for OS/390 log print utility.)
CSQCBACK	Syncpoint backout	Before issuing BACKOUT to MQSeries for OS/390. This is due to an explicit backout request from the application.	Unit of work information.
CSQCCRC	Completion code and reason code	After unsuccessful return from API call.	Completion code and reason code.
CSQCCOMM	Syncpoint commit	Before issuing COMMIT to MQSeries for OS/390. This can be due to a single-phase commit request or the second phase of a two-phase commit request. The request is due to an explicit syncpoint request from the application.	Unit of work information.
CSQCEXER	Execute resolve	Before issuing EXECUTE_RESOLVE to MQSeries for OS/390.	The unit of work information of the unit of work issuing the EXECUTE_RESOLVE. This is the last indoubt unit of work in the resynchronization process.
CSQCGETW	GET wait	Before issuing CICS wait.	Address of the ECB to be waited on.
CSQCGMGD	GET message data	After successful return from MQGET.	Up to 40 bytes of the message data.
CSQCGMGH	GET message handle	Before issuing MQGET to MQSeries for OS/390.	Object handle.
CSQCGMGI	Get message ID	After successful return from MQGET.	Message ID and correlation ID of the message.
CSQCINDL	Indoubt list	After successful return from the second INQUIRE_INDOUBT.	The indoubt units of work list.
CSQCINDO	IBM use only		
CSQCINDS	Indoubt list size	After successful return from the first INQUIRE_INDOUBT and the indoubt list is not empty.	Length of the list. Divided by 64 gives the number of indoubt units of work.

Table 22. CICS adapter trace entries (continued)

Name	Description	Trace sequence	Trace data
CSQCINQH	INQ handle	Before issuing MQINQ to MQSeries for OS/390.	Object handle.
CSQCLOSH	CLOSE handle	Before issuing MQCLOSE to MQSeries for OS/390.	Object handle.
CSQCLOST	Disposition lost	During the resynchronization process, CICS informs the adapter that it has been cold started so no disposition information regarding the unit of work being resynchronized is available.	Unit of work ID known to CICS for the unit of work being resynchronized.
CSQCNIND	Disposition not indoubt	During the resynchronization process, CICS informs the adapter that the unit of work being resynchronized should not have been indoubt (that is, perhaps it is still running).	Unit of work ID known to CICS for the unit of work being resynchronized.
CSQCNORT	Normal termination	Before issuing END_THREAD NORMAL to MQSeries for OS/390. This is due to the end of the task and therefore an implicit syncpoint commit may be performed by the application. A COMMIT request is included in the END_THREAD call in this case.	Unit of work information.
CSQCOPNH	OPEN handle	After successful return from MQOPEN.	Object handle.
CSQCOPNO	OPEN object	Before issuing MQOPEN to MQSeries for OS/390.	Object name.
CSQCPMGD	PUT message data	Before issuing MQPUT to MQSeries for OS/390.	Up to 40 bytes of the message data.
CSQCPMGH	PUT message handle	Before issuing MQPUT to MQSeries for OS/390.	Object handle.
CSQCPMGI	PUT message ID	After successful MQPUT from MQSeries for OS/390.	Message ID and Correlation ID of the message.
CSQCPREP	Syncpoint prepare	Before issuing PREPARE to MQSeries for OS/390 in the first phase of two-phase commit processing. This call can also be issued from the distributed queuing component as an API call.	Unit of work information.
CSQCP1MD	PUTONE message data	Before issuing MQPUT1 to MQSeries for OS/390.	Up to 40 bytes of data of the message.
CSQCP1MI	PUTONE message ID	After successful return from MQPUT1.	Message ID and correlation ID of the message.
CSQCP1ON	PUTONE object name	Before issuing MQPUT1 to MQSeries for OS/390.	Object name.
CSQCRBAK	Resolved backout	Before issuing RESOLVE_ROLLBACK to MQSeries for OS/390.	Unit of work information.
CSQCRGMT	Resolved commit	Before issuing RESOLVE_COMMIT to MQSeries for OS/390.	Unit of work information.
CSQCRMIR	RMI response	Before returning to the CICS RMI (resource manager interface) from a specific invocation.	Architected RMI response value. Its meaning depends of the type of the invocation. These values are documented in the <i>CICS Transaction Server for OS/390 Customization Guide</i> . To determine the type of invocation, look at previous trace entries produced by the CICS RMI component.

Debugging programs

Table 22. CICS adapter trace entries (continued)

Name	Description	Trace sequence	Trace data
CSQCRSYN	Resynchronization	Before the resynchronization process starts for the task.	Unit of work ID known to CICS for the unit of work being resynchronized.
CSQCSETH	SET handle	Before issuing MQSET to MQSeries for OS/390.	Object handle.
CSQCTASE	IBM use only		
CSQCTEST	Trace test	Used in EXEC CICS ENTER TRACE call to verify the trace number supplied by the user or the trace status of the connection.	No data.
CSQCDCFF	IBM use only		

Debugging TSO programs

The following interactive debugging tools are available for TSO programs:

- TEST tool
- VS COBOL II interactive debugging tool
- INSPECT interactive debugging tool for C and PL/I programs

Chapter 26. Building your application on OS/2 Warp

The OS/2 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 OS/2 Warp applications to run under OS/2 Warp. C, C++, and COBOL programming languages are supported. For information about preparing your C++ programs, see the *MQSeries Using C++* book.

The tasks you must perform to create an executable application using MQSeries for OS/2 Warp vary with the programming language your source code is written in. In addition to coding the MQI calls in your source code, you must add the appropriate language statements to include the MQSeries for OS/2 Warp include files for the language you are using. You should make yourself familiar with the contents of these files. See "Appendix F. MQSeries data definition files" on page 541 for a full description.

Preparing C programs

<p>For DOS and Windows 3.1 only Applications must be built using the large memory model.</p>

Work in your normal environment; MQSeries for OS/2 Warp requires nothing special.

- You need to link your programs with the appropriate libraries provided by MQSeries. Link program/exit type server for 32-bit C with library file MQM.LIB. Link program/exit type client for C with library file MQIC.LIB.

The following command gives an example of compiling the sample program amqsget0:

```
icc amqsget0.c /Gm /Gd /B "/pmttype:vio" /Fe"amqsget0.exe" mqm.lib
```

Notes:

1. If you are writing an installable service (see the *MQSeries Programmable System Management* book for further information), link to the MQMZFLIB library.
 2. If you are producing an XA switch load file for external coordination by an XA-compliant transaction manager such as IBM CICS, Transarc Encina, or Novell Tuxedo, use the MQRMIXASwitch structure and link to the MQMXA.LIB library.
 3. If you are writing a CICS exit for use with CICS for OS/2 Version 2.0.1, link to the MQMCICS.LIB library. If you are writing a CICS exit for use with CICS Transaction Server for OS/2, Version 4, link to the MQMCICS3.LIB library.
- **For DOS only:** Your application must also be linked with two of the following libraries, one for each protocol, indicating whether you do or do not require it. If you require TCP/IP you must also link to SOCKETL from the DOS TCP/IP product.

Preparing C programs

Library file	Program/exit type
MQICN.LIB	NetBIOS required
MQICDN.LIB	NetBIOS not required
MQICT.LIB	TCP/IP required
MQICDT.LIB	TCP/IP not required

- You must ensure that you have specified adequate run-time stack and heap sizes:
 - You must link a trusted application with more stack than a normal application. Therefore, a stack size of 200 KB is the recommended minimum.
 - A heap size of 8 KB is the recommended minimum.
- The DLLs must be in the library path (LIBPATH) you have specified.
- If you use lowercase characters whenever possible, you can move from MQSeries for OS/2 Warp to MQSeries on UNIX systems, where use of lowercase is necessary.

Preparing CICS and Transaction Server programs

Sample C source for a CICS MQSeries transaction is provided by AMQSCIC0.CCS. You build it using the standard CICS facilities.

For CICS for OS/2 Version 2:

1. Add the following lines to the CICSENV.COMD file:

```
UserWrk = 'c:\mqm\d11'  
UserInclude = 'c:\mqm\tools\c\include;c:\mqm\tools\c\samples'
```

If necessary replace c:\mqm with the path on which you installed the sample code.

2. Compile using the command:

```
CICS32TC AMQSCIC0.CCS LIBS(MQM)
```

This is described in the *CICS for OS/2 V2.0.1 Application Programming Guide*.

For CICS Transaction Server for OS/2, Version 4:

1. Add the following lines to the CICSENV.COMD file:

```
UserWrk = 'c:\mqm\d11'  
UserInclude = 'c:\mqm\tools\c\include;c:\mqm\tools\c\samples'
```

If necessary replace c:\mqm with the path on which you installed the sample code.

2. Compile using the command:

```
CICSCTCL AMQSCIC0.CCS LIBS(MQM)
```

This is described in the *Transaction Server for OS/2 Warp, V4 Application Programming Guide*.

You can find more information about supporting CICS transactions in the *MQSeries System Administration* book.

Preparing COBOL programs

To prepare COBOL programs on OS/2, link your programs with one of the following libraries provided by MQSeries:

Library file	Program/exit type
MQMCB16	Server for 16-bit Micro Focus COBOL
MQICCB16	Client for 16-bit Micro Focus COBOL
MQMCBB	Server for 32-bit IBM VisualAge COBOL
MQMCB32	Server for 32-bit Micro Focus COBOL
MQICCB	Client for 32-bit IBM VisualAge COBOL
MQICCB32	Client for 32-bit Micro Focus COBOL

To compile, for example, the sample program amq0put0, using IBM VisualAge COBOL:

1. Set the SYSLIB environment variable to include the path to the MQSeries VisualAge COBOL copybooks:

```
set SYSLIB=<drive>:\mqm\tools\cobol\copybook\VAcobol;%SYSLIB%
```

2. Compile and link the program:

```
cob2 amq0put0.cbl -qlib <drive>:\mqm\tools\lib\mqmcbb.lib
```

(for use on the MQSeries server)

```
cob2 amq0put0.cbl -qlib <drive>:\mqm\tools\lib\mqicbb.lib
```

(for use on the MQSeries client)

Note: Although the compiler option CALLINT(SYSTEM) must be used, this is the default for cob2.

To prepare Micro Focus COBOL programs, follow these steps:

1. Compile your applications with the LITLINK directive.
2. Specify adequate run-time stack sizes. You must link a trusted application with more stack than a normal application, so a stack size of 200 KB is the recommended minimum. To do this, use:

```
set cobs=xxxx
```

3. Link the object file to the run-time system.

Set the LIB environment variable to point to the compiler COBOL libraries.

Link the object file for use on the MQSeries server:

```
cbllink amq0put0.obj mqmcb32.lib
```

or

Link the object file for use on the MQSeries client:

```
cbllink amq0put0.obj mqiccb32.lib
```

4. Add the MQSeries copybook directory (\mqm\tools\cobol\copybook) to the cobcpy environment variable.

```
set cobcpy=c:\mqm\tools\cobol\copybook;%COBCPY%
```

Preparing COBOL programs

Preparing Transaction Server programs

To prepare CICS Transaction Server for OS/2, V4 programs using IBM VisualAge COBOL:

1. Add the following lines to the CICSENV.COMD file:

```
UserWrk='c:\mqm\d11'  
UserCobol='IBM'  
UserCobcopy='c:\mqm\tools\cobol\copybook'  
UserCobWork='c:\mq-cics\wrk'
```

Where \mq-cics\wrk is the name of a work directory for output from CICSTRAN and CICS COMP commands (see steps 2 and 3).

2. Translate your program:

```
CICSTRAN MYPROG.CPP
```

This translates your program to a .CBL program.

3. Compile your program:

```
CICS COMP MYPROG.CBL
```

4. Link your program:

```
CICSLINK MYPROG.OBJ LIBS(MQMCBB)
```

For further information about this, see the *CICS for OS/2 Customization V3.0, SC33-1581-00* and the *Transaction Server for OS/2 Warp, V4 Application Programming Guide*.

Preparing PL/I programs

Sample PL/I programs are supplied with MQSeries. PL/I include files are also provided so that the C entry points in the MQSeries libraries can be invoked directly.

To prepare a PL/I program:

1. Link your program with one of the libraries listed in "Preparing C programs" on page 323.
2. Ensure that \mqm\tools\pli\include is in your INCLUDE environment variable.
3. Compile your program:

```
pli amqpput0.pli  
ilink amqpput0.obj mqm.lib
```

Chapter 27. Building your application on SINIX or DC/OSx

This chapter describes the additional tasks, and the changes to the standard tasks, you must perform when building MQSeries for SINIX and DC/OSx applications to run under SINIX or DC/OSx. COBOL and C programming languages are supported.

In addition to coding the MQI calls in your source code, you must add the appropriate include files. You should make yourself familiar with the contents of these files. See “Appendix F. MQSeries data definition files” on page 541 for a full description.

Throughout this chapter the \ character is used to split long commands over more than one line. Do not enter this character, enter each command as a single line.

Preparing C programs

You need to link your programs with the appropriate library provided by MQSeries.

If you are *not* working in a DCE-threaded environment or using CICS, you must link to one of the following libraries:

Library file	Program/exit type
libmqm.so	server for C
libmqic.so	client for C

If you *are* working in a DCE-threaded environment or using CICS, you must link to the C library, libmqm_r.so.

Notes:

1. If you are writing an installable service (see the *MQSeries Programmable System Management* book for further information), you need to link to the libmqmzf.so library. Installable services must not use DCE.
2. If you are producing an XA switch load file for external coordination by an XA-compliant transaction manager such as IBM CICS, Transarc Encina, or Novell Tuxedo, link to the libmqmxa.so library in a non-DCE threaded environment and to the libmqmxa_r.so library in a DCE threaded environment.

C compiler flags

When you compile dynamic libraries, or shared objects, for use with MQSeries for SINIX and DC/OSx, you *must* use the cc command in the final step that creates the library or object, and not merely the ld command. This is because the cc command automatically links various initialization data that is needed for proper dynamic linking and loading.

The order of the libraries specified is important. The following is an example of how to build the sample program amqspu0 for SINIX:

```
cc -o amqspu0 -lmqm -lmqmcs -lmqzse -lnsl \
-lsocket -ldl -lproc -ltext amqspu0.c
```

Preparing C programs

For DC/OSx Version cd087, include `-liconv -lresolv` on the above command, as shown below:

```
cc -o amqspu0 -lmqm -lmqmcs -lmqmzse -lnsl \
-lsocket -ldl -liconv -lresolv -lmproc -lxt amqspu0.c
```

In the same way, for versions preceding cd087 of DC/OSx, include `-liconv`.

Note: If you are using an additional product such as ENCINA, you need to find the appropriate header files. You can do this in two ways:

1. Use the `-I` option to scan the extra include directory, for example:

```
cc -c -I/opt/encina/include amqsaex.c
```

2. Symbolically link the header files into `/usr/include`, for example:

```
ln -s /opt/encina/include/* /usr/include
```

Preparing COBOL programs

You must compile your COBOL programs using the Micro Focus Cobol compiler for SINIX with the LITLINK directive.

You must ensure that you have specified adequate run-time stack sizes; 16 KB is the recommended minimum.

You need to link your program with one of the following:

Library file	Program/exit type
libmqmcbprt.o	MQSeries COBOL run-time
libmqmcb.so	server for COBOL
libmqicb.so	client for COBOL

Export the following variables:

```
COBDIR=/usr/opt/lib/cobol export COBDIR
COBLIB=$COBDIR/coblib export COBLIB
COBCPY=/opt/mqm/inc export COBCPY
LD_LIBRARY_PATH=/opt/lib/cobol/coblib export LD_LIBRARY_PATH
```

Compiling COBOL programs

To compile a COBOL program like `amq0gbr0.cbl` on MQSeries for SINIX and DC/OSx, enter:

```
$ cob -xU -C warning=2 amq0gbr0.cbl -lmqmb -lmqm \
-lmqmcs -lmqmzse -lmproc
```

For DC/OSx Version cd087, include `-liconv -lresolv` on the above command. In the same way, for versions preceding cd087 of DC/OSx, include `-liconv`.

If you want to use the programs on a machine which only has MQSeries client connections, recompile the programs and link them with the `mqicb` library instead of the `mqmcb` library.

Note: The `mqicb` and `mqmcb` libraries *must* come before the `mqm` library on the above command line.

If you have DCE, you can link your COBOL batch programs with either DCE threaded libraries or non-DCE threaded libraries.

Notes:

1. A single program cannot contain both DCE threaded and non-DCE threaded modules.
2. Programs running under CICS must always be DCE threaded.
3. DCE threaded libraries are referred to as *re-entrant*.

If you do not choose to use DCE threaded libraries, remove `/opt/dcelocal/bin` from your `PATH` environment variable before calling the COBOL compiler.

If you do choose to use DCE threaded libraries, export the following:

```
$ export COBLIBLIST="/opt/lib/cobol/coblib/liblist_r"
```

Note: COBLIBLIST is used only in the Micro Focus Compiler for SINIX. It is the same as the Micro Focus Compiler expression LIBLIST.

Preparing CICS programs

An XA switch module is provided to enable you to link CICS with MQSeries:

Table 23. Essential Code for CICS applications (SINIX)

Description	C (source)	C (exec) - add one of the following to your XAD.Stanza
XA initialization routine	amqzscix.c	amqzsc - CICS for Siemens Nixdorf SINIX V2.2

Always link your transactions with the thread safe MQSeries library `libmqm_r.so`.

You can find more information about supporting CICS transactions in the *MQSeries for SINIX and DC/OSx System Management Guide*.

CICS on Open Systems support

MQSeries for SINIX supports CICS on Open Systems via the XA interface.

Note: MQSeries for DC/OSx does not support CICS.

In order to enable the CICS run-time COBOL library to call MQSeries on UNIX systems, you must add the MQSeries COBOL run-time library module to the run-time library using the following command:

```
cicsmkcobol libmqmcbt.o -lmqm_r
```

It is important to ensure that the COBOL run-time library and CICS are linked to the same (DCE) version of the library. All CICS on Open Systems MQSeries transactions *must* link with `libmqm_r`.

CICS sample transaction

Sample C source for a CICS MQSeries transaction is provided by `AMQSCIC0.CCS`. You build it using the standard CICS facilities. Compile it using the following commands.

Export the following environment variables:

```
export CCFLAGS="-I/opt/mqm/inc -I/opt/mqm/samp"
export USERLIB="-L/opt/mqm/lib -L/opt/cics/lib -lmqm_r -lmqmc_r"
```

Preparing CICS programs

Unset the lib path.

Then, use the command:

```
cicstcl -l C amqscic0.ccs
```

The transaction reads messages from the transmission queue SYSTEM.SAMPLE.CICS.WORKQUEUE on the default queue manager and places them onto the local queue whose name is contained in the transmission header of the message. Any failures will be sent to the queue SYSTEM.SAMPLE.CICS.DLQ. The sample MQSC script AMQSCIC0.TST may be used to create these queues and sample input queues.

Linking libraries

You need to link your programs with the appropriate library provided by MQSeries.

You must link to one or more of the following libraries:

Library file	Program/exit type
libmqm.so	Server for C
libmqmzse.so	For C
libmqic.so	Client for C
libmqmcs.so	Client for C
libmqmzf.so	Installable service
libmqmxa.a	XA interface

If you are using an additional product such as ENCINA, you need to find the run-time libraries. There are three ways (the first two are preferred, especially if the module is an exit or trigger monitor):

1. Link the libraries into /usr/lib/, for example:

```
ln -s /opt/encina/lib/*.so /usr/lib
```

Note: You need to check these symbolic links when you install a newer version of ENCINA.

2. Set LD_LIBRARY_PATH to include the ENCINA library directory (this is in the environment when you run the programs), for example:

```
LD_LIBRARY_PATH=/opt/encina/lib export LD_LIBRARY_PATH
```

3. Set LD_RUN_PATH to include /opt/encina/lib when you compile the programs.

To compile an ENCINA program on SINIX which uses the MQI:

```
LD_RUN_PATH=/opt/encina/lib export LD_RUN_PATH
cc -o amqsaex -I/opt/encina/include amqsaex.c -lmqm -lmqmc \
-lmqmcs -lmqmzse -lnsl -lsocket -lencina -ldl -lproc -ltext
```

For DC/OSx, include -liconv on the above command line.

Chapter 28. Building your application on Sun Solaris

This chapter describes the additional tasks, and the changes to the standard tasks, you must perform when building MQSeries for Sun Solaris applications to run under Sun Solaris. COBOL, C, and C++ programming languages are supported. For information about preparing your C++ programs, see the *MQSeries Using C++* book.

In addition to coding the MQI calls in your source code, you must add the appropriate include files. You should make yourself familiar with the contents of these files. See “Appendix F. MQSeries data definition files” on page 541 for a full description.

Sun Solaris applications must be built threaded, regardless of how many threads the application uses. This is because MQSeries will create background threads. Do not use nonthreadsafe functions such as:

- asctime
- ctime
- qmtime
- localtime
- rand
- srand

Use their threadsafe equivalents.

Throughout this chapter the \ character is used to split long commands over more than one line. Do not enter this character, enter each command as a single line.

Preparing C programs

Precompiled C programs are supplied in the /opt/mqm/samp/bin directory. To build a sample from source code, use a supported compiler (see “Appendix A. Language compilers and assemblers” on page 471 for more information).

To compile, for example, the sample program amqsput0:

1. export LIB=/opt/mqm/lib:\$LIB
2. Ensure the environment is set to use the correct versions of the compiler software and man pages:

```
export PATH=/opt/SUNWspro/bin:$PATH
export MANPATH=/opt/SUNWspro/man:/usr/man:$MANPATH
export LD_LIBRARY_PATH= \
/opt/SUNWspro/lib:$OPENWINHOME/lib:$LD_LIBRARY_PATH
```

3. Compile the program (the order of the libraries specified is important):

```
cc -o <amqsput0> <amqsput0>.c -mt -lmqm -lmqmcs -lmqzse \
-lsocket -lnsl -ldl
```

If you wish to compile a DCE application, use the following:

```
cc -o <amqsput0> <amqsput0>.c -mt -lmqm -lmqmcs_d -lmqzse \
-ldce -lthread -lsocket -lnsl -ldl
```

Preparing C programs

If you use the unsupported compiler `/usr/ucb/cc`, your application may compile and link successfully. However when you run it, it will fail when it attempts to connect to the queue manager.

If you want to use the programs on a machine which has only the MQSeries client for Sun Solaris installed, recompile the programs to link them with the client library instead:

```
cc -o <amqsput0> <amqsput0>.c -lmqic -lmqmcs -lsocket
```

To build an MQSeries client application that uses DCE, enter:

```
cc -o <amqsput0> <amqsput0>.c -mt -lmqic -lmqmcs_d -lmqzmse \
-ldce -lm -lpthread -lsocket -lc -lnsl -ldl
```

Linking libraries

You must link with the MQSeries libraries that are appropriate for your application type:

Program/exit type

Library files

Server for C

libmqm.so, libmqmcs.so, and libmqzmse.so

Client for C

libmqic.so, libmqmcs.so, and libmqzmse.so

Server for C with DCE

libmqm.so, libmqmcs_d.so, and libmqzmse.so

Client for C with DCE

libmqic.so, libmqmcs_d.so, and libmqzmse.so

Notes:

1. If you are writing an installable service (see the *MQSeries Programmable System Management* book for further information), link to the libmqmzf.so library.
2. If you are producing an XA switch load file for external coordination by an XA-compliant transaction manager such as IBM CICS, Transarc Encina, or Novell Tuxedo, link to the libmqmxa.a library.
3. To run the sample Encina program, link against the following libraries in addition to the libraries listed above.
 - libmqmxa.a

Also, link against libmqmcs_d.so instead of libmqmcs.so, in addition to the Encina and DCE libraries:

- libEncServer.so
- libEncina.so
- libdce.so

Preparing COBOL programs

Before preparing your COBOL programs, you should check with your system administrator that the COBOL compiler is set up to link with the correct C libraries. By default, the COBOL compiler Version 3.2 links to 3.0 SPARCompiler C libraries. For example, to update the compiler to link with SPARCompiler Version 4.0, ensure that your system administrator has completed the following:

1. Change directory to `$COBDIR/coblib`:

```
cd $COBDIR/coblib
```

Note: By default, COBDIR is `/opt/lib/cobol`.

Preparing COBOL programs

2. Make a backup copy of liblist:

```
cp liblist liblist.saved
```
3. Edit the liblist file using a standard UNIX editor like vi:

```
vi liblist
```
4. Change all references from SC3.0 to SC4.0.

The COBOL compiler is now set up for you to compile COBOL programs.

Precompiled COBOL programs are supplied in the /opt/mqm/samp/bin directory. Use the Micro Focus compiler from the directory /opt/bin to build a sample from source code.

To compile, for example, the sample program amq0put0:

1. Ensure that the environment is set:

```
export COBDIR=/opt/lib/cobol
export PATH=/opt/bin:$PATH
export LD_LIBRARY_PATH=$COBDIR/coblib:$LD_LIBRARY_PATH
```

Note: The above assumes that COBOL is installed in the default directories.

2. Define the location of the copybooks which declare the MQI structures:

```
export COBCPY="/opt/mqm/inc"
```
3. Link your program with one of the following libraries when building the application:
libmqmcb.so
Server for COBOL
libmqicb.so
Client for COBOL
4. Compile the program:

```
cob -vxP amq0put0.cb1 -lmqmc -lmqm -lmqmcs -lmqzse
```

Preparing CICS programs

XA switch modules are provided to enable you to link CICS with MQSeries:

Table 24. Essential Code for CICS applications (Sun Solaris)

Description	C (source)	C (exec) - add one of the following to your XAD.Stanza
XA initialization routine	amqzscix.c	amqzsc - TXSeries for Sun Solaris

Always link your transactions with the thread safe MQSeries library libmqm_so.

You can find more information about supporting CICS transactions in the *MQSeries System Administration* book.

CICS on Open Systems support

MQSeries on UNIX systems supports CICS on Open Systems via the XA interface.

You must ensure that CICS COBOL applications are linked to the threaded version of the library. CICS on Open Systems MQSeries transactions must link with libmqm_r, except on Sun Solaris, where you must link with lmqmcs_d.

Preparing CICS programs

Preparing CICS COBOL programs using Micro Focus COBOL

To use Micro Focus COBOL, follow these steps:

1. Add the MQSeries COBOL run-time library module to the run-time library using the following command:

```
cicsmkcobol /opt/mqm/lib/libmqmcbt.o -lmqcs_d
```

This creates the Micro Focus COBOL language method file and enables the CICS run-time COBOL library to call MQSeries on UNIX systems.

Note: `cicsmkcobol` must be run only when one of the following is installed:

- New version or release of Micro Focus COBOL
- New version or release of TXSeries for Sun Solaris
- New version or release of any supported database product (for COBOL transactions only)
- TXSeries for Sun Solaris

2. Export the following environment variables:

```
COBCPY=/opt/mqm/inc export COBCPY  
LDFLAGS="-L/usr/mqm/lib -lmqcs_d" export LDFLAGS
```

3. Translate, compile, and link the program by typing:

```
cicstcl -l COBOL -e <yourprog>.ccp
```

Preparing CICS C programs

You build CICS C programs using the standard CICS facilities:

1. Export *one* of the following environment variables:
 - `LDFLAGS = "-L/opt/mqm/lib -L/opt/cics/lib -lmqcs_d -lmqm -lmqzse -lsocket -lnsl -ldl"`
2. Translate, compile, and link the program by typing:

```
cicstcl -l C amqscic0.ccs
```

CICS C sample transaction: Sample C source for a CICS MQSeries transaction is provided by `AMQSCIC0.CCS`. The transaction reads messages from the transmission queue `SYSTEM.SAMPLE.CICS.WORKQUEUE` on the default queue manager and places them onto the local queue whose name is contained in the transmission header of the message. Any failures are sent to the queue `SYSTEM.SAMPLE.CICS.DLQ`. The sample MQSC script `AMQSCIC0.TST` may be used to create these queues and sample input queues.

Chapter 29. Building your application on Tandem NSK

The sample programs and the sample compilation and binding scripts, provided in subvolume ZMQSSMPL, illustrate the main features of the MQI in MQSeries for Tandem NonStop Kernel, and demonstrate how to compile and bind an application.

This chapter describes some minor differences between the standard Version 2 MQI interface, as documented in the *MQSeries Application Programming Reference* manual, and the MQI interface for MQSeries for Tandem NonStop Kernel.

Unit of work (transaction) management

Transaction management is performed under the control of Tandem's TM/MP product, rather than by MQSeries itself. See "Syncpoints in MQSeries for Tandem NonStop Kernel applications" on page 188 for details.

General design considerations

Please note that:

- The MQI library (bound into the application process) does not open \$RECEIVE and does not open \$TMP (TM/MP transaction pseudo-file) itself, so you may code your application to use these features.
- The MQI library uses a SERVERCLASS_SEND_() call in initial communication with the Queue Manager. While connected, it maintains two process file opens (with the LINKMON process and a Local Queue Manager Agent) and a small number of disk file opens (fewer than 10).

MQGMO_BROWSE_* with MQGMO_LOCK

As a consequence of the use of TM/MP, MQGMO_BROWSE_* with MQGMO_LOCK is not supported.

Triggered applications

Triggered MQSeries applications in the Tandem NSK environment receive user data through environment variables set up in the TAFL process that is running. This is because there is a limit to the length of the argument list that can be passed to a Tandem C process.

In order to access this information, triggered applications should contain code similar to the following (see sample amqsinqa for more details):

```
MQTMC2 *trig;                /* trigger message structure */
MQTMC2 trigdata;            /* trigger message structure */
char    *applId;
char    *envData;
char    *usrData;
char    *qmName;

/*****
/*
/* Set the program argument into the trigger message
/*
/*
*****/
trig = (MQTMC2*)argv[1];    /* -> trigger message */
```

Unir of work management

```
/* get the environment variables and load the rest of the trigger */
memcpy(&trigdata, trig, sizeof(trigdata));

memset(trigdata.ApplId, ' ', sizeof(trigdata.ApplId));
memset(trigdata.EnvData, ' ', sizeof(trigdata.EnvData));
memset(trigdata.UserData, ' ', sizeof(trigdata.UserData));
memset(trigdata.QMgrName, ' ', sizeof(trigdata.QMgrName));

if( (applId = getenv("TRIGAPPLID")) != 0)
{
    strncpy( trigdata.ApplId ,applId, strlen(applId) );
}

if ( (envData = getenv("TRIGENVDATA")) != 0)
{
    strncpy( trigdata.EnvData , envData, strlen(envData) );
}

if ( (usrData = getenv("TRIGUSERDATA")) != 0)
{
    strncpy( trigdata.UserData, usrData, strlen(usrData) );
}

if ( (qmName = getenv("TRIGQMGRNAME")) != 0)
{
    strncpy( trigdata.QMgrName, qmName, strlen(qmName) );
}

trig = &trigdata;
```

Compiling and binding applications

The MQSeries for Tandem NonStop Kernel.0.1 MQI is implemented using the Tandem wide memory model (the int datatype is 4 bytes) and the Common Run-time Environment (CRE). Applications must be compatible with this environment to work correctly. Refer to the sample build files for the correct options for each compiler to ensure compatibility.

In particular, TAL and COBOL applications must follow the rules that are required for compatibility with the CRE, documented in the Tandem manuals relating to the CRE.

Four versions of the MQI library are delivered with MQSeries for Tandem NonStop Kernel.0.1, contained in ZMQSLIB. You must ensure that you use the correct library, as follows:

mqmlibc	for C, non-native
mqmlibt	for TAL or COBOL, non-native
mqmlibnc	for native C
mqmlibnt	for native TAL or COBOL

Running applications

In order to be able to connect to a queue manager, the environment of an application program must be correctly defined:

- The PARAM MQDEFAULTPREFIX is mandatory in the environment of all applications.
- If you have chosen an alternative (nondefault) location for your MQSINI file, an application will not be able to connect to the queue manager if the PARAM MQMACHINIFILE is not set correctly.
- TAL and COBOL applications must have the PARAM SAVE-ENVIRONMENT ON defined in their environment, or they will not be able to connect to the queue manager.

An application may run as either low-pin or high-pin. MQSeries executables themselves are configured to run as high-pin.

MQSeries applications are supported in the NSK environment only. No support for OSS applications is provided.

An MQSeries application may run under PATHWAY, from TACL, or as a child process of another process. Applications can even be added to the queue manager PATHWAY configuration itself, provided they behave correctly on queue manager shutdown.

Chapter 30. Building your application on VSE/ESA

This chapter describes the additional tasks, and the changes to the standard tasks, you must perform when building MQSeries for VSE/ESA applications to run under MQSeries for VSE/ESA. C, COBOL, and PL/I programming languages are supported.

Linking library

The object decks required by MQSeries for VSE/ESA applications are held in the install sublibrary PRD2.MQSERIES (this is its default name). Reference the sublibrary in a LIBDEF statement in the compile JCL:

```
// LIBDEF SEARCH=(PRD2.MQSERIES,PRD2.SCEECICS,PRD2.SCEEBASE)
```

The MQSeries object code is autolinked into the application.

Using the batch interface

If you invoke the MQSeries API from a VSE/ESA batch application, you must link-edit a special object module, which intercepts and handles the MQSeries calls, with the usercode by specifying:

```
INCLUDE MQBITCH
```

as part of the link-edit JCL.

Preparing C programs

You must meet the requirements of the COBOL language interface when you write C programs. There are no sample programs provided but an include file, equivalent to the COBOL copybooks, is supplied. It is called CMQC.H, and it declares everything required.

Preparing COBOL programs

Sample programs and copybooks are provided in COBOL for VSE/ESA.

Preparing PL/I programs

You must meet the requirements of the COBOL language interface when you write PL/I programs. There are no sample programs provided but two include files, equivalent to the COBOL copybooks, are supplied:

CMQEPP.P

Declares the MQI calls and structures

CMQP.P

Declares the MQI constants

Preparing PL/I programs

Chapter 31. Building your application on Windows

This chapter describes the additional tasks, and the changes to the standard tasks, you must perform when building MQSeries for Windows applications to run under Windows. C and Visual Basic programming languages are supported.

The tasks you must perform to create an executable application using MQSeries for Windows depend on the language in which your source code is written. In addition to coding the MQI calls in your source code, you must add the appropriate language statements to include the MQSeries for Windows data definition files for the language you are using. You should be aware of the contents of these files. See “Appendix F. MQSeries data definition files” on page 541 for a full description.

Linking libraries

You need to link your programs with the appropriate libraries provided by MQSeries:

Library file	Program
MQM.LIB	server for 32-bit C
MQM16.LIB	server for 16-bit C

Preparing Visual Basic programs

To prepare Visual Basic programs on Windows client:

1. Create a new project.
2. Add the supplied module file, CMQB.BAS, to the project.
3. Add other supplied module files if you need them:

CMQBB.BAS	MQAI support
CMQCFB.BAS	PCF support
CMQXB	Channel exits support

Call the procedure MQ_SETDEFAULTS before making any MQI calls in the project code. This procedure sets up default structures that the MQI calls require.

Specify that you are creating an MQSeries client, before you compile or run the project, by setting the conditional compilation variable *MqType* to 2 as follows:

- In a Visual Basic version 4 project:
 1. Select the Tools menu.
 2. Select Options.
 3. Select the Advanced tab in the dialog box.
 4. In the Conditional Compilation Arguments field, enter this:
MqType=2
- In a Visual Basic version 5 project:
 1. Select the Project menu.
 2. Select *Name* Properties (where *Name* is the name of the current project).

Preparing Visual Basic programs

3. Select the Make tab in the dialog box.
4. In the Conditional Compilation Arguments field, enter this:
MqType=2

Chapter 32. Building your application on Windows NT

The Windows NT 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 Windows NT applications to run under Windows NT. ActiveX, C, C++, COBOL, PL/I, and Visual Basic programming languages are supported. For information about preparing your ActiveX programs, see the *MQSeries for Windows NT Using the Component Object Model Interface* book. For information about preparing your C++ programs, see the *MQSeries Using C++* book.

The tasks you must perform to create an executable application using MQSeries for Windows NT vary with the programming language your source code is written in. In addition to coding the MQI calls in your source code, you must add the appropriate language statements to include the MQSeries for Windows NT include files for the language you are using. You should make yourself familiar with the contents of these files. See "Appendix F. MQSeries data definition files" on page 541 for a full description.

Preparing C programs

For DOS and Windows 3.1 only

Applications must be built using the large memory model.

Work in your normal environment; MQSeries for Windows NT requires nothing special.

- You need to link your programs with the appropriate libraries provided by MQSeries:

Library file	Program/exit type
MQM.LIB	server for 32-bit C
MQIC32.LIB	client for 32-bit C

The following command gives an example of compiling the sample program `amqsget0` (using the Microsoft Visual C++[®] compiler):

```
cl amqsget0.c /link mqm.lib
```

Notes:

1. If you are writing an installable service (see the *MQSeries Programmable System Management* book for further information), you need to link to the `MQMZFLIB` library.
2. If you are producing an XA switch load file for external coordination by an XA-compliant transaction manager such as IBM TXSeries, Transarc Encina, or Novell Tuxedo, use the `MQRMIASwitch` structure and link to the `MQMXA.LIB` library.
3. If you are producing an XA switch load file using the `MQRMIASwitchDynamic` structure, link to the Encina `MQMENC.LIB` library.

Preparing C programs

- To build the Encina sample, link against the following libraries:
 - MQM.LIB
 - MQMENC.LIB

Also, link against the Encina and DCE libraries:

- libEncServer.lib
 - libEncina.lib
 - libdce.lib
- If you are writing a CICS exit, link to the MQMCICS.LIB library.
 - If an application is to make changes to environment variables, such as MQSERVER, you must link it to the same C run-time libraries as those used by MQSeries. Use the '-MD' compile switch to accomplish this.
- For DOS only:** Your application must also be linked with two of the following libraries, one for each protocol, indicating whether you do or do not require it. If you require TCP/IP you must also link to SOCKETL from the DOS TCP/IP product.

Library file	Protocol
MQICN.LIB	NetBIOS required
MQICDN.LIB	NetBIOS not required
MQICT.LIB	TCP/IP required
MQICDT.LIB	TCP/IP not required

- You must ensure that you have specified adequate run-time heap and stack sizes. A heap size of 8 KB and stack size of 16 KB are the recommended minimum size.
- The DLLs must be in the path (PATH) you have specified.
- If you use lowercase characters whenever possible, you can move from MQSeries for Windows NT to MQSeries on UNIX systems, where use of lowercase is necessary.

Preparing CICS and Transaction Server programs

Sample C source for a CICS MQSeries transaction is provided by AMQSCIC0.CCS. You build it using the standard CICS facilities:

For CICS for Windows NT V2:

- Add the following lines to the CICSENV.CMD file:

```
UserWork = 'c:\mqm\d11'  
UserIncl = 'c:\mqm\tools\c\include;c:\mqm\tools\c\samples'
```

If necessary replace c:\mqm with the path on which you installed the sample code.

- Edit the CICSCL.CMD file (found in <drive>:\CNT200\UTIL) and add the library mqm.lib to the set of libraries.
- To the LIB environment variable add:
<drive>:\MQM\TOOLS\LIB
- To the INCLUDE environment variable add:
<drive>:\MQM\TOOLS\C\INCLUDE
<drive>:\CNT200\INCLUDE
- Compile using the command:
CICSCTCL AMQSCIC0

Preparing C programs

This is described in the *CICS for Windows NT V2.0 Application Programming Guide*.

For TXSeries for Windows NT, V4:

1. Set the environment variable (enter the following on one line):

```
set CICS_IBMC_FLAGS=-IC:\Program Files\MQSeries\Tools\C\Include;  
%CICS_IBMC_FLAGS%
```
2. Set the USERLIB environment variable:

```
set USERLIB=MQM.LIB;%USERLIB%
```
3. Translate, compile, and link the sample program:

```
cicstcl -l IBMC amqscic0.ccs
```

This is described in the *Transaction Server for Windows NT Application Programming Guide (CICS) V4*.

You can find more information about supporting CICS transactions in the *MQSeries System Administration* book.

Preparing COBOL programs

To prepare COBOL programs on Windows NT, link your program to one of the following libraries provided by MQSeries:

Library file	Program/exit type
MQMCBB	server for 32-bit IBM COBOL
MQMCB32	server for 32-bit Micro Focus COBOL
MQICCB	client for 32-bit IBM COBOL
MQICCB32	client for 32-bit Micro Focus COBOL
MQMCB16	server for 16-bit Micro Focus COBOL
MQICCB16	client for 16-bit Micro Focus COBOL

When you are running a program in the MQI client environment, ensure the DOSCALLS library appears before any COBOL or MQSeries library.

Micro Focus

You must relink any existing MQSeries Micro Focus COBOL programs using either `mqmcb3.lib` or `mqiccb32.lib` rather than the `mqmcbb` and `mqiccb` libraries.

To compile, for example, the sample program `amq0put0`, using IBM VisualAge COBOL:

1. Set the SYSLIB environment variable to include the path to the MQSeries VisualAge COBOL copybooks (enter the following on one line):

```
set SYSLIB=<drive>:\Program Files\MQSeries\  
Tools\Cobol\Copybook\VAcobol;%SYSLIB%
```
2. Compile and link the program (enter the following examples on one line):

```
cob2 amq0put0.cb1 -qlib <drive>:\Program Files\MQSeries\  
Tools\Lib\mqmcbb.lib
```

(for use on the MQSeries server)

```
cob2 amq0put0.cb1 -qlib <drive>:\Program Files\MQSeries\  
Tools\Lib\mqiccb.lib
```

Preparing COBOL programs

(for use on the MQSeries client)

Note: Although the compiler option CALLINT(SYSTEM) must be used, this is the default for cob2.

To compile, for example, the sample program amq0put0, using Micro Focus COBOL:

1. Set the COBCPY environment variable to point to the MQSeries COBOL copybooks (enter the following on one line):

```
set COBCPY=<drive>:\Program Files\MQSeries\Tools\Cobol\Copybook
```
2. Compile the program to give you an object file:

```
cobol amq0put0 LITLINK
```
3. Link the object file to the run-time system.
Set the LIB environment variable to point to the compiler COBOL libraries.
Link the object file for use on the MQSeries server:

```
cbllink amq0put0.obj mqmcb32.lib
```

or

Link the object file for use on the MQSeries client:

```
cbllink amq0put0.obj mqiccb32.lib
```

Preparing CICS and Transaction Server programs

To compile and link a TXSeries for Windows NT, V4 program using IBM VisualAge COBOL:

1. Set the environment variable (enter the following on one line):

```
set CICS_IBMCOB_FLAGS=c:\Program Files\MQSeries\Tools\Cobol\Copybook\VAcobol;%CICS_IBMCOB_FLAGS%
```
2. Set the USERLIB environment variable:

```
set USERLIB=MQMCBB.LIB
```
3. Translate, compile, and link your program:

```
cicstcl -l IBMCOB myprog.ccp
```

This is described in the *Transaction Server for Windows NT, V4 Application Programming Guide*.

To compile and link a CICS for Windows NT V2 program using Micro Focus COBOL:

- Edit the CICS LINK.CMD file and add the library mqmccb.lib to the set of libraries. (This file is called by the CICSTCL.CMD utility.)
- Set the COBCPY environment variable:

```
set  
cobcpy=<drive>:\mqm\tools\cobol\copybook;<drive>:\cnt200\copybook
```
- To the LIB environment variable add:

```
<drive.>:\mqm\tools\lib  
<drive.>:\cobol32\lib
```
- Edit the CICS COMP.CMD file, change LITLINK(2) to LITLINK to enable link-time, not run-time resolution of the MQI calls.
- Compile using the command:

```
CICSTCL MQMXADC
```

Where MQMXADC.CCP (not actually provided as a sample program) is the name of the program. This creates a MQMXADC.DLL.

This is described in the *CICS for Windows NT V2.0 Application Programming Guide*.

Preparing PL/I programs

Sample PL/I programs are supplied with MQSeries. PL/I include files are also provided so that the C entry points in the MQSeries libraries can be invoked directly.

To prepare a PL/I program:

1. Link your program with one of the libraries listed in “Preparing C programs” on page 343.
2. Ensure that `\mqm\tools\pli\include` is in your INCLUDE environment variable.
3. Compile your program:

```
pli amqpput0.pli
ilink amqpput0.obj mqm.lib
```

Preparing Visual Basic programs

To prepare Visual Basic programs on Windows NT:

1. Create a new project.
2. Add the supplied module file, CMQB.BAS, to the project.
3. Add other supplied module files if you need them:

CMQBB.BAS	MQAI support
CMQCFB.BAS	PCF support
CMQXB.BAS	Channel exits support

Call the procedure MQ_SETDEFAULTS before making any MQI calls in the project code. This procedure sets up default structures that the MQI calls require.

Specify whether you are creating an MQSeries server or client, before you compile or run the project, by setting the conditional compilation variable *MqType*. Set *MqType* to 1 for a server or 2 for a client as follows:

- In a Visual Basic version 4 project:
 1. Select the Tools menu.
 2. Select Options.
 3. Select the Advanced tab in the dialog box.
 4. In the Conditional Compilation Arguments field, enter this for a server:
MqType=1

or this for a client:

```
MqType=2
```

- In a Visual Basic version 5 project:
 1. Select the Project menu.
 2. Select *Name Properties* (where *Name* is the name of the current project).
 3. Select the Make tab in the dialog box.

Preparing Visual Basic programs

4. In the Conditional Compilation Arguments field, enter this for a server:
MqType=1

or this for a client:
MqType=2

Chapter 33. Using lightweight directory access protocol services with MQSeries for Windows NT

This chapter explains what a directory service is and the part played by a directory access protocol (DAP). It also explains how MQSeries applications can use a lightweight directory access protocol (LDAP) directory using a sample program as a guide.

Note: The sample program is designed for someone who is already familiar with LDAP.

What is a directory service?

A directory is a repository of information about objects, which is organized in such a way that it is easy to find the information on a specific object. A common example is a telephone directory, where information (address and telephone number) is stored about people and companies. Another example is an address book for an e-mail system, where e-mail addresses, and optionally other information such as telephone numbers, are stored for people.

On computer systems, directories can store information about computer resources, such as printers or shared disks. For example you could use a directory to find out where the nearest color printer is located. In an MQSeries application a directory can be used to provide the association between an application service (such as accounts-receivable processing) and the queue to be used for messages requiring that service (possibly identified through the queue name and its host queue manager name).

Directories are implemented as client-server systems, where the directory server holds all the information and answers requests from clients. The clients could be user-interface programs, which provide the information directly to the user, or application programs which need to locate resources to complete their work. A Directory Service comprises the directory server, administrative programs, and the client libraries and programs which are needed to configure, update, and read the directory.

What is LDAP?

Many directory services exist, such as Novell Directory Services, DCE Cell Directory Service, Banyan StreetTalk, Windows NT Directory Services, X.500, and the address book services associated with e-mail products. X.500 was proposed as a standard for global directory services by the International Standards Organization (ISO). It requires an OSI protocol stack for its communications, and largely because of this, its use has been restricted to large organizations and academic institutions. An X.500 directory server communicates with its clients using the Directory Access Protocol (DAP).

LDAP (Lightweight Directory Access Protocol) was created as a simplified version of DAP. It is easier to implement, omits some of the lesser-used features of DAP, and runs over TCP/IP. As a result of these changes it is rapidly being adopted as the directory access protocol for most purposes, replacing the multitude of proprietary protocols previously used. LDAP clients can still access an X.500 server

What is LDAP?

through a gateway (X.500 still requires the OSI protocol stack), or increasingly X.500 implementations typically include native support for LDAP as well as DAP access.

LDAP directories can be distributed and can use replication to enable efficient access to their contents.

For a more complete description of LDAP, please see the IBM Redbook *Understanding LDAP*.

Using LDAP with MQSeries

In MQSeries configurations, the information that defines message and transmission queues is stored locally. This means that in an MQSeries network the various definitions are distributed, with no central directory of this information being available for browsing. Remote messaging between MQSeries applications is commonly achieved through the use of local definitions of remote queues. The application first issues an MQOPEN call using the name specified in the local definition of the remote queue. To put the message on the remote queue, the application then issues MQPUT, specifying the handle returned from the MQOPEN call. The remote queue definition supplies the name of the destination queue, the destination queue manager, and optionally, a transmission queue. In this technique the application has to know at run-time the name specified in the local queue definition.

A variation on the above avoids the use of local definitions of remote queues. The application can specify the full destination queue name, which includes the remote queue manager name as part of the MQOPEN. The application therefore has to know these two names at run-time. Again the local queue manager must be correctly configured with the local queue definition, and with a suitably named (or default) transmission queue and an associated channel that delivers to the target.

In the case where both the source and target queue managers are defined as being members of the same cluster, then the transmission queue and channel aspects of the above two scenarios can be ignored. If the target transmission queue is a cluster queue then a local definition of a remote queue is also not required. However, similarly to the previous cases described, the application must still know the name of the destination queue.

A directory service can be used to remove this application dependency on queue names (or the combination of queue and queue manager names). The mapping between application criteria and MQSeries object names can be held in a directory and be updated dynamically, and independently of applications. At run-time the MQSeries application wishing to send a message first queries the directory using application-based criteria, for example where: `service_name = "accounts receivable"`, retrieves the relevant MQSeries object names, and then uses these returned values in the MQOPEN call.

Another example of the use of a directory is for a company that has many small depots or offices, MQSeries clients may be used to send messages to MQSeries servers located in the larger offices. The clients need to know the name of the host machine, MQI channel, and queue name for each server they send messages to. Occasionally it may be necessary to move an MQSeries server to another machine; every client that communicates with the server would need to know about the change. An LDAP directory service could be used to store the names of the host machines (and the channel and queue names) and the client programs could

retrieve the information from the directory whenever they want to send a message to a server. In this case only the directory needs to be updated if a host name (or channel or queue name) changed.

Multiple destinations for an application message could be stored in a directory, with the one chosen being dependent on availability or load-sharing considerations.

LDAP sample program

The sample program is designed for someone who is familiar with LDAP and probably already uses it. It is intended to show how MQSeries applications can use an LDAP directory.

Building the sample program

This program has been built and tested only on Windows NT using TCP/IP. As well as the general considerations mentioned in "Preparing C programs" on page 343 the following points must be observed:

- This program is designed to run as a client program, so it should be linked with the MQIC32.LIB library.
- As well as the MQSeries header files and libraries, this program must be built using LDAP client header files and libraries. These are available from several locations, including the IBM eNetwork Web site at:

<http://www.software.ibm.com/enetwork>

For example, using the IBM eNetwork client, the program should be linked with the LIBLDAPSTATIC.LIB and LIBLBERSTATICSSL.LIB libraries.

Configuring the directory

Before the sample program can be run, an LDAP Directory Server must be configured with sample data. The file MQuser.ldif contains some sample data in LDIF (LDAP Data Interchange Format). You can edit this file to suit your needs. It contains data for a fictitious company called MQuser that has a Transport Department comprising three offices. Each of these offices has a machine that runs an MQSeries server.

As a minimum you must edit the three lines that contain the host names of the machines running the MQSeries servers - these are lines 18, 27, and 36:

```
host: LondonHost
...
host: SydneyHost
...
host: WashingtonHost
```

You must change "LondonHost", "SydneyHost", and "WashingtonHost" to the names of three of your machines which run MQSeries servers. You may also change the channel and queue names if you wish (the sample uses names of the system defaults). You may also wish to increase or decrease the number of offices in the sample data.

Configuring the IBM eNetwork LDAP server

Refer to the eNetwork LDAP Directory Administrator's Guide for information about installing the directory. In the chapter "Installing and Configuring Server",

Configuring the IBM eNetwork LDAP server

work through the sections “Installing Server” and “Basic Server Configuration”. If necessary, read through the chapter “Administrator Interface” to familiarize yourself with how the interface works.

In the chapter “Configuring - How Do I”, follow the instructions for starting up the administrator, then work through the section “Configure Database” and create a default database. Skip the section “Configure replica” and using the section “Work with Suffixes”, add a suffix “o=MQuser”.

Before adding any entries to the database, you must extend the directory schema by adding some attribute definitions and an objectclass definition. This is described in the eNetwork LDAP Directory Administrator’s Guide in the chapter “Reference Information” under the section “Directory Schema”. Two sample files are included to help you with this. The file “mq.at.conf” includes the attribute definitions which you must add to the file “/etc/slapd.at.conf”. Do this by including the sample file by editing slapd.at.conf and adding a line:

```
include <pathname>/mq.at.conf
```

Alternatively you can edit the file slapd.at.conf and add the contents of the sample file directly to it, that is, add the lines:

```
# MQ attribute definitions
attribute mqChannel          ces    mqChannel          1000    normal
attribute mqQueueManager    ces    mqQueueManager    1000    normal
attribute mqQueue           ces    mqQueue           1000    normal
attribute mqPort             cis    mqPort            64      normal
```

Similarly for the objectclass definition, you can either include the sample file by editing “etc/slapd.oc.conf” and add the line:

```
include <pathname>/mq.oc.conf
```

or you can add the contents of the sample file directly to slapd.oc.conf, that is, add the lines:

```
# MQ object classdefinition
objectclass mqApplication
    requires
        objectClass,
        cn,
        host,
        mqChannel,
        mqQueue
    allows
        mqQueueManager,
        mqPort,
        description,
        l,
        ou,
        seeAlso
```

You can now start the directory server (Administration, Server, Startup) and add the sample entries to it. To add the sample entries, go to the Administration, Add Entries page of the administrator, type in the full pathname of the sample file “MQuser.ldif” and click the Submit button.

The directory server is now running and loaded with data suitable for running the sample program.

Configuring the Netscape directory server

Using the Netscape Server Administration page, click on "Create New Netscape Directory Server". You should now be presented with a form containing configuration information. Change the Directory Suffix to "o=MQuser" and add a password for the Unrestricted User. You may also, if you wish, change any other information to suit your installation. Click on the OK button, and the directory should be created successfully. Click on "Return to Server Administration" and start the directory server. Click on the directory name to start the Directory Server Administration server for the new directory.

Before adding any entries to the database, you must extend the directory schema by adding some attribute definitions and an objectclass definition. Click on the "Schema" tab of the Directory Server page. You are now presented with a form that allows you to add new attributes. Add the following attributes (the Attribute OID can be left blank for all of them):

Attribute Name	Syntax
-----	-----
mqChannel	Case Exact String
mqQueueManager	Case Exact String
mqQueue	Case Exact String
mqPort	Integer

Add a new objectClass by clicking "Create ObjectClass" in the side panel. Enter "mqApplication" as the ObjectClass Name, select "applicationProcess" as the parent ObjectClass and leave the ObjectClass OID blank. Now add some attributes to the objectClass. Select "host", "mqChannel", and "mqQueue" as Required Attributes, and select "mqQueueManager" and "mqPort" as Allowed attributes. Press the "Create New ObjectClass" button to create the objectClass.

To add the sample data, click on the "Database Management" tab and select "Add Entries" from the side panel. You must enter the pathname of the sample data file <pathname>\MQuser.ldif, enter the password, and click on the OK button.

The sample program runs as an unauthorized user, and by default the Netscape Directory does not allow unauthorized users to search the directory. You must change this by clicking the "Access Control" tab. Enter the password for the Unrestricted User and click the OK button to load in the access control entries for the directory. These should currently be empty. Press the "New ACI" button to create a new access control entry. In the entry box which appears, click on the word "Deny" (which is underlined) and in the resultant dialog box, change it to "Allow". Add a name, for example, "MQuser-access", and click on "choose a suffix" to select "o=MQuser". Enter "o=MQuser" as the target, enter the password for the Unrestricted User, and click on the "Submit" button.

The directory server is now running and loaded with data suitable for running the sample program.

Running the sample program

You should now have an LDAP Directory Server running and populated with the sample data. The data specifies three host machines all of which should be running MQSeries servers. You should ensure that the default queue manager is running on each machine (unless you changed the sample data to specify a different queue manager).

Running the sample

You should also start the MQSeries listener program on each machine; the sample uses TCP/IP with the default MQSeries port number, so you can start the listener with the command:

```
runmq1sr -t tcp
```

To test the sample, you might also wish to run a program to read the messages arriving at each MQSeries server, for example you could use the “amqstrg” sample program:

```
amqstrg SYSTEM.DEFAULT.LOCAL.QUEUE
```

The sample program uses three environment variables, one required and two optional. The required variable is LDAP_BASEDN, which specifies the base Distinguished Name for the directory search. To work with the sample data, you should set this to “ou=Transport, o=MQuser”, for example in a Windows NT Command Window type:

```
set LDAP_BASEDN=ou=Transport, o=MQuser
```

The optional variables are LDAP_HOST and LDAP_VERSION. The LDAP_HOST variable specifies the name of the host where the LDAP server is running, it defaults to the local host if it is not specified. The LDAP_VERSION variable specifies the version of the LDAP protocol to be used, and can be either 2 or 3. Most LDAP servers now support version 3 of the protocol; they all support the older version 2. This sample works equally well with either version of the protocol, and if it is not specified it defaults to version 2.

You can now run the sample by typing the program name followed by the name of the MQSeries application you wish to send messages to, in the case of the sample data the application names are “London”, “Sydney”, and “Washington”. For example, to send messages to the London application:

```
amqsldpc London
```

If the program fails to connect to the MQSeries server, an appropriate error message will appear. If it connects successfully you can start typing messages, each line you type (terminated by <return> or <enter>) is sent as a separate message, an empty line ends the program.

Program design

The program has two distinct parts: the first part uses the environment variables and command line value to query an LDAP directory server; the second part establishes the MQSeries connection using the information returned from the directory and sends the messages.

The LDAP calls used in the first part of the program differ slightly depending on whether LDAP version 2 or 3 is being used, and they are described in detail by the documentation which comes with the LDAP client libraries. This section gives a brief description.

The first part of the program checks that it has been called correctly and reads the environment variables. It then establishes a connection with the LDAP directory server at the specified host:

```
if (ldapVersion == LDAP_VERSION3)
{
    if ((ld = ldap_init(ldapHost, LDAP_PORT)) == NULL)
        ...
}
```

```

else
{
    if ((ld = ldap_open(ldapHost, LDAP_PORT)) == NULL )
        ...
}

```

When a connection has been established, the program sets some options on the server with the “ldap_set_option” call, and then authenticates itself to the server by binding to it:

```

if (ldapVersion == LDAP_VERSION3)
{
    if (ldap_simple_bind_s(ld, bindDN, password) != LDAP_SUCCESS)
        ...
}
else
{
    if (ldap_bind_s(ld, bindDN, password, LDAP_AUTH_SIMPLE) !=
        LDAP_SUCCESS)
        ...
}

```

In the sample program “bindDN” and “password” are set to NULL, which means that the program authenticates itself as an anonymous user, that is, it does not have any special access rights and can access only information which is publicly available. In practice most organizations would restrict access to the information they store in directories so that only authorized users can access it.

The first parameter to the bind call “ld” is a handle which is used to identify this particular LDAP session throughout the rest of the program. After authenticating, the program searches the directory for entries which match the application name:

```

rc = ldap_search_s(ld,          /* LDAP Handle          */
                  baseDN,      /* base distinguished name */
                  LDAP_SCOPE_ONELEVEL, /* one-level search    */
                  filterPattern, /* filter search pattern */
                  attrs,       /* attributes required   */
                  FALSE,       /* NOT attributes only  */
                  &ldapResult); /* search result        */

```

This is a simple synchronous call to the server which returns the results directly. There are other types of search which are more appropriate for complex queries or when a large number of results is expected. The first parameter to the search is the handle “ld” which identifies the session. The second parameter is the base distinguished name, which specifies where in the directory the search is to begin, and the third parameter is the scope of the search, that is, which entries relative to the starting point are searched. These two parameters together define which entries in the directory are searched. The next parameter, “filterPattern” specifies what we are searching for. The “attrs” parameter lists the attributes which we want to get back from the object when we have found it. The next attribute says whether we want just the attributes or their values as well, setting this to FALSE means that we want the attribute values. The final parameter is used to return the result.

The result could contain many directory entries, each with the specified attributes and their values. We have to extract the values we want from the result. In this sample program we only expect one entry to be found, so we only look at the first entry in the result:

```

ldapEntry = ldap_first_entry(ld, ldapResult);

```

Program design

This call returns a handle which represents the first entry, and we set up a for loop to extract all the attributes from the entry:

```
for (attribute = ldap_first_attribute(ld, ldapEntry, &ber);
    attribute != NULL;
    attribute = ldap_next_attribute(ld, ldapEntry, ber ))
{
```

For each of these attributes, we extract the values associated with it. Again we only expect one value per attribute, so we only use the first value; we determine which attribute we have and store the value in the appropriate program variable:

```
values = ldap_get_values(ld, ldapEntry, attribute);
if (values != NULL && values[0] != NULL)
{
    if (strcmp(attribute, MQ_HOST_ATTR) == 0)
    {
        mqHost = strdup(values[0]);
        ...
    }
}
```

Finally we tidy up by freeing memory (`ldap_value_free`, `ldap_memfree`, `ldap_msgfree`) and close the session by “unbinding” from the server:

```
ldap_unbind(ld);
```

We check that we have found all the MQSeries values we need from the directory, and if so we call `sendMessages()` to connect to the MQSeries server and send the MQSeries messages.

The second part of the sample program is the `sendMessages()` routine which contains all of the MQSeries calls. This is modelled on the `amqsput0` sample program, the differences being that the parameters to the program have been extended and `MQCONN` is used instead of the `MQCONN` call.

Part 4. Sample MQSeries programs

Chapter 34. Sample programs (all platforms except OS/390)

Features demonstrated in the sample programs	360
Samples for Compaq (DIGITAL) OpenVMS and UNIX systems	360
Samples for OS/2 Warp and Windows NT	362
PL/I samples for AIX, OS/2 Warp, and Windows NT	363
Visual Basic samples for Windows NT	364
Samples for AS/400	364
Samples for Tandem NonStop Kernel	366
Samples for VSE/ESA	366
Preparing and running the sample programs	367
AS/400	367
UNIX systems	367
Digital OpenVMS	367
OS/2 and Windows NT	368
Tandem NSK	369
Building C sample programs	369
Building COBOL sample programs	370
Building TAL sample programs	371
Windows	371
Running the sample programs	371
On all platforms except AS/400	371
On AS/400	372
Length of queue name	372
Inquire, Set, and Echo examples	372
The Put sample programs	373
Running the amqspu and amqspu samples	373
Running the amqspu sample	373
Running the amq0pu sample	374
Running the AMQSPUT4 C sample	374
Running the AMQ0PUT4 COBOL sample	375
Design of the Put sample program	375
The Distribution List sample program	375
Running the Distribution List sample, amqsptl0	375
Design of the Distribution List sample	375
The Browse sample programs	376
OS/2, UNIX systems, Digital OpenVMS, and Windows NT	376
AS/400	377
Design of the Browse sample program	377
The Browser sample program	378
The Get sample programs	378
Running the amqsget and amqsgetc samples	378
Running the amqsgetw sample	379
Running the amq0get sample	379
Running the AMQSGET4 and the AMQ0GET4 samples	380
Design of the Get sample program	380
The Reference Message sample programs	380
Notes for AS/400 users	381
Running the Reference Message samples	382
Design of the Put Reference Message sample (amqsprma.c, AMQSPRM4).	386
Design of the Reference Message Exit sample (amqsxrma.c, AMQSXRM4).	386
Compiling the Reference Message Exit sample	387
Design of the Get Reference Message sample (amqsgrma.c, AMQSGRM4)	388
The Request sample programs	388
Running the amqsreq0.c, amqsreq, and amqsreqc samples	389
Running the amq0req0.cbl sample	389
Running the AMQSREQ4 sample	389
Running the AMQ0REQ4 sample	389
Running the Request sample using triggering OS/2, UNIX systems, and Windows NT	390
AS/400	392
Design of the Request sample program	393
The Inquire sample programs	394
Design of the Inquire sample program	395
The Set sample programs	395
Design of the Set sample program	396
The Echo sample programs	397
Design of the Echo sample programs	397
The Data-Conversion sample program	398
Design of the data-conversion sample	398
The Triggering sample programs	398
Running the amqstrg0.c, amqstrg, and amqstrgc samples	398
Running the AMQSTRG4 sample	398
Design of the triggering sample	399
Running the AMQSERV4 sample	399
Design of the trigger server	400
Ending the triggering sample programs on AS/400	400
Running the samples using remote queues	400
Database coordination samples	400
Creating the databases and tables	402
Precompiling, compiling, and linking the samples	403
Precompiling in C	403
Precompiling in COBOL	403
Compiling and linking	404
Running the samples	405
C samples	406
COBOL samples	406
The CICS transaction sample	407
TUXEDO samples	407
Building the server environment	407
To build the server environment for MQSeries for AIX:	407
To build the server environment for MQSeries for AT&T GIS UNIX and MQSeries for Sun Solaris:	408
To build the server environment for MQSeries for HP-UX:	409
To build the server environment for MQSeries for SINIX and DC/OSx	410

To build the server environment for MQSeries for Windows NT:	412
To build the server environment for MQSeries for Compaq Tru64 UNIX	415
Server sample program for TUXEDO	416
Put sample program for TUXEDO	417
Get sample for TUXEDO	417
Encina sample program	418
Building the AMQSXAE0.C sample	418
Compiling and linking on Windows NT	418
Compiling and linking on Sun Solaris	419
Dead-letter queue handler sample	419
The Connect sample program	419
Running the amqscnxc sample	419

Chapter 35. Sample programs for MQSeries for OS/390 421

Features demonstrated in the sample applications	421
Put samples	421
Get samples	422
Browse sample	422
Print Message sample	422
Queue Attributes sample	423
Mail Manager sample	423
Credit Check sample	423
The Message Handler sample	424
Distributed queuing exit samples	425
Data-conversion exit samples	425
Preparing and running sample applications for the batch environment	425
Names of the sample batch applications	426
Preparing sample applications for the TSO environment.	427
Names of the sample TSO applications	427
Preparing the sample applications for the CICS environment.	428
QLOP abend	429
Names of the sample CICS applications	429
Preparing the sample application for the IMS environment.	431
Names of the sample IMS application	432
The Put samples	432
Design of the Put sample	432
The Put samples for the batch environment	433
Usage notes	433
The Put samples for the CICS environment	434
Usage notes	434
The Get samples	435
Design of the Get sample	435
The Get samples for the batch environment	435
Usage notes	436
The Get samples for the CICS environment	436
Usage notes	437
The Browse sample	437
Design of the Browse sample	438
Language-dependent design considerations	439
The Print Message sample	439
Design of the sample	441
The Queue Attributes sample	443
Design of the sample	443
The Mail Manager sample	444

Preparing the sample	444
Preparing the sample for the TSO environment.	445
Running the sample	445
Design of the sample	447
Menu program	448
Get-mail and display-message programs	449
Send-mail program	450
Nickname program	451
The Credit Check sample	451
Preparing and running the Credit Check sample	452
Entering information in the inquiry panels	453
Design of the sample	453
User-interface program (CSQ4CVB1)	455
Credit application manager (CSQ4CVB2)	455
Checking-account program (CSQ4CVB3)	458
Distribution program (CSQ4CVB4)	458
Agency-query program (CSQ4CVB5/CSQ4CCB5)	459
Design considerations	460
Separate inquiry and reply queues in the CAM	460
How the sample handles errors	460
How the sample handles unexpected messages	460
How the sample uses syncpoints	461
How the sample uses message context information	461
Use of message and correlation identifiers in the CAM	462
The Credit Check sample with multiple queue managers.	462
The IMS extension to the Credit Check sample	462
Design of the IMS checking-account program (CSQ4ICB3)	463
The Message Handler sample	464
Preparing and running the sample	464
Using the sample	464
Design of the sample	466
Object validation program	467
Message list program.	467
Message content program	467

Chapter 34. Sample programs (all platforms except OS/390)

This chapter describes the sample programs delivered with MQSeries, written in C, COBOL, PL/I, and TAL. 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.

DCE sample exit: For information on compiling and linking the DCE sample exit (amqsdsc0.c) see the *MQSeries Intercommunication* book.

C++ sample programs: See the *MQSeries Using C++* book for a description of the sample programs available in C++.

RPG sample programs: See the *MQSeries for AS/400 Application Programming Reference (ILE RPG)* manual for a description of the sample programs available in RPG.

The names of the samples start with the prefix amq, except for TAL programs that start zmq. The fourth character indicates the programming language, and the compiler where necessary.

s	C language
0	COBOL language on both IBM and Micro Focus compilers
i	COBOL language on IBM compilers only
m	COBOL language on Micro Focus compilers only
v	COBOL language on DEC COBOL V2.3 and subsequent releases
p	PL/I language

This chapter introduces the sample programs, under these headings:

- “Features demonstrated in the sample programs” on page 360
- “Preparing and running the sample programs” on page 367
- “The Put sample programs” on page 373
- “The Distribution List sample program” on page 375
- “The Browse sample programs” on page 376
- “The Browser sample program” on page 378
- “The Get sample programs” on page 378
- “The Reference Message sample programs” on page 380
- “The Request sample programs” on page 388
- “The Inquire sample programs” on page 394
- “The Set sample programs” on page 395
- “The Echo sample programs” on page 397
- “The Data-Conversion sample program” on page 398
- “The Triggering sample programs” on page 398
- “Running the samples using remote queues” on page 400

MQSeries sample programs

- “Database coordination samples” on page 400
- “The CICS transaction sample” on page 407
- “TUXEDO samples” on page 407
- “Encina sample program” on page 418
- “Dead-letter queue handler sample” on page 419
- “The Connect sample program” on page 419

Features demonstrated in the sample programs

The following tables show the techniques demonstrated by the MQSeries sample programs on all systems except OS/390 (see “Chapter 35. Sample programs for MQSeries for OS/390” on page 421). All the samples open and close queues using the MQOPEN and MQCLOSE calls, so these techniques are not listed separately in the tables. See the heading that includes the platform you are interested in:

“Samples for Compaq (DIGITAL) OpenVMS and UNIX systems”

“Samples for OS/2 Warp and Windows NT” on page 362

“PL/I samples for AIX, OS/2 Warp, and Windows NT” on page 363

“Visual Basic samples for Windows NT” on page 364

“Samples for AS/400” on page 364

“Samples for Tandem NonStop Kernel” on page 366

“Samples for VSE/ESA” on page 366

Samples for Compaq (DIGITAL) OpenVMS and UNIX systems

Table 25 shows the techniques demonstrated by the sample programs for MQSeries on UNIX systems and MQSeries for Digital OpenVMS.

Table 25. MQSeries on UNIX and Digital OpenVMS sample programs demonstrating use of the MQI

Technique	C (source) (1)	COBOL (source) (2)	C (executable)	Client (3) (executable)
Putting messages using the MQPUT call	amqsput0 amqsputw (4)	amq0put0	amqsput	amqsputc amqsputw
Putting a single message using the MQPUT1 call	amqsinqa amqsecha	amqminqx amqmechx amqiinqx amqiechx amqvinqx amqviechx	amqsinq amqsech	no sample
Putting messages to a distribution list (5)	amqsptl0	no sample	amqsptl	amqsptlc
Replying to a request message	amqsinq0 amqsinqa	amqminqx amqiinqx amqvinqx	amqsinq	no sample
Getting messages (no wait)	amqsgbr0	amq0gbr0	amqsgbr	no sample
Getting messages (wait with a time limit)	amqsget0 amqsgetw (4)	amq0get0	amqsget	amqsgetc amqsgetw
Getting messages (unlimited wait)	amqstrg0	no sample	amqstrg	amqstrgc
Getting messages (with data conversion)	amqsecha	no sample	amqsech	no sample
Putting reference messages to a queue (5)	amqsprma	no sample	amqsprm	amqsprmc
Getting reference messages from a queue (5)	amqsgrma	no sample	amqsgrm	amqsgrmc
Reference message channel exit (5)	amqsqrma amqsxrma	no sample	amqsxrm	no sample
Browsing first 20 characters of a message	amqsgbr0	amq0gbr0	amqsgbr	no sample
Browsing complete messages	amqsbcg0	no sample	amqsbcg	no sample

Features demonstrated

Table 25. MQSeries on UNIX and Digital OpenVMS sample programs demonstrating use of the MQI (continued)

Technique	C (source) (1)	COBOL (source) (2)	C (executable)	Client (3) (executable)
Using a shared input queue (4)	amqsinq0	amqminqx amqiinqx amqvinqx	amqsinq	no sample
Using an exclusive input queue	amqstrg0	amq0req0	amqstrg	amqstrgc
Using the MQINQ call	amqsinqa	amqminqx amqiinqx amqvinqx	amqsinq	no sample
Using the MQSET call	amqsseta	amqmsetx amqisetx amqvsetx	amqsset	no sample
Using a reply-to queue	amqsreq0	amq0req0	amqsreq	no sample
Requesting message exceptions	amqsreq0	amq0req0	amqsreq	no sample
Accepting a truncated message	amqsgbr0	amq0gbr0	amqsgbr	no sample
Using a resolved queue name	amqsgbr0	amq0gbr0	amqsgbr	no sample
Triggering a process	amqstrg0	no sample	amqstrg	amqstrgc
Using data conversion	(6)	no sample	no sample	no sample
MQSeries (coordinating XA-compliant database managers) accessing a single database using SQL	amqsxas0.sqc	amq0xas0.sqb	no sample	no sample
MQSeries (coordinating XA-compliant database managers) accessing two databases using SQL	amqsxag0.c amqsxab0.sqc amqsxaf0.sqc	amq0xag0.cbl amq0xab0.sqb amq0xaf0.sqb	no sample	no sample
CICS transaction (7)	amqscic0.ccs	no sample	amqscic0	no sample
Encina transaction (5)	amqsxae0	no sample	amqsxae0	no sample
TUXEDO transaction to put messages (8)	amqstxpx	no sample	no sample	no sample
TUXEDO transaction to get messages (8)	amqstxgx	no sample	no sample	no sample
Server for TUXEDO (8)	amqstxsx	no sample	no sample	no sample
Dead-letter queue handler	(9)	no sample	amqsdlq	no sample
From an MQI client, putting a message (4)	amqsputw	no sample	no sample	amqsputc amqsputw
From an MQI client, getting a message (4)	amqsgetw	no sample	no sample	amqsgetc amqsgetw
Connecting to the queue manager using MQCONN	amqscnxc	no sample	no sample	amqscnxc

Features demonstrated

Table 25. MQSeries on UNIX and Digital OpenVMS sample programs demonstrating use of the MQI (continued)

Technique	C (source) (1)	COBOL (source) (2)	C (executable)	Client (3) (executable)
Notes:				
1. The executable version of the MQSeries client samples share the same source as the samples that run in a server environment.				
2. COBOL is not supported by MQSeries for AT&T GIS UNIX. Compile programs beginning 'amqm' with the Micro Focus COBOL compiler, beginning 'amqi' with the IBM COBOL compiler, and beginning 'amq0' with either.				
3. The executable versions of the MQSeries client samples are not available on MQSeries for HP-UX or MQSeries for Digital OpenVMS.				
4. Not available on MQSeries for Compaq Tru64 UNIX.				
5. Supported on MQSeries for AIX, HP-UX, and Sun Solaris only.				
6. On MQSeries for AIX, HP-UX, Compaq Tru64 UNIX, and Sun Solaris this program is called amqsvfc0.c. On MQSeries for AT&T GIS UNIX, Compaq (DIGITAL) OpenVMS, and SINIX and DC/OSx this program is called amqsvfcx.c.				
7. CICS is supported by MQSeries for AIX and MQSeries for HP-UX only.				
8. TUXEDO is not supported by MQSeries for AS/400, Compaq (DIGITAL) OpenVMS, and Windows.				
9. The source for the dead-letter queue handler is made up of several files and provided in a separate directory.				

Samples for OS/2 Warp and Windows NT

Table 26 shows the techniques demonstrated by the sample programs for MQSeries for OS/2 Warp and Windows NT.

Table 26. MQSeries for OS/2 Warp and Windows NT sample programs demonstrating use of the MQI

Technique	C (source)	COBOL (source)	C (executable)	Client (executable)
Putting messages using the MQPUT call	amqsput0	amq0put0	amqsput	amqsputc amqsputw
Putting a single message using the MQPUT1 call	amqsinqa amqsecha	amqminq2 amqmech2 amqiinq2 amqiech2	amqsinq amqsech	amqsinqc amqsechc
Putting messages to a distribution list	amqsptl0	no sample	amqsptl	amqsptlc
Replying to a request message	amqsinq0	amqminq2 amqiinq2	amqsinq	amqsinqc
Getting messages (no wait)	amqsgbr0	amq0gbr0	amqsgbr	amqsgbrc
Getting messages (wait with a time limit)	amqsget0 amqsgetw	amq0get0	amqsget	amqsgetc amqsgetw
Getting messages (unlimited wait)	amqstrg0	no sample	amqstrg	amqstrgc
Getting messages (with data conversion)	amqsecha	no sample	amqsech	amqsechc
Putting reference messages to a queue	amqsprma	no sample	amqsprm	amqsprmc
Getting reference messages from a queue	amqsgrma	no sample	amqsgrm	amqsgrmc
Reference message channel exit	amqsqrma amqsxrma	no sample	amqsxrm	no sample
Browsing first 20 characters of a message	amqsgbr0	amq0gbr0	amqsgbr	amqsgbrc
Browsing complete messages	amqsbcg0	no sample	amqsbcg	amqsbcgc
Using a shared input queue	amqsinq0	amqminq2 amqiinq2	amqsinq	amqsinqc

Table 26. MQSeries for OS/2 Warp and Windows NT sample programs demonstrating use of the MQI (continued)

Technique	C (source)	COBOL (source)	C (executable)	Client (executable)
Using an exclusive input queue	amqstrg0	amq0req0	amqstrg	amqstrgc
Using the MQINQ call	amqsinqa	amqminq2 amqiinq2	amqsinq	amqsinqc
Using the MQSET call	amqsseta	amqmset2 amqiset2	amqsset	amqssetc
Using a reply-to queue	amqsreq0	amq0req0	amqsreq	amqsreqc
Requesting message exceptions	amqsreq0	amq0req0	amqsreq	amqsreqc
Accepting a truncated message	amqsgbr0	amq0gbr0	amqsgbr	amqsgbrc
Using a resolved queue name	amqsgbr0	amq0gbr0	amqsgbr	amqsgbrc
Triggering a process	amqstrg0	no sample	amqstrg	amqstrgc
Using data conversion	amqsvfc0	no sample	no sample	no sample
CICS transaction	amqscic0.ccs	no sample	amqscic0 (1)	no sample
TUXEDO transaction to put messages (Windows NT only)	amqstxpx	no sample	no sample	no sample
TUXEDO transaction to get messages (Windows NT only)	amqstxgx	no sample	no sample	no sample
Server for TUXEDO (Windows NT only)	amqstxsx	no sample	no sample	no sample
Encina transaction	amqsxae0	no sample	amqsxae0	no sample
Dead-letter queue handler	(2)	no sample	amqsdldq	no sample
From an MQSeries client, putting a message	amqsputw	no sample	no sample	amqsputc amqsputw
From an MQSeries client, getting a message	amqsgetw	no sample	no sample	amqsgetc amqsgetw
Connecting to the queue manager using MQCONN	amqscnxc	no sample	no sample	amqscnxc
Notes:				
1. The executable version on OS/2 is for CICS Transaction Server for OS/2, Version 4; the version on Windows NT is for TXSeries for Windows NT, Version 4.				
2. The source for the dead-letter queue handler is made up of several files and provided in a separate directory.				

The following list shows the techniques demonstrated by the MQSeries for Windows sample programs:

Sample program	Technique
AMQSPUTW	Putting a message on a specified queue
AMQSGETW	Getting a message from a specified queue
AMQSBGCW	Browsing a message and its header

For further information about these sample programs, see the following:

- *MQSeries for Windows V2.0 User's Guide.*
- *MQSeries for Windows V2.1 User's Guide.*

PL/I samples for AIX, OS/2 Warp, and Windows NT

Table 27 on page 364 shows the techniques demonstrated by the MQSeries for AIX, MQSeries for OS/2 Warp, and MQSeries for Windows NT sample programs.

Features demonstrated

Table 27. MQSeries for AIX, OS/2 Warp, and Windows NT sample programs demonstrating use of the MQI

Technique	PL/I (source)	PL/I (executable)
Putting messages using the MQPUT call	amqpput0	no sample
Getting messages (wait with a time limit)	amqpget0	no sample

Visual Basic samples for Windows NT

Table 28 shows the techniques demonstrated by the MQSeries for Windows NT sample programs.

A project may contain several files. When you open a project within Visual Basic the other files will be loaded automatically. No executable programs are provided.

All the sample projects, except mqtrivc.vbp, are set up to work with the MQSeries server. To find out how to change the sample projects to work with the MQSeries clients see "Preparing Visual Basic programs" on page 347.

Table 28. MQSeries for Windows NT sample programs demonstrating use of the MQI

Technique	Project file name
Putting messages using the MQPUT call	amqsputb.vbp
Getting messages using the MQGET call	amqsgetb.vbp
Browsing a queue using the MQGET call	amqsbcgb.vbp
Simple MQGET and MQPUT sample (client)	mqtrivc.vbp
Simple MQGET and MQPUT sample (server)	mqtrivs.vbp
Putting and getting strings and user-defined structures using MQPUT and MQGET	strings.vbp
Using PCF structures to start and stop a channel	pcfsamp.vbp
Creating a queue using the MQAI	amqsaiqc.vbp
Listing a queue manager's queues using the MQAI	amqsailq.vbp
Monitoring events using the MQAI	amqsaiem.vbp

Samples for AS/400

Table 29 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.

Table 29. MQSeries for AS/400 sample programs demonstrating use of the MQI

Technique	C (source) (1)	COBOL (source) (2)	RPG (source) (3)
Putting messages using the MQPUT call	AMQSPUT0	AMQ0PUT4	AMQnPUT4
Putting messages from a data file using the MQPUT call	AMQSPUT4	no sample	no sample
Putting a single message using the MQPUT1 call	AMQSINQ4, AMQSECH4	AMQ0INQ4, AMQ0ECH4	AMQnINQ4, AMQnECH4
Putting messages to a distribution list	AMQSPTL4	no sample	no sample
Replying to a request message	AMQSINQ4	AMQ0INQ4	AMQnINQ4
Getting messages (no wait)	AMQSGBR4	AMQ0GBR4	AMQnGBR4
Getting messages (wait with a time limit)	AMQSGET4	AMQ0GET4	AMQnGET4

Table 29. MQSeries for AS/400 sample programs demonstrating use of the MQI (continued)

Technique	C (source) (1)	COBOL (source) (2)	RPG (source) (3)
Getting messages (unlimited wait)	AMQSTRG4	no sample	AMQ3TRG4
Getting messages (with data conversion)	AMQSECH4	AMQ0ECH4	AMQnECH4
Putting reference messages to a queue	AMQSPRM4	no sample	no sample
Getting reference messages from a queue	AMQSGRM4	no sample	no sample
Reference message channel exit	AMQSORM4, AMQSXRM4	no sample	no sample
Message exit	AMQSCMX4	no sample	no sample
Browsing first 49 characters of a message	AMQSGBR4	AMQ0GBR4	AMQnGBR4
Browsing complete messages	AMQSBCG4	no sample	no sample
Using a shared input queue	AMQSINQ4	AMQ0INQ4	AMQnINQ4
Using an exclusive input queue	AMQSREQ4	AMQ0REQ4	AMQnREQ4
Using the MQINQ call	AMQSINQ4	AMQ0INQ4	AMQnINQ4
Using the MQSET call	AMQSSET4	AMQ0SET4	AMQnSET4
Using a reply-to queue	AMQSREQ4	AMQ0REQ4	AMQnREQ4
Requesting message exceptions	AMQSREQ4	AMQ0REQ4	AMQnREQ4
Accepting a truncated message	AMQSGBR4	AMQ0GBR4	AMQnGBR4
Using a resolved queue name	AMQSGBR4	AMQ0GBR4	AMQnGBR4
Triggering a process	AMQSTRG4	no sample	AMQ3TRG4
Trigger server	AMQSERV4	no sample	AMQ3SRV4
Using a trigger server (including CICS transactions)	AMQSERV4	no sample	AMQ3SRV4
Using data conversion	AMQSVFC4	no sample	no sample

Notes:

1. Source for the C samples is in the file QMQMSAMP/QCSRC. Include files exist as members in the file QMQM/H.
2. Source for the COBOL samples are in the files QMQMSAMP/QLBLSRC for the OPM compiler, and QMQMSAMP/QCBLLESRC for the ILE compiler. The members are named AMQ0xxx4, where xxx indicates the sample function.
3. There are three sets of RPG sample programs:
 - a. OPM RPG programs.
The source is in QMQMSAMP/QRPGSRC. Members are named AMQ1xxx4, where xxx indicates the sample function. Copy members exist in QMQM/QRPGSRC.
 - b. ILE RPG programs using the MQI through a call to QMQM.
The source is in QMQMSAMP/QRPGLESRC. Members are named AMQ2xxx4, where xxx indicates the sample function. Copy members exist in QMQM/QRPGLESRC. Each member name is suffixed with "R".
 - c. ILE RPG programs using prototyped calls to the MQI.
The source is in QMQMSAMP/QRPGLESRC. Members are named AMQ3xxx4, where xxx indicates the sample function. Copy members exist in QMQM/QRPGLESRC. Each member name is suffixed with "G".

In addition to these, the MQSeries for AS/400 sample option includes a sample data file, which can be used as input to the sample programs, AMQSDATA and sample CL programs that demonstrate administration tasks. The CL samples are described in the *MQSeries for AS/400 System Administration* book. You could use the sample CL program amqsamp4 to create queues to use with the sample programs described in this chapter.

Features demonstrated

Samples for Tandem NonStop Kernel

The following C and COBOL sample programs are supplied with MQSeries for Tandem NonStop Kernel:

Table 30. MQSeries for Tandem NonStop Kernel C and COBOL sample programs demonstrating use of the MQI

Description	C (source)	C (executable)	COBOL85 (source)	COBOL85 (executable)
Putting messages using the MQPUT call	amqsput0	amqsput	amq0put0	amq0put
Putting a single message using the MQPUT1 call	amqsinqa	amqsinq	No sample	No sample
Getting messages (no wait)	amqsgbr0	amqsgbr	amq0gbr0	amq0gbr
Getting messages (wait with a time limit)	amqsget0	amqsget	amq0get0	amq0get
Getting messages (unlimited wait)	amqstrg0	amqstrg	No sample	No sample
Getting messages (with data conversion)	amqsecha	amqsech	amq0ech0	amq0ech
Browsing complete messages	amqsbcg0	amqsbcg	No sample	No sample
Use a shared input queue	No sample	No sample	amq0inq0	amq0inq
Using the MQSET call	amqsseta	amqsset	amq0set0	amq0set
Using a reply-to queue	amqsreq0	amqsreq	amq0req0	amq0req
Using data conversion	amqsvfcn	No sample	No sample	No sample
Sample skeleton for channel exit	amqsvchn	No sample	No sample	No sample

The following TAL sample programs are supplied with MQSeries for Tandem NonStop Kernel:

Table 31. MQSeries for Tandem NonStop Kernel TAL sample programs demonstrating use of the MQI

Description	TAL (source)	TAL (executable)
Read <i>n</i> messages from a queue	zmqreadt	zmqread
Write <i>n</i> messages of <i>n</i> length to a queue	zmqwritt	zmqwrit

Samples for VSE/ESA

Table 32 shows the techniques demonstrated by the MQSeries for VSE/ESA COBOL sample programs.

Table 32. MQSeries for VSE/ESA COBOL sample programs demonstrating use of the MQI

Description	COBOL (source)	COBOL (executable)
Transaction that demonstrates MQI calls (1)	TTPTST2.Z	TTPTST2
Test facility that starts the sample transaction TTPTST2 (2)	TTPTST3.Z	TTPTST3
Triggered test program that echoes a message from a queue to a reply-to queue	MQPECHO.Z	MQPECHO

Table 32. MQSeries for VSE/ESA COBOL sample programs demonstrating use of the MQI (continued)

Description	COBOL (source)	COBOL (executable)
Notes: 1. Demonstrates MQGET, MQINQ, MQPUT, MQPUT1, both MQPUT and MQGET, MQGET and delete, MQPUT and reply. 2. Each TTPTST2 that is started is a task.		

Preparing and running the sample programs

The following sections help you find the samples that you need to run on the different platforms.

AS/400

The source for MQSeries for AS/400 sample programs are provided in library QMQMSAMP as members of QCSRC, QCLSRC, QLBSRC, QCBLESRC, QRPGSRC, and QRPGLESRC. To run the samples use either the C executable versions, supplied in the library QMQM, or compile them as you would any other MQSeries application. For more information see “Running the sample programs” on page 371.

UNIX systems

Table 33. Where to find the samples for MQSeries on UNIX systems

Content	Directory
source files	/mqmtop/samp
C source file for Windows 3.1 sample	/mqmtop/win_client/samp
dead-letter queue handler source files	/mqmtop/samp/dlq
executable files	/mqmtop/samp/bin
Other MQSeries client executable files	/mqmtop/dos_client/samp/bin /mqmtop/os2_client/samp/bin /mqmtop/win_client/samp/bin
Note: For MQSeries for AIX mqmtop is usr/mqm, for MQSeries for other UNIX systems mqmtop is opt/mqm.	

The MQSeries on UNIX systems sample files will be in the directories listed in Table 33 if the defaults were used at installation time. To run the samples, either use the executable versions supplied or compile the source versions as you would any other applications, using an ANSI compiler. For information on how to do this, see “Running the sample programs” on page 371.

Digital OpenVMS

Table 34. Where to find the samples for MQSeries for Compaq (DIGITAL) OpenVMS

Content	Directory
source files	MQS_EXAMPLES
C source file for Windows 3.1 sample	[.WIN_CLIENT.SAMP] under MQS_EXAMPLES
dead-letter queue handler source files	[.DLQ] under MQS_EXAMPLES

Preparing and running samples

Table 34. Where to find the samples for MQSeries for Compaq (DIGITAL) OpenVMS (continued)

Content	Directory
executable files	[.BIN] under MQS_EXAMPLES
Other MQSeries client executable files	[.DOS_CLIENT.SAMP.BIN] under MQS_EXAMPLES [OS2_CLIENT.SAMP.BIN] under MQS_EXAMPLES [WIN_CLIENT.SAMP.BIN] under MQS_EXAMPLES

The MQSeries for Compaq (DIGITAL) OpenVMS sample files are in the directories listed in Table 34 on page 367 if the defaults were used at installation time. To run the samples, either use the executable versions supplied or compile the source versions as you would any other applications, using an ANSI compiler. For information on how to do this, see “Running the sample programs” on page 371.

OS/2 and Windows NT

Table 35. Where to find the samples for MQSeries for OS/2 Warp and MQSeries for Windows NT

Content	Directory
C source code	<drive:directory>\MQM\TOOLS\C\SAMPLES <drive:directory>\Program Files\MQSeries\Tools\C\Samples
Source code for dead-letter handler sample	<drive:directory>\MQM\TOOLS\C\SAMPLES\DLQ <drive:directory>\Program Files\MQSeries\Tools\C\Samples\DLQ
C source code for Windows 3.1 sample	<drive:directory>\MQM\WIN
COBOL source code	<drive:directory>\MQM\TOOLS\COBOL\SAMPLES <drive:directory>\Program Files\MQSeries\Tools\Cobol\Samples
C executable files	<drive:directory>\MQM\TOOLS\C\SAMPLES\BIN <drive:directory>\Program Files\MQSeries\Tools\C\Samples\Bin
Other MQSeries client executable files	<drive:directory>\MQM\DOS <drive:directory>\MQM\AIX <drive:directory>\MQM\WIN
Sample MQSC files	<drive:directory>\MQM\TOOLS\MQSC\SAMPLES <drive:directory>\Program Files\MQSeries\Tools\MQSC\Samples
PL/I source code	<drive:directory>\MQM\TOOLS\PLI\SAMPLES <drive:directory>\Program Files\MQSeries\Tools\PLI\Samples
Visual Basic Version 4 source code	<drive:directory>\Program Files\MQSeries\Tools\Samples\VB\Sampvb4
Visual Basic Version 5 source code	<drive:directory>\Program Files\MQSeries\Tools\Samples\VB\Sampvb5
Note: The Visual Basic samples are not available for OS/2.	

Note: MQSeries for Windows NT samples are in the directories that begin <drive:directory>\Program Files.

The MQSeries for OS/2 Warp and MQSeries for Windows NT sample files will be in the directories listed in Table 35 on page 368 if the defaults were used at installation time, the <drive:directory> will default to <c:>. To run the samples, either use the executable versions supplied or compile the source versions as you would any other MQSeries for OS/2 Warp or MQSeries for Windows NT applications. For information on how to do this, see "Running the sample programs" on page 371.

Tandem NSK

See the relevant following section for your programming language.

Building C sample programs

The subvolume ZMQSSMPL contains the following TACL macro files to be used for building non-native sample C applications:

CSAMP

Usage: CSAMP *source-code-file-name*

This is a basic macro for compiling a C source file using the include files contained in subvolume ZMQSLIB. For example, to compile the sample AMQSBCG0, use CSAMP AMQSBCG0. If the compilation is successful, the macro produces an object file with the last character of the file name replaced by the letter O; for example, AMQSBCG0.

BSAMP

Usage: BSAMP *exe-file-name*

This is a basic macro used to bind an object file with the user libraries in ZMQSLIB. For example, to bind the compiled sample AMQSBCG0, use BSAMP AMQSBCG. The macro produces an executable file called *exe-file-nameE*; for example, AMQSBCGE.

COMPALL

Usage: COMPALL

This TACL macro compiles each of the sample source code files using the CSAMP macro.

BINDALL

Usage: BINDALL

This TACL macro binds each of the sample object files into executables using the BSAMP macro.

BUILDC

Usage: BUILDC

This TACL macro compiles and binds all of the C sample files using the macros COMPALL and BINDALL.

For a native install, the following TACL macro files are to be used for building sample MQI applications:

NMCCALL

Usage: NMCCALL

Macro to compile all samples native using NMCSAMP.

Preparing and running samples

NMCPSSL

Usage: NMCPSSL *source-code-file-name*

Macro to compile user code for inclusion in the MQSeries PSRL.

NMCSAMP

Usage: NMCSAMP *source-code-file-name*

This is a basic macro for compiling a C source file using the include files contained in subvolume ZMQSLIB. For example, to compile the sample AMQSBCG0, use NMCSAMP AMQSBCG0. If the compilation is successful, the macro produces an object file with the last character of the file name replaced by the letter O; for example, AMQSBCGO.

NMLDSAMP

Usage: NMLDSAMP *exe-file-name*

This basic macro links an object file with the static MQI library in ZMQSLIB.

NMLDPSRL

Usage: NMLDPSRL *exe-file-name*

This basic macro links an object file with the MQSeries private SRL in ZMQSLIB

NMLDUSRL

Usage: NMLDUSRL *object-input-file*, where *object-input-file* is a file containing a list of objects to be linked.

This is a basic macro for linking user code into a relinkable library.

Note: Non-native applications can connect to native queue managers, and native applications can connect to non-native queue managers. All combinations of native and non-native operation are valid and supported.

Building COBOL sample programs

The subvolume ZMQSSMPL contains the following files to be used for building sample COBOL applications.

COBSAMP

Usage: COBSAMP *source-code-file-name*

This is a basic macro for compiling a COBOL source file using the definition files contained in subvolume ZMQSLIB. For example, to compile the program AMQ0GBR0, use COBSAMP AMQ0GBR0. If the compilation is successful, the macro produces an object file with the last character of the file name replaced by the letter O; for example, AMQ0GBRO.

BCOBSAMP

Usage: BCOBSAMP *exe-file-name*

This is a basic macro used to bind an object with the user libraries in ZMQSLIB. For example, to bind the compiled sample AMQ0GBRO, use BCOBSAMP AMQ0GBR. The macro produces an executable called *exe-file-name* AMQ0GBR.

CCBSMPLS

Usage: CCBSMPLS

This TACL macro compiles each of the COBOL sample source code files.

BCBSMPLS

Usage: BIND /IN BCBSMPLS/

Preparing and running samples

This bind input file binds each of the COBOL sample object files into executables.

BUILDCOB

Usage: BUILDCOB

This TACL macro compiles and binds all of the COBOL sample files using the macros CCBSMPLS and BCBSMPLS.

Building TAL sample programs

The subvolume ZMQSSMPL contains the following files to be used for building sample TAL programs.

TALSAMP

Usage: TALSAMP *source-code-file-name*

This is a basic macro for compiling a TAL source file using the definition files contained in subvolume ZMQSLIB. For example, to compile the program ZMQWRITT, use TALSAMP ZMQWRITT. If the compilation is successful, the macro produces an object file with the last character of the file name replaced by the letter O; for example, ZMQWRITO.

BTALSAMP

Usage: BTALSAMP *exe-file-name*

This is a basic macro used to bind an object with the user libraries in ZMQSLIB. For example, to bind the compiled sample ZMQWRITO, use BTALSAMP ZMQWRIT.

CTLSMPLS

Usage: CTLSMPLS

This TACL macro compiles each of the TAL sample source code files.

BTLSMPLS

Usage: BIND /IN BTLSMPLS/

This bind input file binds each of the TAL sample object files into executables.

BUILDTAL

Usage: BUILDTAL

This TACL macro compiles and binds all of the TAL sample files using the macros CTLSMPLS and BTLSMPLS.

Windows

For information about MQSeries for Windows, see the following:

- *MQSeries for Windows V2.0 User's Guide.*
- *MQSeries for Windows V2.1 User's Guide.*

Running the sample programs

Before you can run any of the sample programs, a queue manager must be created and the default definitions set up. This is explained in the *MQSeries System Administration* book for MQSeries for AIX, HP-UX, OS/2, Sun Solaris, and Windows NT; for other platforms, see the appropriate *System Management Guide*.

On all platforms except AS/400

The samples need a set of queues to work with. Either use your own queues or run the sample MQSC file amqscos0.tst to create a set.

Preparing and running samples

To do this on UNIX systems and Digital OpenVMS, enter:

```
runmqsc QManagerName <amqscos0.tst >/tmp/sampobj.out
```

Check the sampobj.out file to ensure that there are no errors.

To do this on OS/2 and Windows NT enter:

```
runmqsc QManagerName <amqscos0.tst > sampobj.out
```

Check the sampobj.out file to ensure that there are no errors. This file will be found in your current directory.

To do this on Tandem NSK enter:

```
runmqsc -i $SYSTEM.ZMQSSMPL.AMQSCOMA
```

Check the sampobj.out file to ensure that there are no errors. This file will be found in your current directory.

The sample applications can now be run. Enter the name of the sample application followed by any parameters, for example:

```
amqsput myqueue qmanagername
```

o where myqueue is the name of the queue on which the messages are going to be put, and qmanagername is the queue manager that owns myqueue.

See the description of the individual samples for information on the parameters that each of them expects.

On AS/400

You can use your own queues when you run the samples, or you can run the sample program 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 using data from member PUT in file AMQSDATA of library QMQMSAMP, use a command like:

```
CALL PGM(QMQM/AMQSPUT4) PARM('QMQMSAMP/AMQSDATA(PUT)')
```

Note: For a compiled module to use the IFS file system specify the option SYSIFCOPT (*IFSIO) on CRTCMOD, then the file name, passed as a parameter, must be specified in the following format:

```
home/me/myfile
```

The sample data only applies to the C/400[®] sample programs.

Length of queue name

For the COBOL sample programs, when you pass queue names as parameters, you must provide 48 characters, padding with blank characters if necessary. Anything other than 48 characters causes the program to fail with reason code 2085.

Inquire, Set, and Echo examples

For the Inquire, Set, and Echo examples, the sample definitions cause the C versions of these samples to be triggered. If you want the COBOL versions you must change the process definitions:

```
SYSTEM.SAMPLE.INQPROCESS
```

```
SYSTEM.SAMPLE.SETPROCESS
```

```
SYSTEM.SAMPLE.ECHOPROCESS
```


On OS/2, Windows NT, and UNIX do this by editing the `amqscos0.tst` file and changing the C executable file names to the COBOL executable file names before using the `runmqsc` command above.

On AS/400, you can use the `CHGMQMPCR` command (described in the *MQSeries for AS/400 System Administration* book), or edit and run `AMQSAMP4` with the alternative definition.

The Put sample programs

The Put sample programs put messages on a queue using the `MQPUT` call. See “Features demonstrated in the sample programs” on page 360 for the names of these programs.

Running the `amqsput` and `amqsputc` samples

These programs each take 2 parameters:

1. The name of the target queue (required)
2. The name of the queue manager (optional)

If a queue manager is not specified, `amqsput` connects to the default queue manager and `amqsputc` connects to the queue manager identified by an environment variable or the client channel definition file. To run these programs, enter one of the following:

```
amqsput myqueue qmanagername
amqsputc myqueue qmanagername
```

where `myqueue` is the name of the queue on which the messages are going to be put, and `qmanagername` is the queue manager that owns `myqueue`.

Running the `amqsputw` sample

This program has no visible interface; all messages are put in the output file.

This program takes 4 parameters:

1. The name of the output file (required)
2. The name of the input file (required)
3. The name of the queue manager (required)
4. The name of the target queue (optional)

To run `amqsputw` from the Windows program manager:

1. Select File and click on Run...
2. On the run dialog, enter into the command line entry field the program name followed by the parameters.

For example:

```
amqsputw outfile.out infile.in qmanagername myqueue
```

where:

`outfile.out` is used to hold the messages generated when the program runs.

`infile.in` contains the data to be put onto the target queue. Each line of data is put as a message. This must be an ASCII file.

`qmanagername` is the queue manager that owns `myqueue`.

`myqueue` is the name of the target queue on which the messages are going to be put. If you don't enter a queue name, the default queue for the queue manager is used.

Put samples

Here is an example of what you would see in the output file if you supplied a target queue name:

```
Sample AMQSPUTW start
Output file "OUTFILE.OUT" opened
Input file "INFILE.IN" opened
Queue Manager name "QMANAGERNAME" will be used
target queue is MYQUEUE
MQPUT OK - message contents: <AMQSPUTW: Windows Client Test Message 1>
MQPUT OK - message contents: <AMQSPUTW: Windows Client Test Message 2>
MQPUT OK - message contents: <AMQSPUTW: Windows Client Test Message 3>
Sample AMQSPUTW end
```

Here is an example of what you would see in the output file if you did not enter a target queue name (for example, amqspuw outfil2.out c:\infil2.in qmanagernam2):

```
Sample AMQSPUTW start
Output file "OUTFIL2.OUT" opened
Input file "C:\INFIL2.IN" opened
Queue Manager name "QMANAGERNAM2" will be used
No parameter for Queue Name. Default Queue Name will be used
target queue is QDEF2.Q
MQPUT OK - message contents: <AMQSPUTW: Windows Client Test Message 1>
MQPUT OK - message contents: <AMQSPUTW: Windows Client Test Message 2>
MQPUT OK - message contents: <AMQSPUTW: Windows Client Test Message 3>
Sample AMQSPUTW end
```

where QDEF2.Q is the name of the default queue for the queue manager.

It is important *always* to look in the output file to see what has happened as there is no visible indication of success or failure when you run this program.

Running the amq0put sample

The COBOL version does not have any parameters. It connects to the default queue manager and when you run it you are prompted:

```
Please enter the name of the target queue
```

It takes input from StdIn and adds each line of input to the target queue. A blank line indicates there is no more data.

Running the AMQSPUT4 C sample

The C program creates messages by reading data from a member of a source file. You must specify the name of the file as a parameter when you start the program. The structure of the file must be:

```
queue name
text of message 1
text of message 2
:
text of message n
blank line
```

A sample of input for the put samples is supplied in library QMQMSAMP file AMQSDATA member PUT.

Note: Remember that queue names are case sensitive. All the queues created by the sample file create program AMQSAMP4 have names created in uppercase characters.

Put samples

The C program puts messages on the queue named in the first line of the file—you could use the supplied queue `SYSTEM.SAMPLE.LOCAL`. The program puts the text of each of the following lines of the file into separate datagram messages, and stops when it reads a blank line at the end of the file.

Using the example data file the command is:

```
CALL PGM(QMQM/AMQSPUT4) PARM('QMMSAMP/AMQSDATA(PUT)')
```

Running the AMQ0PUT4 COBOL sample

The COBOL program creates messages by accepting data from the keyboard. To start the program, call the program and give the name of your target queue as a program parameter. The program accepts input from the keyboard into a buffer and creates a datagram message for each line of text. The program stops when you enter a blank line at the keyboard.

Design of the Put sample program

The program uses the `MQOPEN` call with the `MQOO_OUTPUT` option to open the target queue for putting messages. If it cannot open the queue, the program outputs 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 input, 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 Distribution List sample program

The Distribution List sample `amqsptl0` gives an example of putting a message on several message queues. It is based on the `MQPUT` sample, `amqsput0`.

Running the Distribution List sample, amqsptl0

The Distribution List sample runs in a similar way to the Put samples. It takes the following parameters:

- The names of the queues
- The names of the queue managers

These values are entered as pairs. For example:

```
amqsptl0 queue1 qmanagername1 queue2 qmanagername2
```

The queues are opened using `MQOPEN` and messages are put to the queues using `MQPUT`. Reason codes are returned if any of the queue or queue manager names are not recognized.

Remember to define channels between queue managers so that messages can flow between them. The sample program does not do that for you!

Design of the Distribution List sample

Put Message Records (`MQPMRs`) specify message attributes on a per destination basis. The sample chooses to provide values for *MsgId* and *CorrelId*, and these override the values specified in the `MQMD` structure.

Distribution List sample

The *PutMsgRecFields* field in the MQPMO structure indicates which fields are present in the MQPMRs:

```
MLONG PutMsgRecFields=MQPMRF_MSG_ID + MQPMRF_CORREL_ID;
```

Next, the sample allocates the response records and object records. The object records (MQORs) require at least one pair of names and an even number of names, that is, *ObjectName* and *ObjectQMgrName*.

The next stage involves connecting to the queue managers using MQCONN. The sample attempts to connect to the queue manager associated with the first queue in the MQOR; if this fails, it goes through the object records in turn. You are informed if it is not possible to connect to any queue manager and the program exits.

The target queues are opened using MQOPEN and the message is put to these queues using MQPUT. Any problems and failures are reported in the response records (MQRRs).

Finally, the target queues are closed using MQCLOSE and the program disconnects from the queue manager using MQDISC. The same response records are used for each call stating the *CompCode* and *Reason*.

The Browse sample programs

The Browse sample programs browse messages on a queue using the MQGET call. See "Features demonstrated in the sample programs" on page 360 for the names of these programs.

OS/2, UNIX systems, Digital OpenVMS, and Windows NT

The C version of the program takes 2 parameters

1. The name of the source queue (necessary)
2. The name of the queue manager (optional)

If a queue manager is not specified, it will connect to the default one. For example, enter one of the following:

```
amqsgbr myqueue qmanagername
amqsgbrc myqueue qmanagername
amq0gbr0 myqueue
```

where myqueue is the name of the queue that the messages will be viewed from, and qmanagername is the queue manager that owns myqueue.

If you omit the qmanagername, when running the C sample, it will assume that the default queue manager owns the queue.

The COBOL version does not have any parameters. It connects to the default queue manager and when you run it you are prompted:

```
Please enter the name of the target queue
```

Only the first 50 characters of each message are displayed, followed by - - - truncated when this is the case.

AS/400

Each 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.

The C samples allow for specification of queue manager name, generally as the second parameter, in a similar fashion to the NT samples. For example:

```
CALL PGM(QMQM/AMQSTRG4) PARM('SYSTEM.SAMPLE.TRIGGER' 'QM01')
```

If a queue manager is not specified, it will connect to the default one. This is also relevant to the RPG samples. However, with RPG samples you must supply a queue manager name rather than allowing it to default.

Design of the Browse sample program

The program opens the target queue using the MQOPEN call with the MQOO_BROWSE option. If it cannot open the queue, the program outputs an error message 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:

MQGMO_BROWSE_NEXT

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.

MQGMO_NO_WAIT

The program does not wait if there are no messages on the queue.

MQGMO_ACCEPT_TRUNCATED_MSG

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 *MsgId* and *CorrelId* 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 MQRC_NO_MSG_AVAILABLE reason code and the program displays a warning message. If the MQGET call fails, the program displays an error message that contains the reason code.

The program then closes the queue using the MQCLOSE call.

The Browser sample program

The Browser sample program is written as a utility not just to demonstrate a technique. It reads and outputs both the message descriptor and the message content fields of all the messages on a queue. See “Features demonstrated in the sample programs” on page 360 for the names of these programs.

This program takes 2 parameters:

1. The name of the source queue
2. The name of the queue manager

Both input parameters for this program are mandatory. For example, enter one of the following:

```
amqsbcg myqueue qmanagername  
amqsbcgc myqueue qmanagername
```

where `myqueue` is the name of the queue on which the messages are going to be browsed, and `qmanagername` is the queue manager that owns `myqueue`.

It reads each message from the queue and outputs the following to the stdout:

```
Formatted message descriptor fields  
Message data (dumped in hex and, where possible, character format)
```

The program is restricted to printing the first 32767 characters of the message, and will fail with the reason ‘truncated msg’ if a longer message is read.

See the *System Management Guide* for your platform, for examples of the output from this utility.

The Get sample programs

The Get sample programs get messages from a queue using the MQGET call. See “Features demonstrated in the sample programs” on page 360 for the names of these programs.

Running the amqsget and amqsgetc samples

These programs each take two parameters:

1. The name of the source queue (required)
2. The name of the queue manager (optional)

If a queue manager is not specified, `amqsget` connects to the default queue manager, and `amqsgetc` connects to the queue manager identified by an environment variable or the client channel definition file.

To run these programs, enter one of the following:

```
amqsget myqueue qmanagername  
amqsgetc myqueue qmanagername
```

where `myqueue` is the name of the queue from which the program will get messages, and `qmanagername` is the queue manager that owns `myqueue`.

If you omit the `qmanagername`, the programs assume the default, or, in the case of the MQI client, the queue manager identified by an environment variable or the client channel definition file.

Running the amqsgetw sample

This program has no visible interface, all messages are put in the output file, not to stdout.

This program takes 3 parameters:

1. The name of the output file (required)
2. The name of the queue manager (required)
3. The name of the target queue (optional)

To run amqsgetw from the Windows 3.1 program manager:

1. Select File and click on Run...
2. On the run dialog, enter into the command line entry field the program name followed by the parameters.

For example:

```
amqsgetw outfile.out qmanagename myqueue
```

where:

outfile.out is used to hold the messages generated when the program runs.

qmanagename is the queue manager that owns myqueue.

myqueue is the name of the target queue from which the program will get messages. If you do not enter a queue name, the default queue for the queue manager is used.

Here is an example of the contents of the output file:

```
Sample AMQSGETW start
Output file "OUTFILE.OUT" opened
Queue Manager name "QMANAGERNAME" will be used
Queue Name "MYQUEUE" will be used
MQGET OK - message contents: <AMQSPUTW: Windows Client Test Message 1>
MQGET OK - message contents: <AMQSPUTW: Windows Client Test Message 2>
MQGET OK - message contents: <AMQSPUTW: Windows Client Test Message 3>
no more messages
Sample AMQSGETW end
```

It is important *always* to look in the output file to see what has happened as there is no visible indication of success or failure when you run this program.

Running the amq0get sample

The COBOL version does not have any parameters. It connects to the default queue manager and when you run it you are prompted:

Please enter the name of the source queue

Each program removes messages from the queue you specify when you call the program. 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 the queue is empty or an MQI call fails.

Get samples

Running the AMQSGET4 and the AMQ0GET4 samples

The Get sample programs get messages from a queue using the MQGET call. The programs are named:

C language	AMQSGET4
COBOL language	AMQ0GET4

Each program removes messages from the queue you specify when you call the program. 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 the queue is empty or an MQI call fails.

An example of a command to call the C program is:

```
CALL PGM(QMQM/AMQSGET4) PARM('SYSTEM.SAMPLE.LOCAL')
```

Design of the Get sample program

The program opens the target queue using the MQOPEN call with the MQOO_INPUT_AS_Q_DEF option. If it cannot open the queue, the program displays an error message containing the reason code returned by the MQOPEN call.

For each message on the queue, the program uses the MQGET call to remove the message from the queue, then displays the data contained in the message. The MQGET call uses the MQGMO_WAIT option, specifying a *WaitInterval* 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 MQRC_NO_MSG_AVAILABLE reason code.

The program demonstrates how you must clear the *MsgId* and *CorrelId* 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 MQRC_NO_MSG_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 the queue using the MQCLOSE call.

The Reference Message sample programs

The reference message samples allow a large object to be transferred from one node to another (usually on different systems) without the need for the object to be stored on MQSeries queues at either the source or the destination nodes.

A set of sample programs is provided to demonstrate how reference messages can be 1) put to a queue, 2) received by message exits, and 3) taken from a queue. The sample programs use reference messages to move files. If you want to move other objects such as databases, or if you want to perform security checks, you must

Reference Message samples

define your own exit, based on our sample, amqsxrm. The following sections describe the Reference Message sample programs.

There are four versions of the reference message exit sample program. The one to use depends on the platform on which the channel is running. If the sender channel is running on:

MQSeries Version 5 products (excluding MQSeries for AS/400 and MQSeries for OS/390)

Use amqsxrma at the sending end. Use amqsxrma at the receiving end if the receiver is running under MQSeries Version 5 products (excluding MQSeries for AS/400 and MQSeries for OS/390) or amqsxrm4 if the receiver is running under MQSeries for AS/400.

MQSeries for Windows (not MQSeries for Windows NT)

Use amqsqrma at the receiving end if the receiver is running under MQSeries Version 5 products (excluding MQSeries for AS/400 and MQSeries for OS/390) or amqsqrm4 if the receiver is running under MQSeries for AS/400.

If you use amqsqrma or amqsqrm4 a model queue with the name SYSTEM.DEFAULT.MODEL.PERMDYN.QUEUE and queue definition type of PERMDYN must exist at the receiving end. You can create this queue using the MQSC command:

```
def qm(system.default.model.permdyn.queue) deftype(permdyn)
```

Note: In the following sections references to amqsxrma also apply to amqsqrma and references to AMQSXRM4 also apply to AMQSQRM4.

Notes for AS/400 users

To receive a reference message using the sample message exit, specify a file in the root file system of IFS or any sub-directory so that a stream file can be created. The sample message exit on AS/400 creates the file, converts the data to EBCDIC, and sets the code page to your system code page. You then have the option of copying this file to the QSYS.LIB file system using the CPYFRMSTMF command. For example:

```
CPYFRMSTMF FROMSTMF('JANEP/TEST.TXT')
           TOMBR('qsys.lib.janep.lib/test.fie/test.mbr') MBROPT(*REPLACE)
           CVTDTA(*NONE)
```

Note that the CPYFRMSTMF command does not create the file. You must create it before running this command.

If you send a file from QSYS.LIB no changes are required to the samples. For any other file system ensure that the CCSID specified in the CodedCharSetId field in the MQRMH structure matches the bulk data you are sending.

When using the integrated file system, create program modules with the SYSIFCOPT(*IFSIO) option set. If you want to move database or fixed-length record files, define your own exit based on the supplied sample AMQSXRM4.

Reference Message samples

Running the Reference Message samples

The reference message samples run as follows:

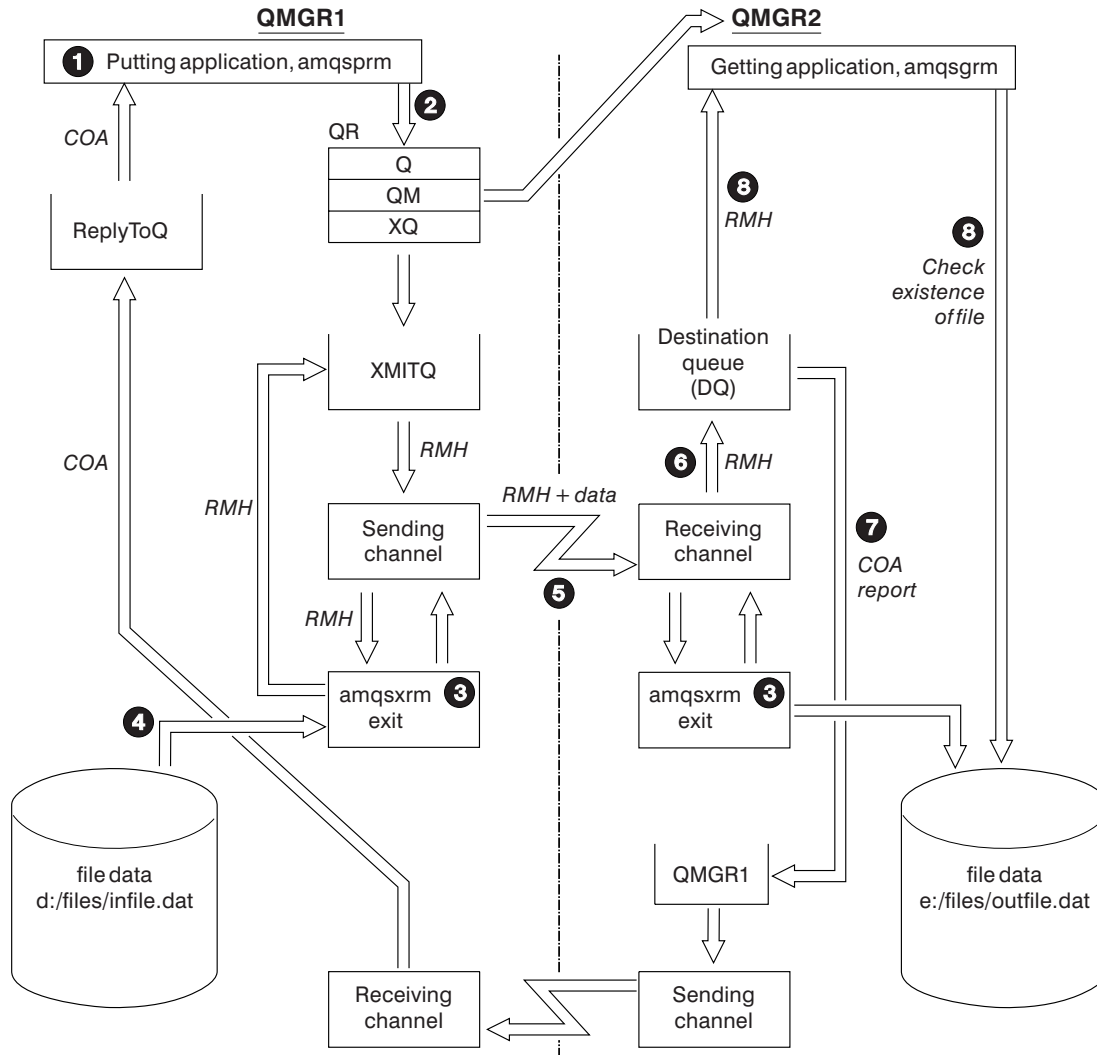


Figure 34. Running the reference message samples

1. Set up the environment to start the listeners, channels, and trigger monitors, and define your channels and queues.

For the purposes of describing how to set-up the reference message example this refers to the sending machine as MACHINE1 with a queue manager called QMGR1 and the receiving machine as MACHINE2 with a queue manager called QMGR2.

Note: The following definitions allow a reference message to be built to send a file with an object type of FLATFILE from queue manager QMGR1 to QMGR2 and to recreate the file as defined in the call to AMQSPRM (or AMQSPRMA on AS/400). The reference message (including the file data) is sent using channel CHL1 and transmission queue XMITQ and placed on queue DQ. Exception and COA reports are sent back to QMGR1 using the channel REPORT and transmission queue QMGR1.

Reference Message samples

The application that receives the reference message (AMQSGRM or AMQSGRMA on the AS/400) is triggered using the initiation queue INITQ and process PROC. You need to ensure the CONNAME fields are set correctly and the MSGEXIT field reflects your directory structure, depending on machine type and where the MQSeries product is installed.

The MQSC definitions have used an OS/2 style for defining the exits, if you are using MQSC on the AS/400 you will need to modify these accordingly. It is important to note that the message data FLATFILE is case sensitive and the sample will not work unless it is in uppercase.

On machine MACHINE1, queue manager QMGR1

MQSC syntax

```
define chl(chl1) chltype(sdr) trptype(tcp) conname('machine2') xmitq(xmitq)
msgdata(FLATFILE) msgexit('d:\mqm\tools\c\samples\bin\amqsxrm(MsgExit  )')

define ql(xmitq) usage(xmitq)

define chl(report) chltype(rcvr) trptype(tcp) replace

define qr(qr) rname(dq) rqmname(qmgr2) xmitq(xmitq) replace
```

AS/400 command syntax

Note: If you do not specify a queue manager name the system uses the default queue manager.

```
CRTMQMCHL CHLNAME(CHL1) CHLTYPE(*SDR) MQMNAME(QMGR1) +
          REPLACE(*YES) TRPTYPE(*TCP) +
          CONNAME('MACHINE2(60501)') TMQNAME(XMITQ) +
          MSGEXIT(QMQM/AMQSXR4) MSGUSRDATA(FLATFILE)

CRTMQMQ QNAME(XMITQ) QTYPE(*LCL) MQMNAME(QMGR1) +
        REPLACE(*YES) USAGE(*TMQ)

CRTMQMCHL CHLNAME(REPORT) CHLTYPE(*RCVR) +
          MQMNAME(QMGR1) REPLACE(*YES) TRPTYPE(*TCP)

CRTMQMQ QNAME(QR) QTYPE(*RMT) MQMNAME(QMGR1) +
        REPLACE(*YES) RMTQNAME(DQ) +
        RMTMQMNAME(QMGR2) TMQNAME(XMITQ)
```

On machine MACHINE2, queue manager QMGR2

MQSC syntax

```
define chl(chl1) chltype(rcvr) trptype(tcp)
msgexit('d:\mqm\tools\c\samples\bin\amqsxrm(MsgExit)')
msgdata(flatfile)

define chl(report) chltype(sdr) trptype(tcp) conname('MACHINE1')
xmitq(qmgr1)

define ql(initq)

define ql(qmgr1) usage(xmitq)

define pro(proc) applicid('d:\mqm\tools\c\samples\bin\amqsgrm')

define ql(dq) initq(initq) process(proc) trigger trigtype(first)
```

Reference Message samples

AS/400 command syntax

Note: If you do not specify a queue manager name the system uses the default queue manager.

```
CRTMQMCHL  CHLNAME(CHL1) CHLTYPE(*RCVR) MQMNAME(QMGR2) +
           REPLACE(*YES) TRPTYPE(*TCP) +
           MSGEXIT(QMQM/AMQSXR4) MSGUSRDATA(FLATFILE)

CRTMQMCHL  CHLNAME(REPORT) CHLTYPE(*SDR) MQMNAME(QMGR2) +
           REPLACE(*YES) TRPTYPE(*TCP) +
           CONNAME('MACHINE1(60500)') TMQNAME(QMGR1)

CRTMQMQ    QNAME(INITQ) QTYPE(*LCL) MQMNAME(QMGR2) +
           REPLACE(*YES) USAGE(*NORMAL)

CRTMQMQ    QNAME(QMGR1) QTYPE(*LCL) MQMNAME(QMGR2) +
           REPLACE(*YES) USAGE(*TMQ)

CRTMQMPRC  PRCNAME(PROC) MQMNAME(QMGR2) REPLACE(*YES) +
           APPID('QMQM/AMQSGRM4')

CRTMQMQ    QNAME(DQ) QTYPE(*LCL) MQMNAME(QMGR2) +
           REPLACE(*YES) PRCNAME(PROC) TRGENBL(*YES) +
           INITQNAME(INITQ)
```

2. Once the above MQSeries objects have been created:
 - a. Where applicable to the platform, start the listener for the sending and receiving queue managers
 - b. Start the channels CHL1 and REPORT
 - c. On the receiving queue manager start the trigger monitor for the initiation queue INITQ
3. Invoke the put reference message sample AMQSPRM (AMQSPRMA on the AS/400) from the command line using the following parameters:

```
-m  Name of the local queue manager, this defaults to the default queue manager
-i  Name and location of source file
-o  Name and location of destination file
-q  Name of queue
-g  Name of queue manager where the queue, defined in the -q parameter exists This
    defaults to the queue manager specified in the -m parameter
-t  Object type
-w  Wait interval, that is, the waiting time for exception and COA reports from the
    receiving queue manager
```

For example, to use the sample with the objects defined above you would use the following parameters:

```
-mQMGR1 -iInput File -oOutput File -qQR -tFLATFILE -w120
```

Increasing the waiting time will allow time for a large file to be sent across a network before the putting program times out.

```
amqsprpm -q QR -m QMGR1 -i d:\x\file.in -o d:\y\file.out -t FLATFILE
```

AS/400 users: On the AS/400 use the following command:

```
CALL      PGM(QMQM/AMQSPRM4) PARM('-mQMGR1' +
           '-i/refmsgs/rmsg1' +
           '-o/refmsgs/rmsgx' '-qQR' +
           '-gQMGR1' '-tFLATFILE' '-w15')
```

Reference Message samples

To use the IFS, use the following commands:

```
CRTCMOD MODULE(QMQMSAMP/AMQSXR4) SRCFILE(QMQMSAMP/QCSRC) SYSIFCOPT(*IFSIO)
CRTCMOD MODULE(QMQMSAMP/AMQSGRM4) SRCFILE(QMQMSAMP/QCSRC) SYSIFCOPT(*IFSIO)
```

You may use the root directory, but it is recommended you create one using the CRTDIR command.

When calling the putting program the output file name will need to reflect the IFS naming convention, for instance /TEST/FILENAME will create a file called FILENAME in the directory TEST.

Note: You can use either a forward slash (/) or a dash (-) when specifying parameters.

For example:

```
amqsprpm /i d:\files\infile.dat /o e:\files\outfile.dat /q QR
/m QMGR1 /w 30 /t FLATFILE
```

Note: For UNIX platforms, you must use two slashes (\\) instead of one to denote the destination file directory. Therefore, the above command looks like this:

```
amqsprpm -i /files/infile.dat -o e:\\files\\outfile.dat -q QR
-m QMGR1 -w 30 -t FLATFILE
```

This demonstrates the following:

- The reference message will be put to queue QR on queue manager QMGR1.
 - The source file and path is d:\files\infile.dat and exists on the system where the example command is issued.
 - If the queue QR is a remote queue, the reference message is sent to another queue manager, on a different system, where a file is created with the name and path e:\files\outfile.dat. The contents of this file are the same as the source file.
 - amqsprpm waits for 30 seconds for a COA report from the destination queue manager.
 - The object type is flatfile, so the channel used to move messages from the queue QR must specify this in the *MsgData* field.
4. When you define your channels, select the message exit at both the sending and receiving ends to be amqsxrm. This is defined on MQSeries for OS/2 Warp, and Windows NT as follows:

```
msgexit('<pathname>\amqsxrm.dll(MsgExit)')
```

This is defined on MQSeries for AIX, HP-UX, Compaq Tru64 UNIX, and Sun Solaris as follows:

```
msgexit('<pathname>/amqsxrm(MsgExit)')
```

If a pathname is specified, the complete name must be specified (with .dll on OS/2). If a pathname is not specified, it is assumed that the program is in the path specified in the qm.ini file (or, on MQSeries for Windows NT, the path specified in the registry). This is explained fully in the *MQSeries Intercommunication* book.

- 5. The channel exit reads the reference message header and finds the file that it is referring to.
- 6. It can then choose to segment the file before sending it down the channel along with the header. On MQSeries for AIX, HP-UX, Compaq Tru64 UNIX, and Sun

Reference Message samples

Solaris, you must change the group owner of the target directory to 'mqm' so that the sample message exit can create the file in that directory. Also, change the permissions of the target directory to allow mqm group members to write to it. The file data is not stored on the MQSeries queues.

7. When the last segment of the file is processed by the receiving message exit, the reference message is put to the destination queue specified by `amqsprm`. If this queue is triggered (that is, the definition specifies *Trigger*, *InitQ*, and *Process* queue attributes), the program specified by the PROC parameter of the destination queue is triggered. The program to be triggered must be defined in the *ApplId* field of the *Process* attribute.
8. When the reference message reaches the destination queue (DQ), a COA report is sent back to the putting application (`amqsprm`).
9. The get reference message sample, `amqsgrm`, gets messages from the queue specified in the input trigger message and checks the existence of the file.

Design of the Put Reference Message sample (`amqsprma.c`, AMQSPRM4)

This sample creates a reference message that refers to a file and puts it on a specified queue:

1. The sample connects to a local queue manager using MQCONN.
2. It then opens (MQOPEN) a model queue which is used to receive report messages.
3. The sample builds a reference message containing the values required to move the file, for example, the source and destination file names and the object type. As an example, the sample shipped with MQSeries builds a reference message to send the file `d:\x\file.in` from QMGR1 to QMGR2 and to recreate the file as `d:\y\file.out` using the following parameters:

```
amqsprm -q QR -m QMGR1 -i d:\x\file.in -o d:\y\file.out -t FLATFILE
```

Where QR is a remote queue definition that refers to a target queue on QMGR2.

Note: For UNIX platforms, you must use two slashes (\\) instead of one to denote the destination file directory. Therefore, the above command looks like this:

```
amqsprm -q QR -m QMGR1 -i /x/file.in -o d:\\y\\file.out -t FLATFILE
```

4. The reference message is put (without any file data) to the queue specified by the `/q` parameter. If this is a remote queue, the message is put to the corresponding transmission queue.
5. The sample waits, for the duration of time specified in the `/w` parameter (which defaults to 15 seconds), for COA reports, which, along with exception reports, are sent back to the dynamic queue created on the local queue manager (QMGR1).

Design of the Reference Message Exit sample (`amqsxrma.c`, AMQSXRM4)

This sample recognizes reference messages with an object type that matches the object type in the message exit user data field of the channel definition. For these messages, the following happens:

- At the sender or server channel, the specified length of data is copied from the specified offset of the specified file into the space remaining in the agent buffer

after the reference message. If the end of the file is not reached, the reference message is put back on the transmission queue after updating the *DataLogicalOffset* field.

- At the requester or receiver channel, if the *DataLogicalOffset* field is zero and the specified file does not exist, it is created. The data following the reference message is added to the end of the specified file. If the reference message is not the last one for the specified file, it is discarded. Otherwise, it is returned to the channel exit, without the appended data, to be put on the target queue.

For sender and server channels, if the *DataLogicalLength* field in the input reference message is zero, the remaining part of the file, from *DataLogicalOffset* to the end of the file, is to be sent along the channel. If it is not zero, only the length specified is sent.

If an error occurs (for example, if the sample is unable to open a file), *MQCXP.ExitResponse* is set to *MQXCC_SUPPRESS_FUNCTION* so that the message being processed is put to the dead-letter queue instead of continuing to the destination queue. A feedback code is returned in *MQCXP.Feedback* and returned to the application that put the message in the *Feedback* field of the message descriptor of a report message. This is because the putting application requested exception reports by setting *MQRO_EXCEPTION* in the *Report* field of the MQMD.

If the encoding or *CodedCharacterSetId* (CCSID) of the reference message is different from that of the queue manager, the reference message is converted to the local encoding and CCSID. In our sample, *amqsprmq*, the format of the object is *MQFMT_STRING*, so *amqsxrm* converts the object data to the local CCSID at the receiving end before the data is written to the file.

The format of the file being transferred should not be specified as *MQFMT_STRING* if the file contains multibyte characters (for example, DBCS or Unicode). This is because a multibyte character could be split when the file is segmented at the sending end. To transfer and convert such a file, the format should be specified as something other than *MQFMT_STRING* so that the reference message exit does not convert it and the file should be converted at the receiving end when the transfer is complete.

Compiling the Reference Message Exit sample

To compile *amqsxrma*, use the following commands:

On AIX:

```
$cc -d -I/usr/mqm/inc amqsxrma.c
$ld -o amqsxrm amqsxrma.o -bE:amqsxrm.exp -H512 -T512 \
-e MQStart -bM:SRE -lc -ls -lmqm
```

On Compaq Tru64 UNIX Version 4.0:

In a non-threaded environment:

```
$ cc -std1 -c -I /opt/mqm/inc amqsxrma.c
$ cc -std1 -shared -o amqsxrma amqsxrma.o -L /opt/mqm/lib -lmqm -e MQStart -lc
```

In a threaded environment:

```
$ cc -std1 -c -I /opt/mqm/inc amqsxrma.c
$ cc -std1 -shared -pthread -o amqsxrma_r amqsxrma.o -L /opt/mqm/lib \
-lmqm_r -e MQStart -lc
```

Reference Message samples

On Compaq Tru64 UNIX Version 5.0:

```
$ cc -std1 -c -I /opt/mqm/inc amqsrma.c
$ cc -std1 -shared -pthread -o amqsrma amqsrma.o -L /opt/mqm/lib \
-lmqm -e MQStart -lc
```

On HP-UX:

```
$ cc -c -Aa +z -I/opt/mqm/inc amqsrma.c
$ ld -b -o amqsrma amqsrma.o -z +b : -lmqm -lc
```

On AS/400: To create the module use the following command:

```
CRTCMOD MODULE(MYLIB/AMQSRMA) SRCFILE(QMQMSAMP/QCSRC)
TERASPACE (*YES *TSIFC)
```

Note: To create your module so that it uses the IFS file system add the option SYSIFCOPT (*IFSIO).

To create the program for use with nonthreaded channels use the following command: CRTPGM PGM(MYLIB/AMQSRMA) BNDSRVPGM(QMQM/LIBMQM)

To create the program for use with threaded channels use the following command: CRTPGM PGM(MYLIB/AMQSRMA) BNDSRVPGM(QMQM/LIBMQM_R)

On Sun Solaris:

```
$ cc -c -KPIC -I/opt/mqm/inc amqsrma.c
$ ld -G -o amqsrma amqsrma.o -dy -lmqm -lc -lnsl -ldl
```

Design of the Get Reference Message sample (amqsrma.c, AMQSGRM4)

The program logic is as follows:

1. The sample is triggered and extracts the queue and queue manager names from the input trigger message.
2. It then connects to the specified queue manager using MQCONN and opens the specified queue using MQOPEN.
3. The sample issues MQGET with a wait interval of 15 seconds within a loop to get messages from the queue.
4. If a message is a reference message, the sample checks the existence of the file that has been transferred.
5. It then closes the queue and disconnects from the queue manager.

The Request sample programs

The Request sample programs demonstrate client/server processing. The samples are the clients that put request messages on a target server queue that is processed by a server program. They wait for the server program to put a reply message on a reply-to queue.

The Request samples put a series of request messages on target server queue using the MQPUT call. These messages specify the local queue, SYSTEM.SAMPLE.REPLY as the reply-to queue, which can be a local or remote queue. The programs wait for reply messages, then display them. Replies are sent only if the target server queue is being processed by a server application, or if an application is triggered for that purpose (the Inquire, Set, and Echo sample programs are designed to be triggered). The C sample waits 1 minute (the COBOL 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 both samples can end without getting any replies. See “Features demonstrated in the sample programs” on page 360 for the names of the Request sample programs.

Running the amqsreq0.c, amqsreq, and amqsreqc samples

The C version of the program takes 2 parameters:

1. The name of the target server queue (necessary)
2. The name of the queue manager (optional)

If a queue manager is not specified, it will connect to the default one. For example, enter one of the following:

```
amqsreq myqueue qmanagername
amqsreqc myqueue qmanagername
amq0req0 myqueue
```

where myqueue is the name of the target server queue, and qmanagername is the queue manager that owns myqueue.

If you omit the qmanagername, when running the C sample, it will assume that the default queue manager owns the queue.

Running the amq0req0.cbl sample

The COBOL version does not have any parameters. It connects to the default queue manager and when you run it you are prompted:

```
Please enter the name of the target server queue
```

The program takes its input from StdIn and adds each line to the target server queue, taking each line of text as the content of a request message. The program ends when a null line is read.

Running the AMQSREQ4 sample

The C program creates messages by taking data from stdin (the keyboard) with a blank line terminating input. The program takes up to three parameters: the name of the target queue (required), the queue manager name (optional), and the reply-to queue name (optional). If no queue manager name is specified, the default queue manager is used. If no reply-to queue is specified, the SYSTEM.SAMPLE.REPLY queue is used.

Here is an example of how to call the C sample program, specifying the reply-to queue, but letting the queue manager default:

```
CALL PGM(QMQM/AMQSREQ4) PARM('SYSTEM.SAMPLE.LOCAL' ' ' 'SYSTEM.SAMPLE.REPLY')
```

Note: Remember that queue names are case sensitive. All the queues created by the sample file create program AMQSAMP4 have names created in uppercase characters.

Running the AMQ0REQ4 sample

The COBOL program creates messages by accepting data from the keyboard. To start the program, call the program and specify the name of your target queue as a parameter. The program accepts input from the keyboard into a buffer and creates a request message for each line of text. The program stops when you enter a blank line at the keyboard.

Request samples

Running the Request sample using triggering

If the sample is used with triggering and one of the Inquire, Set, or Echo sample programs, the line of input must be the queue name of the queue that you want the triggered program to access.

OS/2, UNIX systems, and Windows NT

To run the samples using triggering:

1. Start the trigger monitor program RUNMQTRM in one session (the initiation queue SYSTEM.SAMPLE.TRIGGER is available for you to use).
2. Start the amqsreq program in another session.
3. Make sure you have defined a target server queue.

The sample queues available to you to use as the target server queue for the request sample to put messages are:

- SYSTEM.SAMPLE.INQ - for the Inquire sample program
- SYSTEM.SAMPLE.SET - for the Set sample program
- SYSTEM.SAMPLE.ECHO - for the Echo sample program

These queues have a trigger type of FIRST, so if there are already messages on the queues before you run the Request sample, server applications are not triggered by the messages you send.

4. Make sure you have defined a queue for the Inquire, Set or Echo sample program to use.

This means that the trigger monitor is ready when the request sample sends a message.

Note: The sample process definitions created using RUNMQSC and the amqscos0.tst file cause the C samples to be triggered. Change the process definitions in amqscos0.tst and use RUNMQSC with this updated file if the COBOL versions are required.

Figure 35 on page 391 demonstrates how the Request and Inquire samples can be used together.

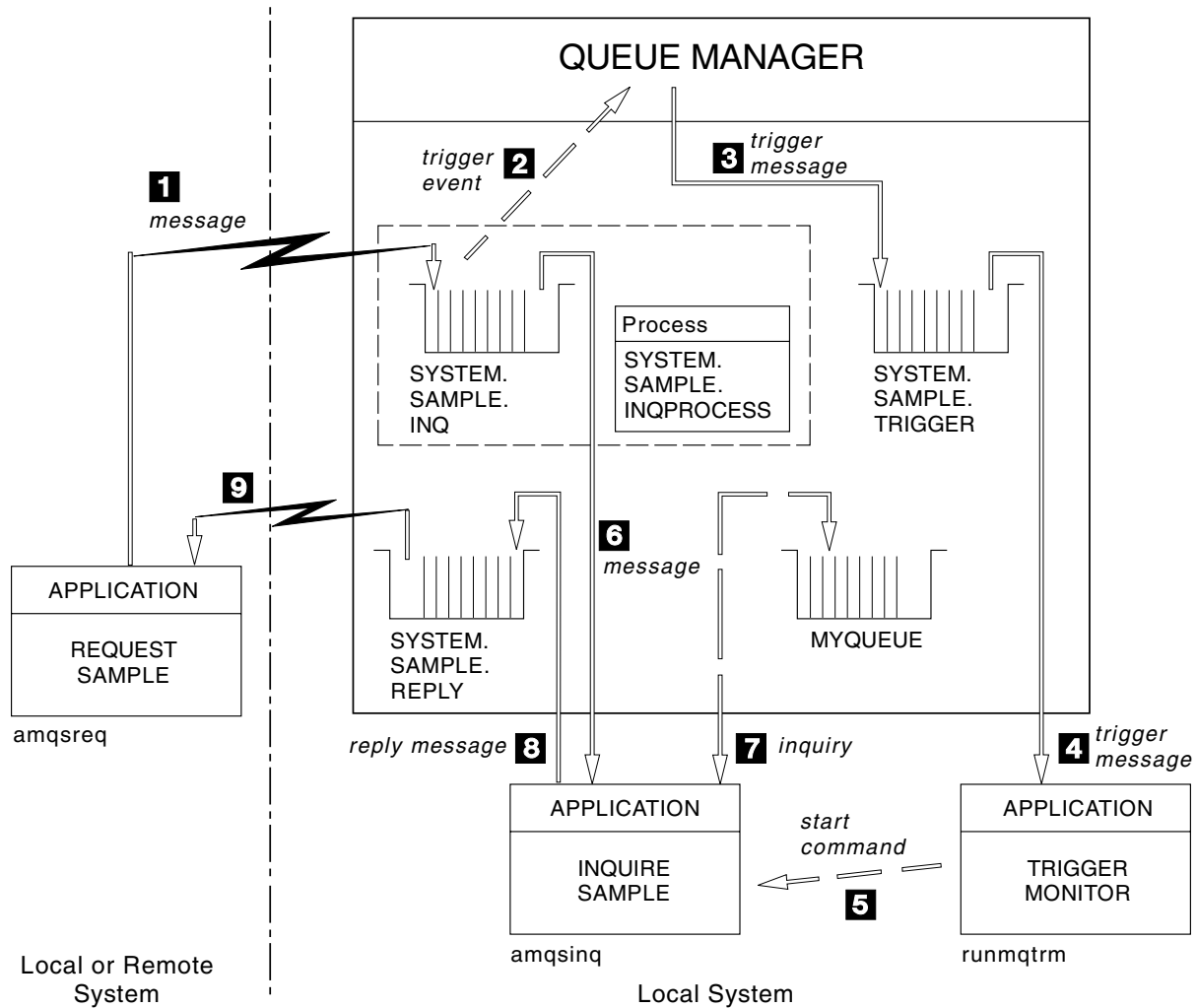


Figure 35. Request and Inquire samples using triggering

In Figure 35 the Request sample puts messages on to the target server queue, SYSTEM.SAMPLE.INQ, and the Inquire sample queries the queue, MYQUEUE. Alternatively, you can use one of the sample queues defined when you ran amqsos0.tst, or any other queue you have defined, for the Inquire sample.

Note: The numbers in Figure 35 show the sequence of events.

To run the Request and Inquire samples, using triggering:

1. Check that the queues you want to use are defined. Run amqsos0.tst, to define the sample queues, and define a queue MYQUEUE.
2. Run the trigger monitor command RUNMQTRM:
`RUNMQTRM -m qmanagername -q SYSTEM.SAMPLE.TRIGGER`
3. Run the request sample
`amqsreq SYSTEM.SAMPLE.INQ`

Note: The process object defines what is to be triggered. If the client and server are not running on the same platform, any processes started by the trigger monitor must define *AppType*, otherwise the server takes its default definitions (that is, the type of application that is normally associated with the server machine) and causes a failure.

Request samples

For example, if the trigger monitor is running on a Windows NT client and wants to send a request to an OS/2 server, MQAT_WINDOWS_NT must be defined otherwise OS/2 uses its default definitions (that is, MQAT_OS2) and the process fails.

For a list of application types, see the *MQSeries Application Programming Reference* manual.

4. Enter the name of the queue you want the Inquire sample to use:
MYQUEUE
5. Enter a blank line (to end the Request program).
6. The request sample will then display a message, containing the data the Inquire program obtained from MYQUEUE.

If you wish, you can use more than one queue. In this case, you enter the names of the other queues at step 4.

For more information on triggering see “Chapter 14. Starting MQSeries applications using triggers” on page 191.

AS/400

To try the samples using triggering on AS/400, start the sample trigger server, AMQSERV4, in one job, then start AMQSREQ4 in another. This means that the trigger server is ready when the Request sample program sends a message.

Notes:

1. The sample definitions created by AMQSAMP4 cause the C versions of the samples to be triggered. If you want to trigger the COBOL versions, you must change the process definitions SYSTEM.SAMPLE.ECHOPROCESS, SYSTEM.SAMPLE.INQPROCESS, and SYSTEM.SAMPLE.SETPROCESS. You can use the CHGMQMPCR command (described in the *MQSeries for AS/400 System Administration* book) to do this, or edit and run your own version of AMQSAMP4.
2. Source code for AMQSERV4 is supplied for the C language only. However, a compiled version (that you can use with the COBOL samples) is supplied in library QMQM.

You could put your request messages on 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 36 on page 394. Using the example data file the command to issue the C program request to this server is:

```
CALL PGM(QMQMSAMP/AMQSREQ4) PARM('QMMSAMP/AMQSDATA(ECHO)')
```

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 AMQSTRG4 (or its command line equivalent STRMQMTRM – see the *MQSeries for AS/400 System Administration* book) instead of AMQSERV4 to submit the job instead, but potential job submission delays could make it less easy to follow what is happening.

- Run the SYSTEM.SAMPLE.INQUIRE and SYSTEM.SAMPLE.SET sample programs. Using the example data file the commands to issue the C program requests to these servers are, respectively:

```
CALL PGM(QMQMSAMP/AMQSREQ4) PARM('QMMSAMP/AMQSDATA(INQ)')
CALL PGM(QMQMSAMP/AMQSREQ4) PARM('QMMSAMP/AMQSDATA(SET)')
```

These sample queues also have a trigger type of FIRST.

Design of the Request sample program

The program opens the target server queue so that it can put messages. It uses the MQOPEN call with the MQOO_OUTPUT 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 MQOO_INPUT_EXCLUSIVE 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 MQRO_EXCEPTION_WITH_DATA 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 MQGMO_WAIT, MQGMO_CONVERT, and MQGMO_ACCEPT_TRUNCATED options. The *waitInterval* is 5 minutes in the COBOL version, and 1 minute in the C version, 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 MQRC_NO_MSG_AVAILABLE reason code. The call also uses the MQGMO_ACCEPT_TRUNCATED_MSG option, so messages longer than the declared buffer size are truncated.

The program demonstrates how you must clear the *MsgId* and *CorrelId* 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 until either the MQGET call returns the MQRC_NO_MSG_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 target server queue and the reply-to queue using the MQCLOSE call.

Inquire samples

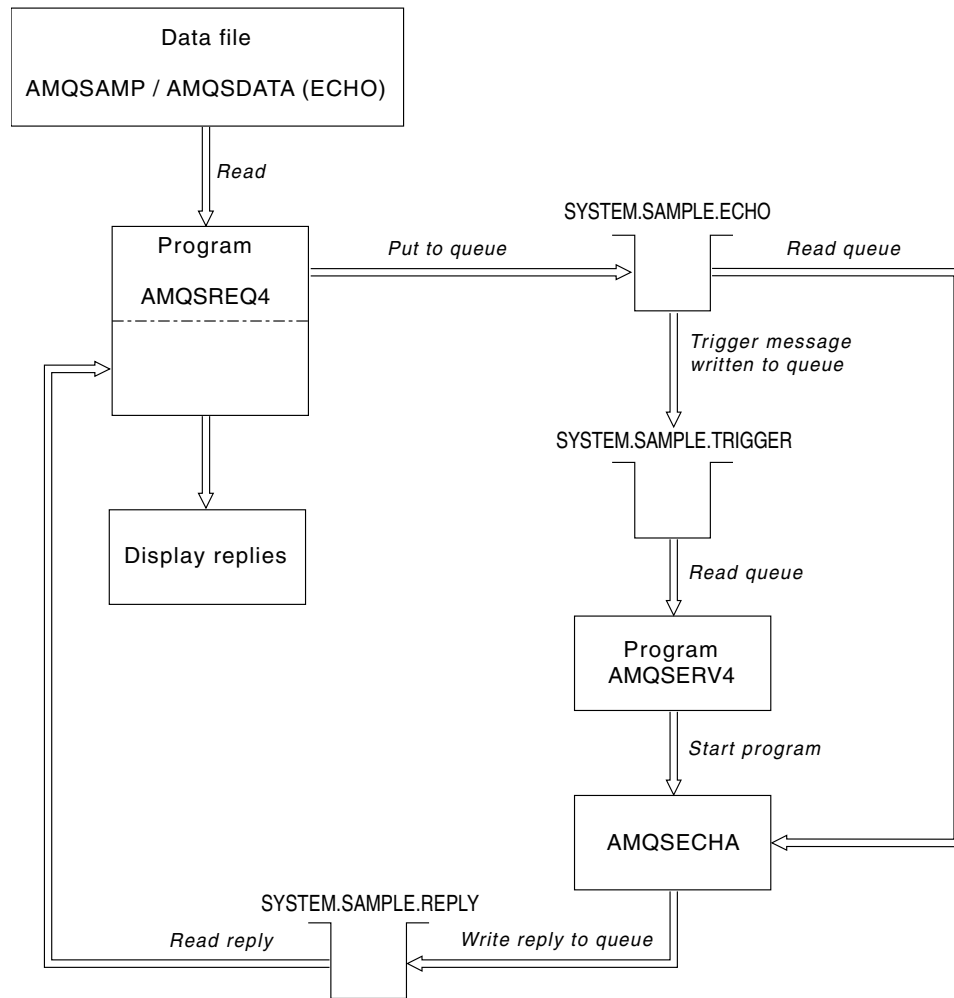


Figure 36. Sample Client/Server (Echo) program flowchart

The Inquire sample programs

The Inquire sample programs inquire about some of the attributes of a queue using the MQINQ call. See “Features demonstrated in the sample programs” on page 360 for the names of these programs.

These programs are intended to run as triggered programs, so their only input is an MQTMC2 (trigger message) structure for OS/2, Windows NT, Digital OpenVMS, and UNIX, and an MQTMC structure for AS/400. These structures contain the name of a target queue whose attributes are to be inquired. The C version also uses the queue manager name. The COBOL version uses the default queue manager.

For the triggering process to work, you must ensure that the Inquire sample program you want to use is triggered by messages arriving on queue SYSTEM.SAMPLE.INQ. To do this, specify the name of the Inquire sample program you want to use in the *ApplicId* field of the process definition SYSTEM.SAMPLE.INQPROCESS. For AS/400, you can use the CHGMQMPCR command described in the *MQSeries for AS/400 System Administration* book for this. The sample queue has a trigger type of FIRST; 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:

- For OS/2, UNIX systems, Digital OpenVMS, and Windows NT, start the `runmqtrm` program in one session, then start the `amqsreq` program in another.
- For AS/400, start the `AMQSERV4` program in one session, then start the `AMQSREQ4` program in another. You could use `AMQSTRG4` instead of `AMQSERV4`, but potential job submission delays could make it less easy to follow what is happening.

Use the Request sample programs to send request messages, each containing just a queue name, to queue `SYSTEM.SAMPLE.INQ`. For each request message, the Inquire sample programs send 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.

On AS/400, if the sample input file member `QMQMSAMP.AMQSDATA(INQ)` is used, the last queue named does not exist, so the sample returns a report message with a reason code for the failure.

Design of the Inquire sample program

The program 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 `MQGMO_ACCEPT_TRUNCATED_MSG` and `MQGMO_WAIT` 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 `MQOO_INQ` 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 `MQPUT1` 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 `MQPUT1` call to put a report message on the reply-to queue. In the *Feedback* 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 programs

The Set sample programs inhibit put operations on a queue by using the `MQSET` call to change the queue's *InhibitPut* attribute. See "Features demonstrated in the sample programs" on page 360 for the names of these programs.

Set samples

The programs are intended to run as triggered programs, so their only input is an MQTMC2 (trigger message) structure that contains the name of a target queue whose attributes are to be inquired. The C version also uses the queue manager name. The COBOL version uses the default queue manager.

For the triggering process to work, you must ensure that the Set sample program you want to use is triggered by messages arriving on queue SYSTEM.SAMPLE.SET. To do this, specify the name of the Set sample program you want to use in the *ApplicId* field of the process definition SYSTEM.SAMPLE.SETPROCESS. The sample queue has a trigger type of FIRST; 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:

- For OS/2, UNIX systems, Digital OpenVMS, and Windows NT, start the **runmqtrm** program in one session, then start the amqsreq program in another.
- For AS/400, start the AMQSERV4 program in one session, then start the AMQSREQ4 program in another. You could use AMQSTRG4 instead of AMQSERV4, but potential job submission delays could make it less easy to follow what is happening.

Use the Request sample programs to send request messages, each containing just a queue name, to queue SYSTEM.SAMPLE.SET. For each request message, the Set sample programs send 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

The program 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 MQGMO_ACCEPT_TRUNCATED_MSG and MQGMO_WAIT 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 MQOO_SET option. The program then uses the MQSET call to set the value of the *InhibitPut* attribute of the target queue to MQQA_PUT_INHIBITED.

If the MQSET call is successful, the program uses the MQPUT1 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 MQPUT1 call to put a report message on the reply-to queue. In the *Feedback* 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 Echo sample programs

The Echo sample programs echo a message from a message queue to the reply queue. See “Features demonstrated in the sample programs” on page 360 for the names of these programs.

The programs are intended to run as triggered programs.

On AS/400, OS/2, UNIX systems, and Windows NT, their only input is an MQTMC2 (trigger message) structure that contains the name of a target queue and the queue manager. The COBOL version uses the default queue manager.

On AS/400, 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 CHGMQMPRC command, described in the *MQSeries for AS/400 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 Echo sample is not triggered by the messages you send.

When you have set the definition correctly, first start AMQSERV4 in one job, then start AMQSREQ4 in another. You could use AMQSTRG4 instead of AMQSERV4, 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 programs

The program 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 MQGMO_ACCEPT_TRUNCATED_MSG, MQGMO_CONVERT, and MQGMO_WAIT 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 line of input, the program then reads the text into a buffer and uses the MQPUT1 call to put a request message, containing the text of that line, on to the reply-to queue.

If the MQGET call fails, the program puts a report message on the reply-to queue, setting the *Feedback* field of the message descriptor to the reason code returned by the MQGET.

Echo samples

When there are no messages remaining on the request queue, the program closes that queue and disconnects from the queue manager.

On AS/400, the 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* parameters, 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 Data-Conversion sample program

The data-conversion sample program is a skeleton of a data conversion exit routine. See “Features demonstrated in the sample programs” on page 360 for the names of these programs.

Design of the data-conversion sample

Each data-conversion exit routine converts a single named message format. This skeleton is intended as a wrapper for code fragments generated by the data-conversion exit generation utility program.

The utility produces one code fragment for each data structure; several such structures make up a format, so several code fragments are added to this skeleton to produce a routine to do data conversion of the entire format.

The program then checks whether the conversion is a success or failure, and returns the values required to the caller.

The Triggering sample programs

The function provided in the triggering sample is a subset of that provided in the trigger monitor in the **runmqtrm** program. See “Features demonstrated in the sample programs” on page 360 for the names of these programs.

Running the amqstrg0.c, amqstrg, and amqstrgc samples

The program takes 2 parameters:

1. The name of the initiation queue (necessary)
2. The name of the queue manager (optional)

If a queue manager is not specified, it will connect to the default one. A sample initiation queue will have been defined when you ran amqscos0.tst. the name of that queue is SYSTEM.SAMPLE.TRIGGER, and you can use it when you run this program.

Note: The function in this sample is a subset of the full triggering function that is supplied in the **runmqtrm** program.

Running the AMQSTRG4 sample

This is a trigger monitor for the AS/400 environment. It submits an AS/400 job for the application to be started, but this means there is a processing overhead associated with each trigger message.

AMQSTRG4 (in QCSRC) takes two parameters: the name of the initiation queue it is to serve, and the name of the queue manager (optional). AMQSAMP4 (in QCLSRC) defines a sample initiation queue, SYSTEM.SAMPLE.TRIGGER, that you can use when you try the sample programs.

Using the example trigger queue the command to issue is:

```
CALL PGM(QMQM/AMQSTRG4) PARM('SYSTEM.SAMPLE.TRIGGER')
```

Alternatively, you can use the command line (CL) equivalent STRMQMTRM – see the *MQSeries for AS/400 System Administration* book.

Design of the triggering sample

The triggering sample program opens the initiation queue using the MQOPEN call with the MQOO_INPUT_AS_Q_DEF option. It gets messages from the initiation queue using the MQGET call with the MQGMO_ACCEPT_TRUNCATED_MSG and MQGMO_WAIT options, specifying an unlimited wait interval. The program clears the *MsgId* and *CorrelId* fields before each MQGET call to get messages in sequence.

When it has retrieved a message from the initiation queue, the program tests the message:

- It checks the size of the message to make sure it is the same size as an MQTM structure.
- It checks the *ApplType* field to make sure it contains the value MQAT_UNIX.

If either of these tests fail, the program displays a warning.

For valid trigger messages, the triggering sample copies data from these fields: *ApplicId*, *EnvrData*, *Version*, and *ApplType*. The last two of these fields are numeric, so the program creates character replacements to use in an MQTMC2 structure for OS/2, UNIX, and Windows NT, and in an MQTMC structure for AS/400.

The triggering sample issues a start command to the application specified in the *ApplicId* field of the trigger message, and passes an MQTMC2 or MQTMC (a character version of the trigger message) structure. In OS/2, UNIX systems, and Windows NT, the *EnvrData* field is used as an extension to the invoking command string. In AS/400, it is used as job submission parameters, for example, the job priority or the job description. See the *MQSeries for AS/400 System Administration* book for a discussion of job priority and job description.

Finally, the program closes the initiation queue.

Running the AMQSERV4 sample

This is a trigger server for the AS/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.

AMQSERV4 takes two parameters: the name of the initiation queue it is to serve, and the name of the queue manager (optional). AMQSAMP4 defines a sample initiation queue, SYSTEM.SAMPLE.TRIGGER, that you can use when you try the sample programs.

Using the example trigger queue the command to issue is:

```
CALL PGM(QMQM/AMQSERV4) PARM('SYSTEM.SAMPLE.TRIGGER')
```

Triggering samples

Design of the trigger server

The design of the trigger server is similar to that of the trigger monitor, except the trigger server:

- Allows MQAT_CICS as well as MQAT_OS400 applications
- Calls AS/400 applications in its own job (or uses STRCICSUSR to start CICS applications) rather than submitting an AS/400 job
- For CICS applications, substitutes the *EnvData*, for example, to specify the CICS region, from the trigger message in the STRCICSUSR command
- Opens the initiation queue for shared input, so many trigger servers can run at the same time

Note: Programs started by AMQSERV4 must not use the MQDISC call because this will stop the trigger server. If programs started by AMQSERV4 use the MQCONN call, they will get the MQRC_ALREADY_CONNECTED reason code.

Ending the triggering sample programs on AS/400

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') MQMNAME 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 queue managers.

Program amqscos0.tst 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 queue manager you want to use. You must also set up a message channel between your two queue managers; for information on how to do this, see the *MQSeries Intercommunication* book.

The Request sample programs put their own local queue manager name in the *ReplyToQMgr* field of messages they send. The Inquire and Set samples send reply messages to the queue and message queue manager named in the *ReplyToQ* and *ReplyToQMgr* fields of the request messages they process.

Database coordination samples

Two samples are provided which demonstrate how MQSeries can coordinate both MQSeries updates and database updates within the same unit of work:

1. AMQSXAS0 (in C) or AMQ0XAS0 (in COBOL), which updates a single database within an MQSeries unit of work.
2. AMQSXAG0 (in C) or AMQ0XAG0 (in COBOL), AMQSXAB0 (in C) or AMQ0XAB0 (in COBOL), and AMQSXAF0 (in C) or AMQ0XAF0 (in COBOL), which together update two databases within an MQSeries unit of work, showing how multiple databases can be accessed. These samples are provided

Database coordination samples

to show the use of the MQBEGIN call, mixed SQL and MQSeries calls, and where and when to connect to a database.

Figure 37 shows how the samples provided are used to update databases:

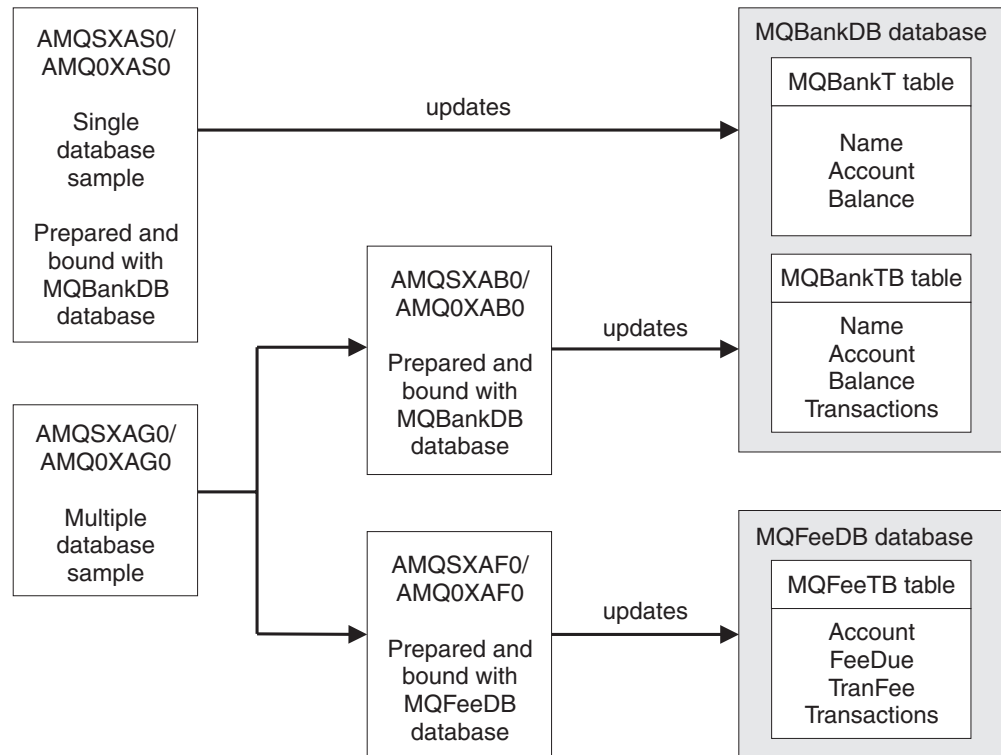


Figure 37. The database coordination samples

The programs read a message from a queue (under syncpoint), then, using the information in the message, obtain the relevant information from the database and update it. The new status of the database is then printed.

The program logic is as follows:

1. Use name of input queue from program argument
2. Connect to default queue manager (or optionally supplied name in C) using MQCONN
3. Open queue (using MQOPEN) for input while no failures
4. Start a unit of work using MQBEGIN
5. Get next message (using MQGET) from queue under syncpoint
6. Get information from databases
7. Update information from databases
8. Commit changes using MQCOMMIT
9. Print updated information (no message available counts as failure, and loop ends)
10. Close queue using MQCLOSE
11. Disconnect from queue using MQDISC

Database coordination samples

SQL cursors are used in the samples, so that reads from the databases (that is, multiple instances) are locked whilst a message is being processed, thus multiple instances of these programs can be run simultaneously. The cursors are explicitly opened, but implicitly closed by the MQCMIT call.

The single database sample (AMQXSAS0 or AMQ0XAS0) has no SQL CONNECT statements and the connection to the database is implicitly made by MQSeries with the MQBEGIN call. The multiple database sample (AMQXSAG0 or AMQ0XAG0, AMQXSAB0 or AMQ0XAB0, and AMQSXAF0 or AMQ0XAF0) has SQL CONNECT statements, as some database products allow only one active connection. If this is not the case for your database product, or if you are accessing a single database in multiple database products, the SQL CONNECT statements can be removed.

The samples are prepared with the IBM DB2 database product, so they may need some modification to work with other database products.

The SQL error checking uses routines in UTIL.C and CHECKERR.CBL supplied by DB2. These must be compiled or replaced before compiling and linking.

Note: If you are using the Micro Focus COBOL source CHECKERR.MFC for SQL error checking, you must change the program ID to uppercase, that is CHECKERR, for AMQ0XAS0 to link correctly.

Creating the databases and tables

The databases and tables must be created before the samples can be compiled. To create the databases, use the normal method for your database product, for example:

```
DB2 CREATE DB MQBankDB
DB2 CREATE DB MQFeeDB
```

Create the tables using SQL statements as follows:

In C:

```
EXEC SQL CREATE TABLE MQBankT(Name          VARCHAR(40) NOT NULL,
                                Account       INTEGER   NOT NULL,
                                Balance       INTEGER   NOT NULL,
                                PRIMARY KEY (Account));

EXEC SQL CREATE TABLE MQBankTB(Name         VARCHAR(40) NOT NULL,
                                Account       INTEGER   NOT NULL,
                                Balance       INTEGER   NOT NULL,
                                Transactions  INTEGER,
                                PRIMARY KEY (Account));

EXEC SQL CREATE TABLE MQFeeTB(Account      INTEGER   NOT NULL,
                                FeeDue       INTEGER   NOT NULL,
                                TranFee     INTEGER   NOT NULL,
                                Transactions  INTEGER,
                                PRIMARY KEY (Account));
```

In COBOL:

```
EXEC SQL CREATE TABLE
MQBankT(Name          VARCHAR(40) NOT NULL,
         Account       INTEGER   NOT NULL,
         Balance       INTEGER   NOT NULL,
         PRIMARY KEY (Account))
END-EXEC.
```

```
EXEC SQL CREATE TABLE
```

```

MQBankTB(Name      VARCHAR(40) NOT NULL,
          Account   INTEGER   NOT NULL,
          Balance   INTEGER   NOT NULL,
          Transactions INTEGER,
          PRIMARY KEY (Account))
END-EXEC.

EXEC SQL CREATE TABLE
MQFeeTB(Account      INTEGER   NOT NULL,
          FeeDue      INTEGER   NOT NULL,
          TranFee     INTEGER   NOT NULL,
          Transactions INTEGER,
          PRIMARY KEY (Account))
END-EXEC.

```

Fill in the tables using SQL statements as follows:

```

EXEC SQL INSERT INTO MQBankT VALUES ('Mr Fred Bloggs',1,0);
EXEC SQL INSERT INTO MQBankT VALUES ('Mrs S Smith',2,0);
EXEC SQL INSERT INTO MQBankT VALUES ('Ms Mary Brown',3,0);
:
EXEC SQL INSERT INTO MQBankTB VALUES ('Mr Fred Bloggs',1,0,0);
EXEC SQL INSERT INTO MQBankTB VALUES ('Mrs S Smith',2,0,0);
EXEC SQL INSERT INTO MQBankTB VALUES ('Ms Mary Brown',3,0,0);
:
EXEC SQL INSERT INTO MQFeeTB VALUES (1,0,50,0);
EXEC SQL INSERT INTO MQFeeTB VALUES (2,0,50,0);
EXEC SQL INSERT INTO MQFeeTB VALUES (3,0,50,0);
:

```

Note: For COBOL, use the same SQL statements but add END_EXEC at the end of each line.

Precompiling, compiling, and linking the samples

The .SQC files (in C) and .SQB files (in COBOL) must be precompiled and bound against the appropriate database to produce the .C or .CBL files. To do this, use the normal method for your database product, as shown below.

Precompiling in C

```

db2 connect to MQBankDB
db2 prep AMQSXAS0.SQC
db2 connect reset

```

```

db2 connect to MQBankDB
db2 prep AMQSXAB0.SQC
db2 connect reset

```

```

db2 connect to MQFeeDB
db2 prep AMQSXAF0.SQC
db2 connect reset

```

Precompiling in COBOL

```

db2 connect to MQBankDB
db2 prep AMQ0XAS0.SQB bindfile target ibmcob
db2 bind AMQ0XAS0.BND
db2 connect reset

```

```

db2 connect to MQBankDB
db2 prep AMQ0XAB0.SQB bindfile target ibmcob
db2 bind AMQ0XAB0.BND
db2 connect reset

```

Database coordination samples

```
db2 connect to MQFeeDB
db2 prep AMQ0XAF0.SQB bindfile target ibmcob
db2 bind AMQ0XAF0.BND
db2 connect reset
```

Compiling and linking

The following sample commands use the symbol <DB2TOP>. <DB2TOP> represents the installation directory for the DB2 product.

- On AIX the directory path is:
/usr/lpp/db2_05_00
- On HP-UX and Sun Solaris the directory path is:
/opt/IBMdb2/V5.0
- On Windows NT and OS/2 the directory path depends on the path chosen when installing the product. If you chose the default settings the path is:
c:\sql11b

Note: Before issuing the link command on Windows NT or OS/2 ensure that the LIB environment variable contains paths to the DB2 and MQSeries libraries.

Copy the following files into a temporary directory:

- The amqsxag0.c file from your MQSeries installation

Note: This file can be found in the following directories:

- On UNIX:
<MQMTOP>/samp/xatm
- On Windows NT and OS/2:
<MQMTOP>\tools\c\samples\xatm
- The .c files that you have obtained by precompiling the .sqc source files, amqsxas0.sqc, amqsxaf0.sqc, and amqsxab0.sqc
- The files util.c and util.h from your DB2 installation.

Note: These files can be found in the directory:

<DB2TOP>/samples/c

Build the object files for each .c file using the following compiler command for the platform that you are using:

- AIX
xlc_r -I<MQMTOP>/inc -I<DB2TOP>/include -c -o
<FILENAME>.o <FILENAME>.c
- HP-UX
cc -Aa +z -I<MQMTOP>/inc -I<DB2TOP>/include -c -o
<FILENAME>.o <FILENAME>.c
- OS/2
icc /c /I<MQMTOP>\tools\c\include /I<DB2TOP>\include <FILENAME>.c
- Sun Solaris
cc -Aa -KPIC -mt -I<MQMTOP>/inc -I<DB2TOP>/include -c -o
<FILENAME>.o <FILENAME>.c
- Windows NT
cl /c /I<MQMTOP>\tools\c\include /I<DB2TOP>\include
<FILENAME>.c

Database coordination samples

Build the amqsxag0 executable using the following link command for the platform that you are using:

- AIX

```
xlc_r -H512 -T512 -L<DB2TOP>/lib -ldb2 -L<MQMTP>/lib
-lmqm util.o amqsxaf0.o amqsxab0.o amqsxag0.o -o amqsxag0
```
- HP-UX Revision 10.20

```
ld -E -L<DB2TOP>/lib -ldb2 -L<MQMTP>/lib -lmqm -lc /lib/crt0.o
util.o amqsxaf0.o amqsxab0.o amqsxag0.o -o amqsxag0
```
- HP-UX Revision 11.00

```
ld -E -L<DB2TOP>/lib -ldb2 -L<MQMTP>/lib -lmqm -lc -lpthread -lcl
/lib/crt0.o util.o amqsxaf0.o amqsxab0.o amqsxag0.o -o amqsxag0
```
- OS/2

```
ilink util.obj amqsxaf0.obj amqsxab0.obj amqsxag0.obj mqm.lib
db2api.lib /out:amqsxag0.exe
```
- Sun Solaris

```
cc -mt -L<DB2TOP>/lib -ldb2 -L<MQMTP>/lib
-lmqm -lmqmzse -lmqmcs -lthread -lsocket -lc -lnsl -ldl util.o
amqsxaf0.o amqsxab0.o amqsxag0.o -o amqsxag0
```
- Windows NT

```
link util.obj amqsxaf0.obj amqsxab0.obj amqsxag0.obj mqm.lib db2api.lib
/out:amqsxag0.exe
```

Build the amqsxas0 executable using the following compile and link commands for the platform that you are using:

- AIX

```
xlc_r -H512 -T512 -L<DB2TOP>/lib -ldb2
-L<MQMTP>/lib -lmqm util.o amqsxas0.o -o amqsxas0
```
- HP-UX Revision 10.20

```
ld -E -L<DB2TOP>/lib -ldb2 -L<MQMTP>/lib -lmqm -lc
/lib/crt0.o util.o amqsxas0.o -o amqsxas0
```
- HP-UX Revision 11.00

```
ld -E -L<DB2TOP>/lib -ldb2 -L<MQMTP>/lib -lmqm -lc -lpthread
-lcl /lib/crt0.o util.o amqsxas0.o -o amqsxas0
```
- OS/2

```
ilink util.obj amqsxas0.obj mqm.lib db2api.lib /out:amqsxas0.exe
```
- Sun Solaris

```
cc -mt -L<DB2TOP>/lib -ldb2 -L<MQMTP>/lib
-lqm -lmqmzse -lmqmcs -lthread -lsocket -lc -lnsl -ldl util.o
amqsxas0.o -o amqsxas0
```
- Windows NT

```
link util.obj amqsxas0.obj mqm.lib db2api.lib /out:amqsxas0.exe
```

Additional information

If you are working on AIX or HP-UX and wish to access Oracle, use the xlc_r compiler and link to libmqm_r.a.

Running the samples

Before the samples can be run, the queue manager must be configured with the database product you are using. For information about how to do this, see the *MQSeries System Administration* book.

Database coordination samples

C samples

Messages must be in the following format to be read from a queue:

```
UPDATE Balance change=nnn WHERE Account=nnn
```

AMQSPUT can be used to put the messages on the queue.

The database coordination samples take two parameters:

1. Queue name (required)
2. Queue manager name (optional)

Assuming that you have created and configured a queue manager for the single database sample called `singDBQM`, with a queue called `singDBQ`, you increment Mr Fred Bloggs's account by 50 as follows:

```
AMQSPUT singDBQ singDBQM
```

Then key in the following message:

```
UPDATE Balance change=50 WHERE Account=1
```

You can put multiple messages on the queue.

```
AMQXSAS0 singDBQ singDBQM
```

The updated status of Mr Fred Bloggs's account is then printed.

Assuming that you have created and configured a queue manager for the multiple-database sample called `multDBQM`, with a queue called `multDBQ`, you decrement Ms Mary Brown's account by 75 as follows:

```
AMQSPUT multDBQ multDBQM
```

Then key in the following message:

```
UPDATE Balance change=-75 WHERE Account=3
```

You can put multiple messages on the queue.

```
AMQXSAG0 multDBQ multDBQM
```

The updated status of Ms Mary Brown's account is then printed.

COBOL samples

Messages must be in the following format to be read from a queue:

```
UPDATE Balance change=snnnnnnnn WHERE Account=nnnnnnnn
```

For simplicity, the Balance change must be a signed eight-character number and the Account must be an eight-character number.

The sample AMQSPUT can be used to put the messages on the queue.

The samples take no parameters and use the default queue manager. It can be configured to run only one of the samples at any time. Assuming that you have configured the default queue manager for the single database sample, with a queue called `singDBQ`, you increment Mr Fred Bloggs's account by 50 as follows:

```
AMQSPUT singDBQ
```

Then key in the following message:

```
UPDATE Balance change=+00000050 WHERE Account=00000001
```

You can put multiple messages on the queue.

```
AMQ0XAS0
```

Type in the name of the queue:

```
singDBQ
```

The updated status of Mr Fred Bloggs's account is then printed.

Assuming that you have configured the default queue manager for the multiple database sample, with a queue called multDBQ, you decrement Ms Mary Brown's account by 75 as follows:

```
AMQSPUT multDBQ
```

Then key in the following message:

```
UPDATE Balance change=-00000075 WHERE Account=00000003
```

You can put multiple messages on the queue.

```
AMQ0XAG0
```

Type in the name of the queue:

```
multDBQ
```

The updated status of Ms Mary Brown's account is then printed.

The CICS transaction sample

A sample CICS transaction program is provided, named amqscic0.ccs for source code and amqscic0 for the executable version. Transactions may be built using the standard CICS facilities. See "Part 3. Building an MQSeries application" on page 283 for details on the commands needed for your platform.

The transaction reads messages from the transmission queue SYSTEM.SAMPLE.CICS.WORKQUEUE on the default queue manager and places them on to the local queue, the name of which is contained in the transmission header of the message. Any failures will be sent to the queue SYSTEM.SAMPLE.CICS.DLQ.

Note: A sample MQSC script amqscic0.tst may be used to create these queues and sample input queues.

TUXEDO samples

Before running these samples, you must build the server environment.

Throughout this section the \ character is used to split long commands over more than one line. Do not enter this character, enter each command as a single line.

Building the server environment

It is assumed that you have a working TUXEDO environment.

To build the server environment for MQSeries for AIX:

1. Create a directory (for example, <APPDIR>) in which the server environment is built and execute all commands in this directory.

TUXEDO samples

- Export the following environment variables, where TUXDIR is the root directory for TUXEDO:

```
$ export CFLAGS="-I /usr/mqm/inc -I /<APPDIR> -L /usr/mqm/lib"
$ export LDOPTS="-lmqm -lmqmzse -lnet -insl -lsocket -lc -ldl"
$ export FIELDTBLS=/usr/mqm/samp/amqstxvx.flds
$ export VIEWFILES=/<APPDIR>/amqstxvx.V
$ export LIBPATH=$TUXDIR/lib:/usr/mqm/lib:/lib
```

- Add the following to the TUXEDO file udataobj/RM

```
MQSeries_XA_RMI:MQRMIXASwitchDynamic: \
/usr/mqm/lib/libmqmxa.a /usr/mqm/lib/libmqm.a
```

- Run the commands:

```
$ mkfldhdr /usr/mqm/samp/amqstxvx.flds
$ viewc /usr/mqm/samp/amqstxvx.v
$ buildtms -o MQXA -r MQSeries_XA_RMI
$ buildserver -o MQSERV1 -f /usr/mqm/samp/amqstx.c \
-f /usr/mqm/lib/libmqm.a \
-r MQSeries_XA_RMI -s MPUT1:MPUT \
-s MGET1:MGET \
-v -bshm
$ buildserver -o MQSERV2 -f /usr/mqm/samp/amqstx.c \
-f /usr/mqm/lib/libmqm.a \
-r MQSeries_XA_RMI -s MPUT2:MPUT \
-s MGET2:MGET \
-v -bshm
$ buildclient -o doputs -f /usr/mqm/samp/amqstx.c \
-f /usr/mqm/lib/libmqm.a
$ buildclient -o dogets -f /usr/mqm/samp/amqstx.c \
-f /usr/mqm/lib/libmqm.a
```

- Edit ubbstxcx.cfg (see Figure 38 on page 412), and add details of the machine name, working directories, and queue manager as necessary:

```
$ tmloadcf -y /usr/mqm/samp/ubbstxcx.cfg
```

- Create the TLOGDEVICE:

```
$tmadmin -c
```

A prompt then appears. At this prompt, enter:

```
> crdl -z /<APPDIR>/TLOG1
```

- Start the queue manager:

```
$ strmqm
```

- Start Tuxedo:

```
$ tmbot -y
```

You can now use the doputs and dogets programs to put messages to a queue and retrieve them from a queue.

To build the server environment for MQSeries for AT&T GIS UNIX and MQSeries for Sun Solaris:

- Create a directory (for example, <APPDIR>) in which the server environment is built and execute all commands in this directory.
- Export the following environment variables, where TUXDIR is the root directory for TUXEDO:

```
$ export CFLAGS="-I /<APPDIR>"
$ export FIELDTBLS=amqstxvx.flds
$ export VIEWFILES=amqstxvx.V
$ export SHLIB_PATH=$TUXDIR/lib:/opt/mqm/lib:/lib
$ export LD_LIBRARY_PATH=$TUXDIR/lib:/opt/mqm/lib:/lib
```

- Add the following to the TUXEDO file udataobj/RM (RM must include /opt/mqm/lib/libmqmcs and /opt/mqm/lib/libmqmzse).

```
MQSeries_XA_RMI:MQRMIXASwitchDynamic: \
/opt/mqm/lib/libmqmxa.a /opt/mqm/lib/libmqm.so \
/opt/tuxedo/lib/libtux.a /opt/mqm/lib/libmqmcs.so \
/opt/mqm/lib/libmqmzse.so
```

4. Run the commands:

```
$ mkfldhdr amqstvx.flds
$ viewc amqstvx.v
$ buildtms -o MQXA -r MQSeries_XA_RMI
$ buildserver -o MQSERV1 -f amqstxsx.c \
-f /opt/mqm/lib/libmqm.so \
-r MQSeries_XA_RMI -s MPUT1:MPUT \
-s MGET1:MGET \
-v -bshm
-l -ldl
$ buildserver -o MQSERV2 -f amqstxsx.c \
-f /opt/mqm/lib/libmqm.so \
-r MQSeries_XA_RMI -s MPUT2:MPUT \
-s MGET2:MGET \
-v -bshm
-l -ldl
$ buildclient -o doputs -f amqstxpx.c \
-f /opt/mqm/lib/libmqm.so \
-f /opt/mqm/lib/libmqmzse.co \
-f /opt/mqm/lib/libmqmcs.so
$ buildclient -o dogets -f amqstxgx.c \
-f /opt/mqm/lib/libmqm.so
-f /opt/mqm/lib/libmqmzse.co \
-f /opt/mqm/lib/libmqmcs.so
```

5. Edit ubbstxcx.cfg (see Figure 38 on page 412), and add details of the machine name, working directories, and Queue Manager as necessary:

```
$ tmloadcf -y ubbstxcx.cfg
```

6. Create the TLOGDEVICE:

```
$ tmadmin -c
```

A prompt then appears. At this prompt, enter:

```
> crdl -z /<APPDIR>/TLOG1
```

7. Start the queue manager:

```
$ strmqm
```

8. Start Tuxedo:

```
$ tmbot -y
```

You can now use the doputs and dogets programs to put messages to a queue and retrieve them from a queue.

To build the server environment for MQSeries for HP-UX:

1. Create a directory (for example, <APPDIR>) in which the server environment is built and execute all commands in this directory.
2. Export the following environment variables, where TUXDIR is the root directory for TUXEDO:

```
$ export CFLAGS="-Aa -D_HPUX_SOURCE"
$ export LDOPTS="-lmqm"
$ export FIELDTBLS=/opt/mqm/samp/amqstvx.flds
$ export VIEWFILES=<APPDIR>/amqstvx.V
$ export SHLIB_PATH=$TUXDIR/lib:/opt/mqm/lib:/lib
$ export LPATH=$TUXDIR/lib:/opt/mqm/lib:/lib
```

3. Add the following to the TUXEDO file udataobj/RM

```
MQSeries_XA_RMI:MQRMIXASwitchDynamic: \
/opt/mqm/lib/libmqmxa.a /opt/mqm/lib/libmqm.sl \
/opt/tuxedo/lib/libtux.sl
```

TUXEDO samples

4. Run the commands:

```
$ mkfldhdr /opt/mqm/samp/amqstxvx.flds
$ viewc /opt/mqm/samp/amqstxvx.v
$ buildtms -o MQXA -r MQSeries_XA_RMI
$ buildserver -o MQSERV1 -f /opt/mqm/samp/amqstxsx.c \
-f /opt/mqm/lib/libmqm.sl \
-r MQSeries_XA_RMI -s MPUT1:MPUT \
-s MGET1:MGET \
-v -bshm
$ buildserver -o MQSERV2 -f /opt/mqm/samp/amqstxsx.c \
-f /opt/mqm/lib/libmqm.sl \
-r MQSeries_XA_RMI -s MPUT2:MPUT \
-s MGET2:MGET \
-v -bshm
$ buildclient -o doputs -f /opt/mqm/samp/amqstxpx.c \
-f /opt/mqm/lib/libmqm.sl
$ buildclient -o dogets -f /opt/mqm/samp/amqstxgx.c \
-f /opt/mqm/lib/libmqm.sl
```

5. Edit ubbstxcx.cfg (see Figure 38 on page 412), and add details of the machine name, working directories, and Queue Manager as necessary:

```
$ tmloadcf -y /opt/mqm/samp/ubbstxcx.cfg
```

6. Create the TLOGDEVICE:

```
$ tmadmin -c
```

A prompt then appears. At this prompt, enter:

```
> crdl -z /<APPDIR>/TLOG1
```

7. Start the queue manager:

```
$ strmqm
```

8. Start Tuxedo:

```
$ tmbot -y
```

You can now use the doputs and dogets programs to put messages to a queue and retrieve them from a queue.

To build the server environment for MQSeries for SINIX and DC/OSx

1. Export the following environment variables where TUXDIR is the root directory for TUXEDO:

```
$ export CFLAGS="-lmqm -lmqmcs -lmqzse -lmqmx \
-lns1 -lsocket -ldl -lmproc -lxt"
```

Note: For DC/OSx, add “-liconv” to the above.

```
$ export FIELDTBLS=amqstxvx.flds
$ export VIEWFILES=amqstxvx.V
$ export VIEWDIR=The path to the directory where the views
are held
$ export TUXDIR=The path to the directory where TUXEDO
is installed (/opt/tuxedo).
$ export CFLAGS="-lmqm -lmqmcs -lmqzse -lmqmx \
-lns1 -lsocket -ldl -lmproc -lxt"
```

2. Add the following to the TUXEDO file udataobj/RM

```
MQSeries_XA_RMI:MQRMIXASwitchDynamic: \
/opt/mqm/lib/libmqmx.so
/opt/mqm/lib/libmqm.so /opt/mqm/lib/libmqmcs.s
```

3. Ensure that your LD_LIBRARY_PATH contains the path to the Tuxedo libraries (/opt/tuxedo/lib), and that it is exported.
4. Ensure that your PATH contains the path to the Tuxedo bin directory (/opt/tuxedo/bin), and that it is exported.

- Run the commands:

```
$ mkfldhdr amqstvx.flds
$ viewc amqstvx.v
```

- Alter the value of the CFLAGS variable:

```
export CFLAGS="$CFLAGS -LDuMQRMIXASwitchDynamic -lmqxa"
```

- Run the commands:

```
$ buildtms -o MQXA -r MQSeries_XA_RMI
$ buildserver -o MQSERV1 -f amqstxsx.c \
-f /opt/mqm/lib/libmqm.so i \
-r MQSeries_XA_RMI -s MPUT1:MPUT \
-s MGET1:MGET \
-v -bshm
$ buildserver -o MQSERV2 -f amqstxsx.c \
-f /opt/mqm/lib/libmqm.so \
-r MQSeries_XA_RMI -s MPUT2:MPUT \
-s MGET2:MGET \
-v -bshm
$ buildclient -o doputs -f amqstxpx.c \
-f /opt/mqm/lib/libmqm.so
$ buildclient -o dogets -f amqstxgx.c \
-f /opt/mqm/lib/libmqm.so
```

- Ensure that your NLS_PATH contains the path to the Tuxedo messages (/opt/tuxedo/locale/C/%N), and that it is exported.
- Edit ubbstxcx.cfg (see Figure 38 on page 412), and add details of the machine name, working directories, and Queue Manager as necessary.
- Set the environment variable TUXCONFIG to the value specified in the MACHINES section of the ubbstxcx.cfg file.
- If you are using the Tuxedo main machine, run the following commands:

```
tmadmin -c
```

At the prompt (>), enter:

```
crdl -z filename
```

where *filename* is the path to the Tuxedo TLOG file.

- Run the following command:

```
$ tmloadcf -y ubbstxcx.cfg
```

- Start the queue manager:

```
$ strmqm
```

- Start Tuxedo:

```
$ tmbot -y
```

You can now use the doputs and dogets programs to put messages to a queue and retrieve them from a queue.

For further information on building the TUXEDO server environment, see the README file in the MQSeries sample directory, /opt/mqm/samp.

TUXEDO samples

```
*RESOURCES
IPCKEY          <IPCKey>

#Example:
#IPCKEY         123456

MASTER         <MachineName>
MAXACCESSERS   20
MAXSERVERS     20
MAXSERVICES    50
MODEL          SHM
LDBAL          N

*MACHINES
DEFAULT:
                APPDIR="<WorkDirectory>"
                TUXCONFIG="<WorkDirectory>/tuxconfig"
                ROOTDIR="<RootDirectory>"

<MachineName>  LMID=<MachineName>
                TLOGDEVICE="<WorkDirectory>/TLOG1"
                TLOGNAME=TLOG

*GROUPS
GROUP1
                LMID=<MachineName>  GRPNO=1
                TMSNAME=MQXA
                OPENINFO="MQSeries_XA_RMI:MYQUEUEMANAGER"

*SERVERS
DEFAULT:
                CLOPT="-A -- -m MYQUEUEMANAGER

MQSERV1        SRVGRP=GROUP1  SRVID=1
MQSERV2        SRVGRP=GROUP1  SRVID=2

*SERVICES
MPUT1
MGET1
MPUT2
MGET2
```

Figure 38. Example of ubbstxcx.cfg file for UNIX systems

Note: Other information that you need to add is identified by <> characters. In this file, the queue manager name has been changed to MYQUEUEMANAGER:

To build the server environment for MQSeries for Windows NT:

Note: Change the fields identified by <> in the following, to the directory paths:

<MQMDIR>

the directory path specified when MQSeries was installed, for example g:\Program Files\MQSeries

<TUXDIR>

the directory path specified when TUXEDO was installed, for example f:\tuxedo

<APPDIR>

the directory path to be used for the sample application, for example f:\tuxedo\apps\mqapp

To build the server environment and samples:

1. Create an application directory in which to build the sample application, for example:

```
f:\tuxedo\apps\mqapp
```

2. Copy the following sample files from the MQSeries sample directory to the application directory:

```
amqstxmn.mak
amqstxen.env
ubbstxcn.cfg
```

3. Edit each of these files to set the directory names and directory paths used on your installation.
4. Edit ubbstxcn.cfg (see Figure 39 on page 414) to add details of the machine name and the Queue Manager that you wish to connect to.
5. Add the following line to the TUXEDO file <TUXDIR>\udataobj\rm

```
MQSeries_XA_RMI;MQRMIXASwitchDynamic;
<MQMDIR>\tools\lib\mqmtux.lib <MQMDIR>\tools\lib\mqm.lib
```

where <MQMDIR> is replaced as above. Although shown here as two lines, the new entry must be one line in the file.

6. Set the following environment variables:

```
TUXDIR=<TUXDIR>
TUXCONFIG=<APPDIR>\tuxconfig
FIELDTBLS=<MQMDIR>\tools\c\samples\amqstxvx.fld
LANG=C
```

7. Create a TLOG device for TUXEDO. To do this, invoke tadmin -c, and enter the command:

```
crdl -z <APPDIR>\TLOG
```

where <APPDIR> is replaced as above.

8. Set the current directory to <APPDIR>, and invoke the sample makefile (amqstxmn.mak) as an external project makefile. For example, with Microsoft Visual C++ Version 2.0, issue the command:

```
msvc amqstxmn.mak
```

Select **build** to build all the sample programs.

TUXEDO samples

```
*RESOURCES
IPCKEY      99999
UID         0
GID         0
MAXACCESSERS 20
MAXSERVERS  20
MAXSERVICES 50
MASTER     SITE1
MODEL       SHM
LDBAL       N

*MACHINES
<MachineName> LMID=SITE1
                TUXDIR="f:\tuxedo"
                APPDIR="f:\tuxedo\apps\mqapp;g:\Program Files\MQSeries\bin"
                ENVFILE="f:\tuxedo\apps\mqapp\amgstxen.env"
                TUXCONFIG="f:\tuxedo\apps\mqapp\tuxconfig"
                ULOGPFX="f:\tuxedo\apps\mqapp\ULOG"
                TLOGDEVICE="f:\tuxedo\apps\mqapp\TLOG"
                TLOGNAME=TLOG
                TYPE="i386NT"
                UID=0
                GID=0

*GROUPS
GROUP1
                LMID=SITE1 GRPNO=1
                TMSNAME=MQXA
                OPENINFO="MQSeries_XA_RMI:MYQUEUEMANAGER"

*SERVERS
DEFAULT: CLOPT="-A -- -m MYQUEUEMANAGER"

MQSERV1    SRVGRP=GROUP1 SRVID=1
MQSERV2    SRVGRP=GROUP1 SRVID=2

*SERVICES
MPUT1
MGET1
MPUT2
MGET2
```

Figure 39. Example of ubbstxcn.cfg file for Windows NT

Note: The directory names and directory paths must be changed to match your installation. The queue manager name MYQUEUEMANAGER should also be changed to the name of the queue manager you wish to connect to. Other information that you need to add is identified by <> characters.

The sample ubbconfig file for MQSeries for Windows NT is listed in Figure 39. It is supplied as ubbstxcn.cfg in the MQSeries samples directory.

The sample makefile (see Figure 40 on page 415) supplied for MQSeries for Windows NT is called ubbstxmn.mak, and is held in the MQSeries samples directory.

```

TUXDIR = f:\tuxedo
MQMDIR = g:\Program Files\MQSeries
APPDIR = f:\tuxedo\apps\mqapp
MQMLIB = $(MQMDIR)\tools\lib
MQMINC = $(MQMDIR)\tools\c\include
MQMSAMP = $(MQMDIR)\tools\c\samples
INC = -f "-I$(MQMINC) -I$(APPDIR)"
DBG = -f "/Zi"

amqstx.exe:
$(TUXDIR)\bin\mkfldhdr -d$(APPDIR) $(MQMSAMP)\amqstxvx.fld
$(TUXDIR)\bin\viewc -d$(APPDIR) $(MQMSAMP)\amqstxvx.v
$(TUXDIR)\bin\builtdtms -o MQXA -r MQSeries_XA_RMI
$(TUXDIR)\bin\buildserver -o MQSERV1 -f $(MQMSAMP)\amqstxsx.c \
-f $(MQMLIB)\mqm.lib -v $(INC) $(DBG) \
-r MQSeries_XA_RMI \
-s MPUT1:MPUT -s MGET1:MGET
$(TUXDIR)\bin\buildserver -o MQSERV2 -f $(MQMSAMP)\amqstxsx.c \
-f $(MQMLIB)\mqm.lib -v $(INC) $(DBG) \
-r MQSeries_XA_RMI \
-s MPUT2:MPUT -s MGET2:MGET
$(TUXDIR)\bin\buildclient -o doputs -f $(MQMSAMP)\amqstxpx.c \
-f $(MQMLIB)\mqm.lib -v $(INC) $(DBG)
$(TUXDIR)\bin\buildclient -o dogets -f $(MQMSAMP)\amqstxgx.c \
-f $(MQMLIB)\mqm.lib $(INC) -v $(DBG)
$(TUXDIR)\bin\tmloadcf -y $(APPDIR)\ubbstxcn.cfg

```

Figure 40. Sample TUXEDO makefile for MQSeries for Windows NT

To build the server environment for MQSeries for Compaq Tru64 UNIX

The procedure is the same on Compaq Tru64 UNIX Version 4.0 and Version 5.0. It is assumed that you have a working TUXEDO environment.

1. Create a directory (for example <appdir>) in which the server environment is built and execute all commands in this directory.
2. Export the following environment variables, where TUXDIR is the root directory for TUXEDO:

```

$ TUXDIR=/TUXDIR; export TUXDIR
$ PATH=$TUXDIR/bin:$PATH; export PATH
$ COBCPY=$TUXDIR/cobinclude; export COBCPY
$ COBOPT="-C ANS85 -C ALIGN=8 -C NOIBMCOMP -C TRUNC=ANSI \
-C OSEXT=cbl"; export COBOPT
$ LD_LIBRARY_PATH=$TUXDIR/lib:$LD_LIBRARY_PATH; export LD_LIBRARY_PATH

```

3. Add the following to the TUXEDO file TUXDIR/udataobj/RM

```

MQSeries_XA_RMI:MQRMIXASwitchDynamic:-lmqm -lmqmzse \
-lmqmxa -lmqmcs -L/$TUXDIR/lib -ltux

```

4. Run the commands:

```

$ mkfldhdr /opt/mqm/samp/amqstxvx.flds
$ viewc /opt/mqm/samp/amqstxvx.v
$ builtdtms -o MQXA -r MQSeries_XA_RMI
$ buildserver -o MQSERV1 -f /opt/mqm/samp/amqstxsx.c \
-f /opt/mqm/lib/libmqm.so -r MQSeries_XA_RMI \
-s MPUT1:MPUT -s MGET1:MGET -v -bsh
$ buildserver -o MQSERV2 -f /opt/mqm/samp/amqstxsx.c \
-f /opt/mqm/lib/libmqm.so -r MQSeries_XA_RMI \
-s MPUT2:MPUT -s MGET2:MGET -v -bsh
$ buildclient -o doputs -f /opt/mqm/samp/amqstxpx.c \
-f /opt/mqm/lib/libmqm.so -f /opt/mqm/lib/libmqmzse.so \
-f /opt/mqm/lib/libmqmcs.so

```

Building Tuxedo applications

```
|
|           $ buildclient -o dogets -f /opt/mqm/samp/amqstxgx.c \
|               -f /opt/mqm/lib/libmqm.so -f /opt/mqm/lib/libmqmzse.so \
|               -f /opt/mqm/lib/libmqmcs.so
|
| 5. Edit the file ubbstxcx.cfg and add details of the machine name, working
|     directories, and queue manager as necessary. Execute the following command:
|
|     $ tmloadcf -y ubbstxcx.cfg
|
| 6. Create the TLOGDEVICE:
|
|     $ tmadm -c
|
|
|     A > prompt then appears. At this point, enter:
|
|     > crdl -z /<appdir>/TLOG1
|
| 7. Start the queue manager MYQUEUEMANAGER
|
|     $ strmqm MYQUEUEMANAGER
|
| 8. Start the Tuxedo server:
|
|     $ tmboot -y
```

You can now use the doputs and dogets programs to put messages to a queue and retrieve them from a queue.

Server sample program for TUXEDO

This program is designed to run with the Put (amqstxpx.c) and the Get (amqstxgx.c) sample programs. The sample server program runs automatically when TUXEDO is started.

Note: You must start your queue manager *before* you start TUXEDO.

The sample server provides two TUXEDO services, MPUT1 and MGET1.

The MPUT1 service is driven by the PUT sample and uses MQPUT1 in syncpoint to put a message in a unit of work controlled by TUXEDO. It takes the parameters QName and Message Text, which are supplied by the PUT sample.

The MGET1 service opens and closes the queue each time it gets a message. It takes the parameters QName and Message Text, which are supplied by the GET sample.

Any error messages, reason codes, and status messages are written to the TUXEDO log file.

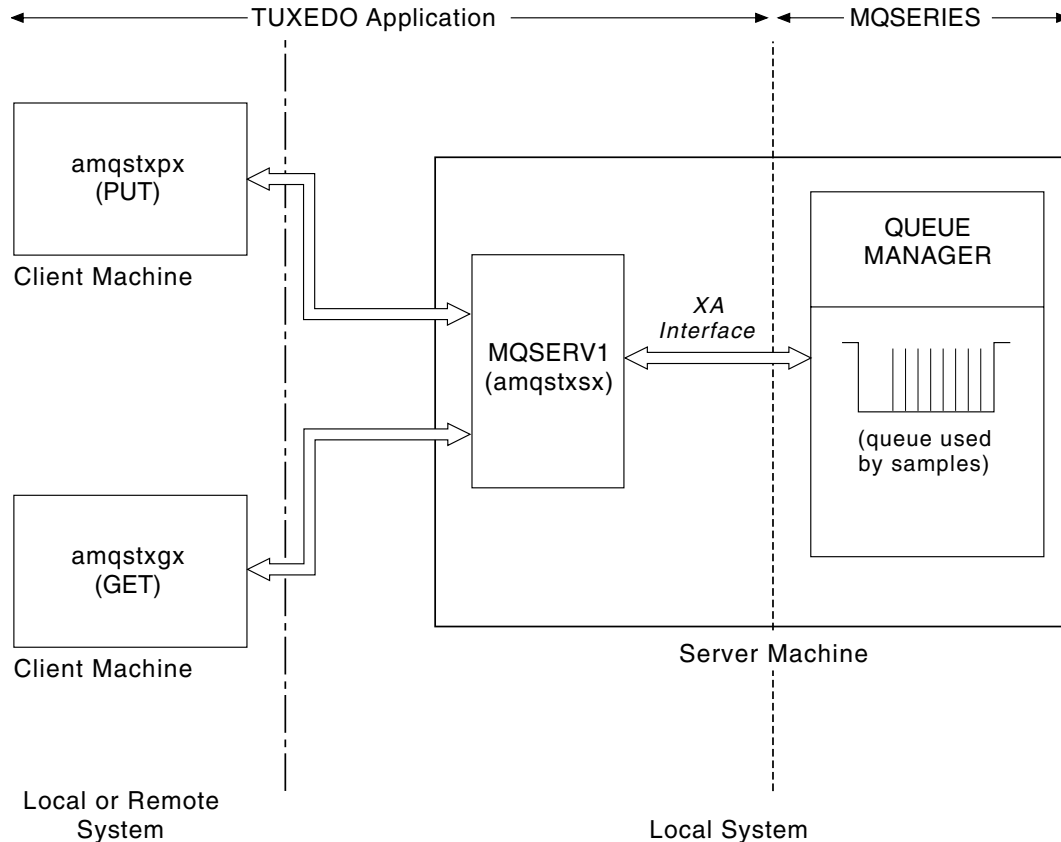


Figure 41. How TUXEDO samples work together

Put sample program for TUXEDO

This sample allows you to put a message on a queue multiple times, in batches, demonstrating syncpointing using TUXEDO as the resource manager. The sample server program `amqstxsx` must be running for the put sample to succeed - the server sample program makes the connection to the queue manager and uses the XA interface. To run the sample enter:

- `doputs -n queuename -b batchsize -c trancount -t message`

For example:

- `doputs -n myqueue -b 5 -c 6 -t "Hello World"`

This puts 30 messages on to the queue named `myqueue`, in 6 batches each with 5 messages in them. If there are any problems it will back a batch of messages out, otherwise it will commit them.

Any error messages are written to the TUXEDO log file and to `stderr`. Any reason codes are written to `stderr`.

Get sample for TUXEDO

This sample allows you to get messages from a queue in batches. The sample server program `amqstxsx` must be running for the put sample to succeed - the server sample program makes the connection to the queue manager and uses the XA interface. To run the sample enter:

- `dogets -n queuename -b batchsize -c trancount`

Building Tuxedo applications

For example:

- `dogets -n myqueue -b 6 -c 4`

This takes 24 messages off the queue named `myqueue`, in 6 batches each with 4 messages in them. If you ran this after the `put` example, which put 30 messages on `myqueue`, you would now have only 6 messages on `myqueue`. Note that the number of batches and the batch size can vary between the putting of messages and the getting of them.

Any error messages are written to the TUXEDO log file and to `stderr`. Any reason codes are written to `stderr`.

Encina sample program

This program puts 10 messages to the queue, backing out the odd numbered messages and committing the even numbered messages. The message is a 4-byte number.

The queue used by this sample is the `SYSTEM.DEFAULT.MODEL.QUEUE`, so a temporary dynamic queue is created each time the program is run. You will need to run `trace` to see what happens when the program runs.

Building the AMQSXAE0.C sample

When compiling for a UNIX or OS/2 platform, ensure that the symbolic constant, `WIN32` is not defined. This constant is used in the preprocessor statements for processing specific to Windows NT:

```
#if defined(WIN32)
```

Compiling and linking on Windows NT

When compiling, specify the following options (in addition to those usually specified for an MQSeries application) to the C compiler:

```
-MD -DWIN32 -DDEC_DCE -Gz
```

The sample contains references to the Encina header files:

```
#include <tc/tc.h>
#include <tmxa/tmxa_status.h>
#include <tmxa/tmxa.h>
```

At compile time, also include the parent directory path name containing these files, using the compiler `-I` option with a value which names the directory. For example:

```
-Ic:\opt\encina\include
```

At link time, the directory path names containing the Encina and DCE library files must also be specified to the linker, by setting the `LIB` environment variable. For example:

```
SET LIB=C:\OPT\ENCINA\LIB;C:\OPT\DCE\LIB;%LIB%
```

When linking, specify the following library files:

- `mqm.lib`
- `mqmenc.lib`
- `libEncServer.lib`
- `libEncina.lib`
- `msvcrt.lib`
- `pthread.lib`
- `libdce.lib`

Compiling and linking on Sun Solaris

Use the following invocation:

```
cc -I/opt/encina/include -c amqsxae0.c && cc -mt -o amqsxae0 amqsxae0.o \
-L/opt/encina/lib -L/opt/mqm/lib -lmqm -lmqmcs_d -lmqmzse -lmqmxa \
-lsocket -lnsl -ldce -lthread -lEncServer -lEncina -lc -lm
```

Dead-letter queue handler sample

A sample dead-letter queue handler is provided, the name of the executable version is amqsdlq. If you want a dead-letter queue handler that is different to RUNMQDLQ, the source of the sample is available for you to use your base.

The sample is similar to the dead-letter handler provided within the product but trace and error reporting are different. There are two environment variables available to you:

ODQ_TRACE

set to YES or yes to switch tracing on

ODQ_MSG

set to the name of the file containing error and information messages. The file provided is called amqsdlq.msg.

These need to be made known to your environment using either the **export** or **set** commands, depending on your platform; trace is turned off using the **unset** command.

You can modify the error message file, amqsdlq.msg, to suit your own requirements. The sample puts messages out to stdout, *not* to the MQSeries error log file.

The *System Management Guide* for your platform explains how the dead-letter handler works, and how you run it.

The Connect sample program

The Connect sample program allows you to explore the MQCONNX call and its options from a client. The sample connects to the queue manager using the MQCONNX call, inquires about the name of the queue manager using the MQINQ call, and displays it.

Note: The Connect sample program is a client sample. You can compile and run it on a server but the function is meaningful only on a client, and only client executables are supplied.

Running the amqscnxc sample

The command-line syntax of the Connect sample program is:

```
amqscnxc [-x ConnName [-c SvrconnChannelName]] [QMGrName]
```

The parameters are optional and their order is not important with the exception that QMGrName, if it is specified, must come last. The parameters are:

ConnName

The TCP/IP connection name of the server queue manager

SvrconnChannelName

The name of the server connection channel

Connect sample program

QMgrName

The name of the target queue manager

If you do not specify the TCP/IP connection name, MQCONN is issued with the *ClientConnPtr* set to NULL. If you specify the TCP/IP connection name but not the server connection channel (the reverse is not allowed) the sample uses the name SYSTEM.DEF.SVRCONN. If you do not specify the target queue manager the sample connects to whichever queue manager is listening at the given TCP/IP connection name.

Note: If you enter a question mark as the only parameter or if you enter incorrect parameters you will see a message explaining how to use the program.

If you run the sample with no command-line options the contents of the MQSERVER environment variable are used to determine the connection information. (In this example MQSERVER is set to "SYSTEM.DEF.SVRCONN/TCP/machine.site.company.com".) You see output like this:

```
Sample AMQSCNXC start
Connecting to the default queue manager
with no client connection information specified.
Connection established to queue manager machine
```

```
Sample AMQSCNXC end
```

If you run the sample and provide a TCP/IP connection name and a server connection channel name but no target queue manager name, like this:

```
amqscnxc -x machine.site.company.com -c SYSTEM.ADMIN.SVRCONN
```

the default queue manager name is used and you see output like this:

```
Sample AMQSCNXC start
Connecting to the default queue manager
using the server connection channel SYSTEM.ADMIN.SVRCONN
on connection name machine.site.company.com.
Connection established to queue manager MACHINE
```

```
Sample AMQSCNXC end
```

If you run the sample and provide a TCP/IP connection name and a target queue manager name, like this:

```
amqscnxc -x machine.site.company.com MACHINE
```

you see output like this:

```
Sample AMQSCNXC start
Connecting to queue manager MACHINE
using the server connection channel SYSTEM.DEF.SVRCONN
on connection name machine.site.company.com.
Connection established to queue manager MACHINE
```

```
Sample AMQSCNXC end
```

Chapter 35. Sample programs for MQSeries for OS/390

This chapter describes the sample applications that are delivered with MQSeries for OS/390. These samples demonstrate typical uses of the Message Queue Interface (MQI).

MQSeries for OS/390 also provides a sample API-crossing exit program, described in “The API-crossing exit for OS/390” on page 221, and sample data-conversion exits, described in “Chapter 11. Writing data-conversion exits” on page 155.

The sample applications are supplied in source form only. The source modules include pseudocode that describes the program logic.

Note: Although some of the sample applications have basic panel-driven interfaces, they do not aim to demonstrate how to design the “look and feel” of your applications. For more information on how to design panel-driven interfaces for nonprogrammable terminals, see the SAA[®] *Common User Access[®]: Basic Interface Design Guide* (SC26-4583) and its addendum (GG22-9508). These provide guidelines to help you design applications that are consistent both within the application and across other applications.

This chapter introduces the sample programs, under these headings:

- “Features demonstrated in the sample applications”
- “Preparing and running sample applications for the batch environment” on page 425
- “Preparing sample applications for the TSO environment” on page 427
- “Preparing the sample applications for the CICS environment” on page 428
- “Preparing the sample application for the IMS environment” on page 431
- “The Put samples” on page 432
- “The Get samples” on page 435
- “The Browse sample” on page 437
- “The Print Message sample” on page 439
- “The Queue Attributes sample” on page 443
- “The Mail Manager sample” on page 444
- “The Credit Check sample” on page 451
- “The Message Handler sample” on page 464

Features demonstrated in the sample applications

This section summarizes the MQI features demonstrated in each of the sample applications, shows the programming languages in which each sample is written, and the environment in which each sample runs.

Put samples

The Put samples demonstrate how to put messages on a queue using the MQPUT call.

The application uses these MQI calls:

- MQCONN
- MQOPEN
- MQPUT

Features demonstrated

- MQCLOSE
- MQDISC

The program is delivered in COBOL and C, and runs in the batch and CICS environment. See Table 38 on page 426 for the batch application and Table 43 on page 429 for the CICS application.

Get samples

The Get samples demonstrate how to get messages from a queue using the MQGET call.

The application uses these MQI calls:

- MQCONN
- MQOPEN
- MQGET
- MQCLOSE
- MQDISC

The program is delivered in COBOL and C, and runs in the batch and CICS environment. See Table 38 on page 426 for the batch application and Table 43 on page 429 for the CICS application.

Browse sample

The Browse sample demonstrates how to browse a message, print it, then step through the messages on a queue.

The application uses these MQI calls:

- MQCONN
- MQOPEN
- MQGET for browsing messages
- MQCLOSE
- MQDISC

The program is delivered in the COBOL, assembler, PL/I, and C languages. The application runs in the batch environment. See Table 39 on page 426 for the batch application.

Print Message sample

The Print Message sample demonstrates how to remove a message from a queue and print the data in the message, together with all the fields of its message descriptor. By removing comment characters from two lines in the source module, you can change the program so that it browses, rather than removes, the messages on a queue. This program can usefully be used for diagnosing problems with an application that is putting messages on a queue.

The application uses these MQI calls:

- MQCONN
- MQOPEN
- MQGET for removing messages from a queue (with an option to browse)
- MQCLOSE
- MQDISC

The program is delivered in the C language. The application runs in the batch environment. See Table 40 on page 426 for the batch application.

Queue Attributes sample

The Queue Attributes sample demonstrates how to inquire about and set the values of MQSeries for OS/390 object attributes.

The application uses these MQI calls:

- MQOPEN
- MQINQ
- MQSET
- MQCLOSE

The program is delivered in the COBOL, assembler, and C languages. The application runs in the CICS environment. See Table 44 on page 430 for the CICS application.

Mail Manager sample

The Mail Manager sample demonstrates these techniques:

- Using alias queues
- Using a model queue to create a temporary dynamic queue
- Using reply-to queues
- Using syncpoints in the CICS and batch environments
- Sending commands to the system-command input queue
- Testing return codes
- Sending messages to remote queue managers, both by using a local definition of a remote queue and by putting messages directly on a named queue at a remote queue manager

The application uses these MQI calls:

- MQCONN
- MQOPEN
- MQPUT1
- MQGET
- MQINQ
- MQCMIT
- MQCLOSE
- MQDISC

Three versions of the application are provided:

- A CICS application written in COBOL
- A TSO application written in COBOL
- A TSO application written in C

The TSO applications use the MQSeries for OS/390 batch adapter and include some ISPF panels.

See Table 41 on page 427 for the TSO application, and Table 45 on page 430 for the CICS application.

Credit Check sample

The Credit Check sample is a suite of programs that demonstrates these techniques:

- Developing an application that runs in more than one environment
- Using a model queue to create a temporary dynamic queue
- Using a correlation identifier
- The setting and passing of context information

Features demonstrated

- Using message priority and persistence
- Starting programs by using triggering
- Using reply-to queues
- Using alias queues
- Using a dead-letter queue
- Using a namelist
- Testing return codes

The application uses these MQI calls:

- MQOPEN
- MQPUT
- MQPUT1
- MQGET for browsing and getting messages, using the wait and signal options, and for getting a specific message
- MQINQ
- MQSET
- MQCLOSE

The sample can run as a stand-alone CICS application. However, to demonstrate how to design a message queuing application that uses the facilities provided by both the CICS and IMS environments, one module is also supplied as an IMS batch message processing program.

The CICS programs are delivered in C and COBOL. The single IMS program is delivered in C.

See Table 46 on page 430 for the CICS application, and Table 47 on page 432 for the IMS application.

The Message Handler sample

The Message Handler sample allows you to browse, forward, and delete messages on a queue.

The application uses these MQI calls:

- MQCONN
- MQOPEN
- MQINQ
- MQPUT1
- MQCMIT
- MQBACK
- MQGET
- MQCLOSE
- MQDISC

The program is delivered in C and COBOL programming languages. The application runs under TSO. See Table 42 on page 428 for the TSO application.

Distributed queuing exit samples

The names of the source programs of the distributed queuing exit samples are listed in the following table:

Table 36. Source for the distributed queuing exit samples

Member name	For language	Description	Supplied in library
CSQ4BAX0	Assembler	Source program	SCSQASMS
CSQ4BCX1	C	Source program	SCSQC37S
CSQ4BCX2	C	Source program	SCSQC37S

Note: The source programs are link-edited with CSQXSTUB.

See the *MQSeries Intercommunication* book for a description of the distributed queuing exit samples.

Data-conversion exit samples

A skeleton is provided for a data-conversion exit routine, and a sample is shipped with MQSeries illustrating the MQXCNCV call. The names of the source programs of the data-conversion exit samples are listed in the following table:

Table 37. Source for the data conversion exit samples (Assembler language only)

Member name	Description	Supplied in library
CSQ4BAX8	Source program	SCSQASMS
CSQ4BAX9	Source program	SCSQASMS
CSQ4CAX9	Source program	SCSQASMS

Note: The source programs are link-edited with CSQASTUB.

See “Chapter 11. Writing data-conversion exits” on page 155 for more information.

Preparing and running sample applications for the batch environment

To prepare a sample application that runs in the batch environment, perform the same steps that you would when building any batch MQSeries for OS/390 application. These steps are listed in “Building OS/390 batch applications” on page 312.

Note: The assembler language version of the Browse sample uses data control blocks (DCBs), so you must link-edit it using RMODE(24).

The library members that you will use are listed in Table 38, Table 39, and Table 40 on page 426.

You must edit the run JCL supplied for the samples that you want to use (see Table 38, Table 39, and Table 40 on page 426).

The PARM statement in the supplied JCL contains a number of parameters that you need to modify. To run the C sample programs, separate the parameters by spaces; to run the Assembler, COBOL, and PL/I sample programs, separate them by commas. For example, if the name of your queue manager is CSQ1 and you

Preparing batch samples

want to run the application with a queue named LOCALQ1, in the COBOL, PL/I, and assembler-language JCL, your PARM statement should look like this:

```
PARM=(CSQ1,LOCALQ1)
```

In the C language JCL, your PARM statement should look like this:

```
PARM=('CSQ1 LOCALQ1')
```

You are now ready to submit the jobs.

Names of the sample batch applications

The names of the source programs and JCL that are supplied for each of the sample batch applications are listed in the following tables:

Put and Get samples Table 38

Browse sample Table 39

Print message sample Table 40

Table 38. Source and JCL for the Put and Get samples

Member name	For language	Description	Supplied in library
CSQ4BCJ1	C	Get source program	SCSQC37S
CSQ4BCK1	C	Put source program	SCSQC37S
CSQ4BVJ1	COBOL	Get source program	SCSQCOBS
CSQ4BVK1	COBOL	Put source program	SCSQCOBS
CSQ4BCJR	C	Sample run JCL	SCSQPROC
CSQ4BVJR	COBOL	Sample run JCL	SCSQPROC

Table 39. Source and JCL for the Browse sample

Member name	For language	Description	Supplied in library
CSQ4BVA1	COBOL	Source program	SCSQCOBS
CSQ4BVAR	COBOL	Sample run JCL	SCSQPROC
CSQ4BAA1	Assembler	Source program	SCSQASMS
CSQ4BAAR	Assembler	Sample run JCL	SCSQPROC
CSQ4BCA1	C	Source program	SCSQC37S
CSQ4BCAR	C	Sample run JCL	SCSQPROC
CSQ4BPA1	PL/I	Source program	SCSQPLIS
CSQ4BPAR	PL/I	Sample run JCL	SCSQPROC

Table 40. Source for the Print Message sample (C language only)

Member name	Description	Supplied in library
CSQ4BCG1	Source program	SCSQC37S
CSQ4BCGR	Sample run JCL	SCSQPROC

Preparing sample applications for the TSO environment

To prepare a sample application that runs in the TSO environment, perform the same steps that you would when building any batch MQSeries for OS/390 application—these steps are listed in “Building OS/390 batch applications” on page 312. The library members you will use are listed in Table 41.

For the Mail Manager sample application, ensure that the queues it uses are available on your system. They are defined in the member **thlqual.SCSQPROC(CSQ4CVD)**. To ensure that these queues are always available, you could add these members to your CSQINP2 initialization input data set, or use the CSQUTIL program to load these queue definitions.

Names of the sample TSO applications

The names of the source programs that are supplied for each of the sample TSO applications are listed in the following tables:

Mail manager sample Table 41
 Message handler sample Table 42 on page 428

These samples use ISPF panels. You must therefore include the ISPF stub, ISPLINK, when you link-edit the programs.

Table 41. Source and JCL for the Mail Manager (TSO) sample

Member name	For language	Description	Supplied in library
CSQ4CVD	independent	MQSeries for OS/390 object definitions	SCSQPROC
CSQ40	independent	ISPF messages	SCSQMSGE
CSQ4RVD1	COBOL	CLIST to initiate CSQ4TVD1	SCSQCLST
CSQ4TVD1	COBOL	Source program for Menu program	SCSQCOBS
CSQ4TVD2	COBOL	Source program for Get Mail program	SCSQCOBS
CSQ4TVD4	COBOL	Source program for Send Mail program	SCSQCOBS
CSQ4TVD5	COBOL	Source program for Nickname program	SCSQCOBS
CSQ4VDP1-6	COBOL	Panel definitions	SCSQPNLA
CSQ4VD0	COBOL	Data definition	SCSQCOBC
CSQ4VD1	COBOL	Data definition	SCSQCOBC
CSQ4VD2	COBOL	Data definition	SCSQCOBC
CSQ4VD4	COBOL	Data definition	SCSQCOBC
CSQ4RCD1	C	CLIST to initiate CSQ4TCD1	SCSQCLST
CSQ4TCD1	C	Source program for Menu program	SCSQ37S
CSQ4TCD2	C	Source program for Get Mail program	SCSQ37S

Preparing TSO samples

Table 41. Source and JCL for the Mail Manager (TSO) sample (continued)

Member name	For language	Description	Supplied in library
CSQ4TCD4	C	Source program for Send Mail program	SCSQC37S
CSQ4TCD5	C	Source program for Nickname program	SCSQC37S
CSQ4CDP1-6	C	Panel definitions	SCSQPNLA
CSQ4TC0	C	Include file	SCSQC370

Table 42. Source for the Message Handler sample

Member name	For language	Description	Supplied in library
CSQ4TCH0	C	Data definition	SCSQC370
CSQ4TCH1	C	Source program	SCSQC37S
CSQ4TCH2	C	Source program	SCSQC37S
CSQ4TCH3	C	Source program	SCSQC37S
CSQ4RCH1	C and COBOL	CLIST to initiate CSQ4TCH1 or CSQ4TVH1	SCSQCLST
CSQ4CHP1	C and COBOL	Panel definition	SCSQPNLA
CSQ4CHP2	C and COBOL	Panel definition	SCSQPNLA
CSQ4CHP3	C and COBOL	Panel definition	SCSQPNLA
CSQ4CHP9	C and COBOL	Panel definition	SCSQPNLA
CSQ4TVH0	COBOL	Data definition	SCSQCOBC
CSQ4TVH1	COBOL	Source program	SCSQCOBS
CSQ4TVH2	COBOL	Source program	SCSQCOBS
CSQ4TVH3	COBOL	Source program	SCSQCOBS

Preparing the sample applications for the CICS environment

Before you run the CICS sample programs, you must log on to CICS using a LOGMODE of 32702. This is because the sample programs have been written to use a 3270 mode 2 screen.

To prepare a sample application that runs in the CICS environment, perform the following steps:

1. Create the symbolic description map and the physical screen map for the sample by assembling the BMS screen definition source (supplied in library **thlqual.SCSQMAPS**, where **thlqual** is the high-level qualifier used by your installation). When you name the maps, use the name of the BMS screen definition source (not available for Put and Get sample programs), but omit the last character of that name.
2. Perform the same steps that you would when building any CICS MQSeries for OS/390 application—these steps are listed in “Building CICS applications” on page 313. The library members that you will use are listed in Table 43 on page 429, Table 44 on page 430, Table 45 on page 430, and Table 46 on page 430.

Preparing CICS samples

3. Identify the map set, programs, and transaction to CICS by updating the CICS system definition (CSD) data set. The definitions you require are in the member **thlqual.SCSQPROC(CSQ4S100)**. For guidance on how to do this, see the *MQSeries for OS/390 System Setup Guide*.

Note: For the Credit Check sample application, you will get an error message at this stage if you have not already created the VSAM data set that the sample uses.

4. For the Credit Check and Mail Manager sample applications, ensure that the queues they use are available on your system. For the Credit Check sample, they are defined in the member **thlqual.SCSQPROC(CSQ4CVB)** for COBOL, and **thlqual.SCSQPROC(CSQ4CCB)** for C. For the Mail Manager sample, they are defined in the member **thlqual.SCSQPROC(CSQ4CVD)**. To ensure that these queues are always available, you could add these members to your CSQINP2 initialization input data set, or use the CSQUTIL program to load these queue definitions.

For the Queue Attributes sample application, you could use one or more of the queues that are supplied for the other sample applications. Alternatively, you could use your own queues. However, note that in the form that it is supplied, this sample works only with queues that have the characters CSQ4SAMP in the first eight bytes of their name.

QLOPabend

When the CICS sample applications supplied with MQSeries for OS/390 use MQI calls, they do not test for the return codes that indicate that the queue manager is not available. If the queue manager is not available when you attempt to run one of the CICS samples, the sample abends with the CICS abend code QLOP. If this happens, you must connect your queue manager to your CICS system before you attempt to start the sample application again. For information about starting a connection, see the *MQSeries for OS/390 System Administration Guide*.

Names of the sample CICS applications

The source and JCL files that are supplied for each of the sample CICS applications are listed in the following tables:

Put and Get samples	Table 43
Queue attributes sample	Table 44 on page 430
Mail Manager (CICS) sample	Table 45 on page 430
Credit Check (CICS) sample	Table 46 on page 430

Table 43. Source and JCL for the Put and Get samples

Member name	For language	Description	Supplied in library
CSQ4CCK1	C	Source program	SCSQ37S
CSQ4CCJ1	C	Source program	SCSQ37S
CSQ4CVJ1	COBOL	Source program	SCSQCOBS
CSQ4CVK1	COBOL	Source program	SCSQCOBS
CSQ4S100	independent	CICS system definition data set	SCSQPROC

Preparing CICS samples

Table 44. Source for the Queue Attributes sample

Member name	For language	Description	Supplied in library
CSQ4CVC1	COBOL	Source program	SCSQCOBS
CSQ4VMSG	COBOL	Message definition	SCSQCOBC
CSQ4VCMS	COBOL	BMS screen definition	SCSQMAPS
CSQ4CAC1	Assembler	Source program	SCSQASMS
CSQ4AMSG	Assembler	Message definition	SCSQMACS
CSQ4ACMS	Assembler	BMS screen definition	SCSQMAPS
CSQ4CCC1	C	Source program	SCSQ37S
CSQ4CMMSG	C	Message definition	SCSQ370
CSQ4CCMS	C	BMS screen definition	SCSQMAPS
CSQ4S100	independent	CICS system definition data set	SCSQPROC

Table 45. Source and JCL for the Mail Manager (CICS) sample (COBOL only)

Member name	Description	Supplied in library
CSQ4CVD	MQSeries for OS/390 object definitions	SCSQPROC
CSQ4CVD1	Source for Menu program	SCSQCOBS
CSQ4CVD2	Source for Get Mail program	SCSQCOBS
CSQ4CVD3	Source for Display Message program	SCSQCOBS
CSQ4CVD4	Source for Send Mail program	SCSQCOBS
CSQ4CVD5	Source for Nickname program	SCSQCOBS
CSQ4VDMS	BMS screen definition source	SCSQMAPS
CSQ4S100	CICS system definition data set	SCSQPROC
CSQ4VD0	Data definition	SCSQCOBC
CSQ4VD3	Data definition	SCSQCOBC
CSQ4VD4	Data definition	SCSQCOBC

Table 46. Source and JCL for the Credit Check CICS sample

Member name	For language	Description	Supplied in library
CSQ4CVB	independent	MQSeries object definitions	SCSQPROC
CSQ4CCB	independent	MQSeries object definitions	SCSQPROC
CSQ4CVB1	COBOL	Source for user-interface program	SCSQCOBS
CSQ4CVB2	COBOL	Source for credit application manager	SCSQCOBS
CSQ4CVB3	COBOL	Source for checking-account program	SCSQCOBS
CSQ4CVB4	COBOL	Source for distribution program	SCSQCOBS
CSQ4CVB5	COBOL	Source for agency-query program	SCSQCOBS

Table 46. Source and JCL for the Credit Check CICS sample (continued)

Member name	For language	Description	Supplied in library
CSQ4CCB1	C	Source for user-interface program	SCSQ37S
CSQ4CCB2	C	Source for credit application manager	SCSQ37S
CSQ4CCB3	C	Source for checking-account program	SCSQ37S
CSQ4CCB4	C	Source for distribution program	SCSQ37S
CSQ4CCB5	C	Source for agency-query program	SCSQ37S
CSQ4CB0	C	Include file	SCSQ370
CSQ4CBMS	C	BMS screen definition source	SCSQMAPS
CSQ4VBMS	COBOL	BMS screen definition source	SCSQMAPS
CSQ4VB0	COBOL	Data definition	SCSQCOBC
CSQ4VB1	COBOL	Data definition	SCSQCOBC
CSQ4VB2	COBOL	Data definition	SCSQCOBC
CSQ4VB3	COBOL	Data definition	SCSQCOBC
CSQ4VB4	COBOL	Data definition	SCSQCOBC
CSQ4VB5	COBOL	Data definition	SCSQCOBC
CSQ4VB6	COBOL	Data definition	SCSQCOBC
CSQ4VB7	COBOL	Data definition	SCSQCOBC
CSQ4VB8	COBOL	Data definition	SCSQCOBC
CSQ4BAQ	independent	Source for VSAM data set	SCSQPROC
CSQ4FILE	independent	JCL to build VSAM data set used by CSQ4CVB3	SCSQPROC
CSQ4S100	independent	CICS system definition data set	SCSQPROC

Preparing the sample application for the IMS environment

Part of the Credit Check sample application can run in the IMS environment. To prepare this part of the application to run with the CICS sample you must first perform the steps described in “Preparing the sample applications for the CICS environment” on page 428.

Then perform the following steps:

1. Perform the same steps that you would when building any IMS MQSeries for OS/390 application—these steps are listed in “Building IMS (BMP or MPP) applications” on page 314. The library members that you will use are listed in Table 47 on page 432.
2. Identify the application program and database to IMS. Samples are provided with PSBGEN, DBDGEN, ACB definition, MSGEN, and IMSDALOC statements to enable this.
3. Load the database CSQ4CA by tailoring and running the sample JCL provided for this purpose (CSQ4ILDB). This JCL loads the database with data from the file CSQ4BAQ. Update the IMS control region with a DD statement for the database CSQ4CA.

Preparing IMS sample

4. Start the checking-account program as a batch message processing (BMP) program by tailoring and running the sample JCL provided for this purpose. This JCL starts a batch-oriented BMP program. To run the program as a message-oriented BMP program, remove the comment characters from the line in the JCL that contains the IN= statement.

Names of the sample IMS application

The source and JCL that are supplied for the Credit Check sample IMS application are listed in Table 47.

Table 47. Source and JCL for the Credit Check IMS sample (C only)

Member name	Description	Supplied in library
CSQ4CVB	MQSeries object definitions	SCSQPROC
CSQ4ICB3	Source for checking-account program	SCSQC37S
CSQ4ICBL	Source for loading the checking-account database	SCSQC37S
CSQ4CBI	Data definition	SCSQC370
CSQ4PSBL	PSBGEN JCL for database-load program	SCSQPROC
CSQ4PSB3	PSBGEN JCL for checking-account program	SCSQPROC
CSQ4DBDS	DBDGEN JCL for database CSQ4CA	SCSQPROC
CSQ4GIMS	IMSGEN macro definitions for CSQ4IVB3 and CSQ4CA	SCSQPROC
CSQ4ACBG	Application control block (ACB) definition for CSQ4IVB3	SCSQPROC
CSQ4BAQ	Source for database	SCSQPROC
CSQ4ILDB	Sample run JCL for database-load job	SCSQPROC
CSQ4ICBR	Sample run JCL for checking-account program	SCSQPROC
CSQ4DYNA	IMSDALOC macro definitions for database	SCSQPROC

The Put samples

The Put sample programs put messages on a queue using the MQPUT call.

The source programs are supplied in C and COBOL in the batch and CICS environments (see Table 38 on page 426 and Table 43 on page 429).

Design of the Put sample

The flow through the program logic is:

1. Connect to the queue manager using the MQCONN call. If this call fails, print the completion and reason codes and stop processing.

Note: If you are running the sample in a CICS environment, you do not need to issue an MQCONN call; if you do, it returns DEF_HCONN. You can use the connection handle MQHC_DEF_HCONN for the MQI calls that follow.

2. Open the queue using the MQOPEN call with the MQOO_OUTPUT option. On input to this call, the program uses the connection handle that is returned in step 1. For the object descriptor structure (MQOD), it uses the default values for all fields except the queue name field which is passed as a parameter to the program. If the MQOPEN call fails, print the completion and reason codes and stop processing.
3. Create a loop within the program issuing MQPUT calls until the required number of messages are put on the queue. If an MQPUT call fails, the loop is abandoned early, no further MQPUT calls are attempted, and the completion and reason codes are returned.
4. Close the queue using the MQCLOSE call with the object handle returned in step 2. If this call fails, print the completion and reason codes.
5. Disconnect from the queue manager using the MQDISC call with the connection handle returned in step 1. If this call fails, print the completion and reason codes.

Note: If you are running the sample in a CICS environment, you do not need to issue an MQDISC call.

The Put samples for the batch environment

To run the samples, you must edit and run the sample JCL, as described in “Preparing and running sample applications for the batch environment” on page 425.

The programs take the following parameters in an EXEC PARM, separated by spaces in C and commas in COBOL:

1. The name of the queue manager (4 characters)
2. The name of the target queue (48 characters)
3. The number of messages (up to 4 digits)
4. The padding character to be written in the message (1 character)
5. The number of characters to write in the message (up to 4 digits)
6. The persistence of the message (1 character: 'P' for persistent or 'N' for nonpersistent)

If you enter any of the above parameters wrongly, you will receive appropriate error messages.

Any messages from the samples are written to the SYSPRINT data set.

Usage notes

- To keep the samples simple, there are some minor functional differences between language versions. However, these differences are minimized if the layout of the parameters shown in the sample run JCL, CSQ4BCJR, and CSQ4BVJR, is used. None of the differences relate to the MQI.
- CSQ4BCK1 allows you to enter more than four digits for the number of messages sent and the length of the messages.
- For the two numeric fields, enter any digit between 1 and 9999. The value you enter should be a positive number. For example, to put a single message, you can enter 1 or 01 or 001 or 0001 as the value. If you enter non-numeric or

Put samples

negative values, you may receive an error. For example, if you enter '-1', the COBOL program will send a one-byte message, but the C program will receive an error.

- For both programs, CSQ4BCK1 and CSQ4BVK1, you must enter 'P' in the persistence parameter, ++PER++, if you require the message to be persistent. If you fail to do so, the message will be nonpersistent.

The Put samples for the CICS environment

The transactions take the following parameters separated by commas:

1. The number of messages (up to 4 digits)
2. The padding character to be written in the message (1 character)
3. The number of characters to write in the message (up to 4 digits)
4. The persistence of the message (1 character: 'P' for persistent or 'N' for nonpersistent)
5. The name of the target queue (48 characters)

If you enter any of the above parameters wrongly, you will receive appropriate error messages.

For the COBOL sample, invoke the Put sample in the CICS environment by entering:

```
MVPT,9999,*,9999,P,QUEUE.NAME
```

For the C sample, invoke the Put sample in the CICS environment by entering:

```
MCPT,9999,*,9999,P,QUEUE.NAME
```

Any messages from the samples are displayed on the screen.

Usage notes

- To keep the samples simple, there are some minor functional differences between language versions. None of the differences relate to the MQI.
- If you enter a queue name that is longer than 48 characters, its length is truncated to the maximum of 48 characters but no error message is returned.
- Before entering the transaction, press the 'CLEAR' key.
- For the two numeric fields, enter any number between 1 and 9999. The value you enter should be a positive number. For example, to put a single message, you can enter the value 1 or 01 or 001 or 0001. If you enter non-numeric or negative values, you may receive an error. For example, if you enter '-1', the COBOL program will send a 1 byte message, and the C program will abend with an error from malloc().
- For both programs, CSQ4CCK1 and CSQ4CVK1, you must enter 'P' in the persistence parameter, if you require the message to be persistent. For non-persistent messages, enter 'N' in the persistence parameter. If you enter any other value you will receive an error message.
- The messages are put in syncpoint because default values are used for all parameters except those set during program invocation.

The Get samples

The Get sample programs get messages from a queue using the MQGET call.

The source programs are supplied in C and COBOL in the batch and CICS environments (see Table 38 on page 426 and Table 43 on page 429).

Design of the Get sample

The flow through the program logic is:

1. Connect to the queue manager using the MQCONN call. If this call fails, print the completion and reason codes and stop processing.

Note: If you are running the sample in a CICS environment, you do not need to issue an MQCONN call; if you do, it returns DEF_HCONN. You can use the connection handle MQHC_DEF_HCONN for the MQI calls that follow.

2. Open the queue using the MQOPEN call with the MQOO_INPUT_SHARED and MQOO_BROWSE options. On input to this call, the program uses the connection handle that is returned in step 1. For the object descriptor structure (MQOD), it uses the default values for all fields except the queue name field which is passed as a parameter to the program. If the MQOPEN call fails, print the completion and reason codes and stop processing.
3. Create a loop within the program issuing MQGET calls until the required number of messages are retrieved from the queue. If an MQGET call fails, the loop is abandoned early, no further MQGET calls are attempted, and the completion and reason codes are returned. The following options are specified on the MQGET call:
 - MQGMO_NO_WAIT
 - MQGMO_ACCEPT_TRUNCATED_MESSAGE
 - MQGMO_SYNCPOINT or MQGMO_NO_SYNCPOINT
 - MQGMO_BROWSE_FIRST and MQGMO_BROWSE_NEXT

For a description of these options, see the *MQSeries Application Programming Reference* manual. For each message, the message number is printed followed by the length of the message and the message data.

4. Close the queue using the MQCLOSE call with the object handle returned in step 2. If this call fails, print the completion and reason codes.
5. Disconnect from the queue manager using the MQDISC call with the connection handle returned in step 1. If this call fails, print the completion and reason codes.

Note: If you are running the sample in a CICS environment, you do not need to issue an MQDISC call.

The Get samples for the batch environment

To run the samples, you must edit and run the sample JCL, as described in “Preparing and running sample applications for the batch environment” on page 425.

The programs take the following parameters in an EXEC PARM, separated by spaces in C and commas in COBOL:

1. The name of the queue manager (4 characters)
2. The name of the target queue (48 characters)
3. The number of messages to get (up to 4 digits)

Get samples

4. The browse/get message option (1 character: 'B' to browse or 'D' to destructively get the messages)
5. The syncpoint control (1 character: 'S' for syncpoint or 'N' for no syncpoint)

If you enter any of the above parameters wrongly, you will receive appropriate error messages.

Output from the samples is written to the SYSPRINT data set:

```
=====
PARAMETERS PASSED :
  QMGR      - VC9
  QNAME     - A.Q
  NUMMSGS   - 000000002
  GET       - D
  SYNCPOINT - N
=====
MQCONN SUCCESSFUL
MQOPEN SUCCESSFUL
000000000 : 000000010 : *****
000000001 : 000000010 : *****
000000002 MESSAGES GOT FROM QUEUE
MQCLOSE SUCCESSFUL
MQDISC SUCCESSFUL
```

Usage notes

- To keep the samples simple, there are some minor functional differences between language versions. However, these differences are minimized if the layout of the parameters shown in the sample run JCL, CSQ4BCJR, and CSQ4BVJR, are used. None of the differences relate to the MQI.
- CSQ4BCJ1 allows you to enter more than four digits for the number of messages retrieved.
- Messages longer than 64 KB are truncated.
- CSQ4BCJ1 can only correctly display character messages as it only displays until the first NULL (\0) character is displayed.
- For the numeric number-of-messages field, enter any digit between 1 and 9999. The value you enter should be a positive number. For example, to get a single message, you can enter 1 or 01 or 001 or 0001 as the value. If you enter non-numeric or negative values, you may receive an error. For example, if you enter '-1', the COBOL program will retrieve one message, but the C program will not retrieve any messages.
- For both programs, CSQ4BCJ1 and CSQ4BVJ1, you must enter 'B' in the get parameter, ++GET++, if you want to browse the messages.
- For both programs, CSQ4BCJ1 and CSQ4BVJ1, you must enter 'S' in the syncpoint parameter, ++SYNC++, for messages to be retrieved in syncpoint.

The Get samples for the CICS environment

The transactions take the following parameters in an EXEC PARM, separated by commas:

1. The number of messages to get (up to 4 digits)
2. The browse/get message option (1 character: 'B' to browse or 'D' to destructively get the messages)
3. The syncpoint control (1 character: 'S' for syncpoint or 'N' for no syncpoint)
4. The name of the target queue (48 characters)

If you enter any of the above parameters wrongly, you will receive appropriate error messages.

For the COBOL sample, invoke the Get sample in the CICS environment by entering:

```
MVGT,9999,B,S,QUEUE.NAME
```

For the C sample, invoke the Get sample in the CICS environment by entering:

```
MCGT,9999,B,S,QUEUE.NAME
```

When the messages are retrieved from the queue, they are put on a CICS temporary storage queue with the same name as the CICS transaction (for example, MCGT for the C sample).

Here is example output of the Get samples:

```
***** TOP OF QUEUE *****
000000000 : 000000010 : *****
000000001 : 000000010 : *****
***** BOTTOM OF QUEUE *****
```

Usage notes

- To keep the samples simple, there are some minor functional differences between language versions. None of the differences relate to the MQI.
- If you enter a queue name that is longer than 48 characters, its length is truncated to the maximum of 48 characters but no error message is returned.
- Before entering the transaction, press the 'CLEAR' key.
- CSQ4CCJ1 can only correctly display character messages as it only displays until the first NULL (\0) character is displayed.
- For the numeric field, enter any number between 1 and 9999. The value you enter should be a positive number. For example, to get a single message, you can enter the value 1 or 01 or 001 or 0001. If you enter a non-numeric or negative value, you may receive an error.
- Messages longer than 24 526 bytes in C and 9 950 bytes in COBOL are truncated. This is due to the way the CICS temporary storage queues are used.
- For both programs, CSQ4CCK1 and CSQ4CVK1, you must enter 'B' in the get parameter if you want to browse the messages, otherwise enter 'D'. This will perform destructive MQGET calls. If you enter any other value you will receive an error message.
- For both programs, CSQ4CCJ1 and CSQ4CVJ1, you must enter 'S' in the syncpoint parameter for messages to be retrieved in syncpoint. If you enter 'N' in the syncpoint parameter the MQGET calls will be issued out of syncpoint. If you enter any other value you will receive an error message.

The Browse sample

The Browse sample is a batch application that demonstrates how to browse messages on a queue using the MQGET call. The application steps through all the messages in a queue, printing the first 80 bytes of each one. You could use this application to look at the messages on a queue without changing them.

Source programs and sample run JCL are supplied in the COBOL, assembler, PL/I, and C languages (see Table 39 on page 426).

Browse sample

To start the application, you must edit and run the sample run JCL, as described in “Preparing and running sample applications for the batch environment” on page 425. You can look at messages on one of your own queues by specifying the name of the queue in the run JCL.

When you run the application (and there are some messages on the queue), the output data set looks this:

```
07/12/1998                SAMPLE QUEUE REPORT                PAGE    1
                          QUEUE MANAGER NAME : VC4
                          QUEUE NAME   : CSQ4SAMP.DEAD.QUEUE

RELATIVE
MESSAGE  MESSAGE
NUMBER  LENGTH ----- MESSAGE DATA -----
      1      740 HELLO. PLEASE CALL ME WHEN YOU GET BACK.
      2      429 CSQ4BQRM
      3      429 CSQ4BQRM
      4      429 CSQ4BQRM
      5       22 THIS IS A TEST MESSAGE
      6       8  CSQ4TEST
      7      36 CSQ4MSG - ANOTHER TEST MESSAGE.....!
      8       9  CSQ4STOP

                          ***** END OF REPORT *****
```

If there are no messages on the queue, the data set contains the headings and the “End of report” message only. If an error occurs with any of the MQI calls, the completion and reason codes are added to the output data set.

Design of the Browse sample

The Browse sample application uses a single program module—one is provided in each of the supported programming languages.

The flow through the program logic is:

1. Open a print data set and print the title line of the report. Check that names of the queue manager and queue have been passed from the run JCL. If both names have been passed, print the lines of the report that contain the names. If they have not, print an error message, close the print data set, and stop processing.

The way that the program tests the parameters it is passed from the JCL depends on the language in which the program is written—for more information, see “Language-dependent design considerations” on page 439.

2. Connect to the queue manager using the MQCONN call. If this call is not successful, print the completion and reason codes, close the print data set, and stop processing.
3. Open the queue using the MQOPEN call with the MQOO_BROWSE option. On input to this call, the program uses the connection handle returned in step 2. For the object descriptor structure (MQOD), it uses the default values for all the fields except the queue name (which was passed in step 1). If this call is not successful, print the completion and reason codes, close the print data set, and stop processing.
4. Browse the first message on the queue, using the MQGET call. On input to this call, the program specifies:
 - The connection and queue handles from steps 2 and 3
 - An MQMD structure with all fields set to their initial values

- Two options:
 - MQGMO_BROWSE_FIRST
 - MQGMO_ACCEPT_TRUNCATED_MSG
- A buffer of size 80 bytes to hold the data copied from the message

The MQGMO_ACCEPT_TRUNCATED_MSG option allows the call to complete even if the message is longer than the 80-byte buffer specified in the call. If the message is longer than the buffer, the message is truncated to fit the buffer, and the completion and reason codes are set to show this. The sample was designed so that messages are truncated to 80 characters simply to make the report easy to read. The buffer size is set by a DEFINE statement, so you can easily change it if you want to.

5. Perform the following loop until the MQGET call fails:
 - a. Print a line of the report showing:
 - The sequence number of the message (this is a count of the browse operations).
 - The true length of the message (not the truncated length). This value is returned in the *DataLength* field of the MQGET call.
 - The first 80 bytes of the message data.
 - b. Reset the *MsgId* and *CorrelId* fields of the MQMD structure to nulls
 - c. Browse the next message, using the MQGET call with these two options:
 - MQGMO_BROWSE_NEXT
 - MQGMO_ACCEPT_TRUNCATED_MSG
6. If the MQGET call fails, test the reason code to see if the call has failed because the browse cursor has got to the end of the queue. In this case, print the “End of report” message and go to step 7; otherwise, print the completion and reason codes, close the print data set, and stop processing.
7. Close the queue using the MQCLOSE call with the object handle returned in step 3 on page 438.
8. Disconnect from the queue manager using the MQDISC call with the connection handle returned in step 2 on page 438.
9. Close the print data set and stop processing.

Language-dependent design considerations

Source modules are provided for the Browse sample in four programming languages. There are two main differences between the source modules:

- When testing the parameters passed from the run JCL, the COBOL, PL/I, and assembler-language modules search for the comma character (,). If the JCL passes PARM=(,LOCALQ1), the application attempts to open queue LOCALQ1 on the default queue manager. If there is no name after the comma (or no comma), the application returns an error. The C module does not search for the comma character. If the JCL passes a single parameter (for example, PARM=('LOCALQ1')), the C module uses this as a queue name on the default queue manager.
- To keep the assembler-language module simple, it uses the date format yy/ddd (for example, 93/116) when it creates the print report. The other modules use the calendar date in mm/dd/yy format.

The Print Message sample

The Print Message sample is a simple batch application that demonstrates how to remove all the messages from a queue using the MQGET call. It also prints, for each message, the fields of the message descriptor, followed by the message data. The program prints the data both in hexadecimal and as characters (if they are

Print Message sample

printable). If a character is not printable, the program replaces it with a period character (.). You can use the program when diagnosing problems with an application that is putting messages on a queue.

You can change the application so that it browses the messages, rather than removing them from the queue. To do this, remove the comment characters from two lines in the code, as indicated in “Design of the sample” on page 441.

The application has a single source program, which is written in the C language. Sample run JCL code is also supplied (see Table 40 on page 426).

To start the application, you must edit and run the sample run JCL, as described in “Preparing and running sample applications for the batch environment” on page 425. When you run the application (and there are some messages on the queue), the output data set looks like that in Figure 42.

```
MQCONN to VC4
MQOPEN - 'CSQ4SAMP.DEAD.QUEUE'

MQGET of message number 1
****Message descriptor****
  StrucId : 'MD ' Version : 1
  Report  : 0 MsgType : 2
  Expiry  : -1 Feedback : 0
  Encoding : 785 CodedCharSetId : 500
  Format   : ' '
  Priority : 3 Persistence : 0
  MsgId    : X'C3E2D840E5C3F4404040404040404040A6FE06A95105C620'
  CorrelId : X'C3E2D840E5C3F4404040404040404040A6FE062950C2F125'
  BackoutCount : 0
  ReplyToQ      : ' '
  ReplyToQMgr   : 'VC4 '
  ** Identity Context
  UserIdentifier : 'CICSUSER '
  Account.Token  :
  X'160DD5E3E2D5C5E34BC9C7D7C2F6F1FE060D3B55B600010000000000000000'
  ApplIdentData : ' '
  ** Origin Context
  PutApplType   : '1'
  PutApplName   : 'VICAUT4 MVB5 '
  PutDate       : '19930203' PutTime : '20165982'
  ApplOriginData : ' '

```

Figure 42. Example of a report from the Print Message sample application (Part 1 of 2)

Print Message sample

```
**** Message ****
length - 429 bytes

00000000: C3E2 D8F4 C2D8 D9D4 4040 4040 4040 4040 'CSQ4BQRM      '
00000010: 4040 4040 4040 4040 4040 4040 4040 4040 '              '
00000020: 4040 4040 4040 4040 4040 4040 4040 4040 '              '
00000030: 4040 4040 4040 4040 4040 4040 4040 4040 '              '
00000040: 4040 4040 4040 4040 4040 4040 4040 4040 '              '
00000050: 4040 4040 4040 40D1 D6C8 D540 D140 4040 '          JOHN J '
00000060: 4040 4040 4040 4040 4040 40F1 F2F3 F4F5 '          12345'
00000070: F6F7 F8F9 C6C9 D9E2 E340 C7C1 D3C1 C3E3 '6789FIRST GALACT'
00000080: C9C3 40C2 C1D5 D240 4040 4040 4040 4040 'IC BANK      '
00000090: 4040 E2D6 D4C5 E3C8 C9D5 C740 C4C9 C6C6 ' SOMETHING DIFF'
000000A0: C5D9 C5D5 E340 4040 4040 4040 4040 4040 'ERENT        '
000000B0: F3F5 F0F1 F6F7 F6F2 F1F2 F1F0 F0F0 F0F0 '3501676212100000'
000000C0: D985 A297 9695 A285 4086 9996 9440 C3E2 'Response from CS'
000000D0: D8F4 E2C1 D4D7 4BC2 F74B D4C5 E2E2 C1C7 'Q4SAMP.B7.MESSAG'
000000E0: C5E2 4040 4040 4040 4040 4040 4040 4040 'ES           '
000000F0: 4040 4040 4040 4040 4040 4040 4040 4040 '              '
00000100: 4040 4040 4040 4040 4040 4040 4040 4040 '              '
00000110: 4040 4040 40D3 9681 9540 8194 96A4 95A3 '          Loan amount'
00000120: 40F1 F0F0 F0F0 F040 8696 9940 D1D6 C8D5 ' 100000 for JOHN'
00000130: 40D1 4040 4040 4040 4040 4040 4040 4040 ' J           '
00000140: 4040 4040 4040 4040 4040 4040 4040 4040 '              '
00000150: 4040 4040 4040 4040 4040 4040 4040 4040 '              '
00000160: 4040 4040 C399 8584 89A3 40A6 9699 A388 '          Credit worth'
00000170: 8995 85A2 A240 8995 8485 A740 6040 C2C1 'iness index - BA'
00000180: C440 4040 4040 4040 4040 4040 4040 4040 'D            '
00000190: 4040 4040 4040 4040 4040 4040 4040 4040 '              '
000001A0: 4040 4040 4040 4040 4040 4040 4040 40   '              '
```

```
No more messages
MQCLOSE
MQDISC
```

Figure 42. Example of a report from the Print Message sample application (Part 2 of 2)

Design of the sample

The Print message sample application uses a single program written in the C language.

The flow through the program logic is:

1. Check that names of the queue manager and queue have been passed from the run JCL. If they have not, print an error message and stop processing.
2. Connect to the queue manager using the MQCONN call. If this call is not successful, print the completion and reason codes and stop processing; otherwise print the name of the queue manager.
3. Open the queue using the MQOPEN call with the MQOO_INPUT_SHARED option.

Note: If you want the application to browse the messages rather than remove them from the queue, remove the comment characters from the line in the program that adds the MQOO_BROWSE option.

On input to this call, the program uses the connection handle returned in step 2. For the object descriptor structure (MQOD), it uses the default values for all the fields except the queue name (which was passed in step 1). If this call is not successful, print the completion and reason codes and stop processing; otherwise, print the name of the queue.

Print Message sample

4. Perform the following loop until the MQGET call fails:
 - a. Initialize the buffer to blanks so that the message data does not get corrupted by any data already in the buffer.
 - b. Set the *MsgId* and *CorrelId* fields of the MQMD structure to nulls so that the MQGET call selects the first message from the queue.
 - c. Get a message from the queue, using the MQGET call. On input to this call, the program specifies:
 - The connection and object handles from steps 2 and 3.
 - An MQMD structure with all fields set to their initial values. (Note that *MsgId* and *CorrelId* are reset to nulls for each MQGET call.)
 - The option MQGMO_NO_WAIT.

Note: If you want the application to browse the messages rather than remove them from the queue, remove the comment characters from the line in the program that adds the MQOO_BROWSE_NEXT option. When this option is used on a call against a queue for which no browse cursor has previously been used with the current object handle, the browse cursor is positioned logically before the first message.

- A buffer of size 32 KB to hold the data copied from the message.
- d. Call the printMD subroutine. This prints the name of each field in the message descriptor, followed by its contents.
 - e. Print the length of the message, followed by the message data. Each line of message data is in this format:
 - Relative position (in hexadecimal) of this part of the data
 - 16 bytes of hexadecimal data
 - The same 16 bytes of data in character format, if it is printable (nonprintable characters are replaced by periods)
5. If the MQGET call fails, test the reason code to see if the call failed because there are no more messages on the queue. In this case, print the message: "No more messages"; otherwise, print the completion and reason codes. In both cases, go to step 6.

Note: The MQGET call fails if it finds a message that has more than 32 KB of data. To change the program to handle larger messages, you could do one of the following:

- Add the MQGMO_ACCEPT_TRUNCATED_MSG option to the MQGET call, so that the call gets the first 32 KB of data and discards the remainder
 - Make the program leave the message on the queue when it finds one with this amount of data
 - Increase the size of the buffer
6. Close the queue using the MQCLOSE call with the object handle returned in step 3 on page 441.
 7. Disconnect from the queue manager using the MQDISC call with the connection handle returned in step 2 on page 441.

The Queue Attributes sample

The Queue Attributes sample is a conversational-mode CICS application that demonstrates the use of the MQINQ and MQSET calls. It shows how to inquire about the values of the *InhibitPut* and *InhibitGet* attributes of queues, and how to change them so that programs cannot put messages on, or get messages from, a queue. You may want to *lock* a queue in this way when you are testing a program.

To prevent accidental interference with your own queues, this sample works only on a queue object that has the characters CSQ4SAMP in the first eight bytes of its name. However, the source code includes comments to show you how to remove this restriction.

Source programs are supplied in the COBOL, assembler, and C languages (see Table 44 on page 430).

The assembler-language version of the sample uses reenterable code. To do this, you will notice that the code for each MQI call in that version of the sample includes the MF keyword; for example:

```
CALL MQCONN, (NAME, HCONN, COMPCODE, REASON), MF=(E, PARMAREA), VL
```

(The VL keyword means that you can use the CICS Execution Diagnostic Facility (CEDF) supplied transaction for debugging the program.) For more information on writing reenterable programs, see “Writing reenterable programs” on page 82.

To start the application, start your CICS system and use the following CICS transactions:

- For COBOL, MVC1
- For Assembler language, MAC1
- For C, MCC1

You can change the name of any of these transactions by changing the CSD data set mentioned in step 3 on page 429.

Design of the sample

When you start the sample, firstly it displays a screen map that has fields for:

- Name of the queue
- User request (valid actions are: inquire, allow, or inhibit)
- Current status of put operations for the queue
- Current status of get operations for the queue

The first two fields are for user input. The last two fields are filled by the application: they show the word INHIBITED or the word ALLOWED.

The application validates the values you enter in the first two fields. It checks that the queue name starts with the characters CSQ4SAMP and that you entered one of the three valid requests in the Action field. The application converts all your input to uppercase, so you cannot use any queues with names that contain lowercase characters.

If you enter ‘inquire’ in the Action field, the flow through the program logic is:

1. Open the queue using the MQOPEN call with the MQOO_INQUIRE option
2. Call MQINQ using the selectors MQIA_INHIBIT_GET and MQIA_INHIBIT_PUT
3. Close the queue using the MQCLOSE call

Queue Attributes sample

4. Analyze the attributes that are returned in the *IntAttrs* parameter of the MQINQ call and move the words 'INHIBITED' or 'ALLOWED', as appropriate, to the relevant screen fields

If you enter 'inhibit' in the Action field, the flow through the program logic is:

1. Open the queue using the MQOPEN call with the MQOO_SET option
2. Call MQSET using the selectors MQIA_INHIBIT_GET and MQIA_INHIBIT_PUT, and with the values MQQA_GET_INHIBITED and MQQA_PUT_INHIBITED in the *IntAttrs* parameter
3. Close the queue using the MQCLOSE call
4. Move the word 'INHIBITED' to the relevant screen fields

If you enter 'allow' in the Action field, the application performs similar processing to that for an 'inhibit' request. The only differences are the settings of the attributes and the words displayed on the screen.

When the application opens the queue, it uses the default connection handle to the queue manager. (CICS establishes a connection to the queue manager when you start your CICS system.) The application can trap the following errors at this stage:

- The application is not connected to the queue manager
- The queue does not exist
- The user is not authorized to access the queue
- The application is not authorized to open the queue

For other MQI errors, the application displays the completion and reason codes.

The Mail Manager sample

The Mail Manager sample application is a suite of programs that demonstrates the sending and receiving of messages, both within a single environment and across different environments. The application is a simple electronic mailing system that allows users to exchange messages, even if they use different queue managers.

The application demonstrates how to create queues using the MQOPEN call and by putting MQSeries for OS/390 commands on the system-command input queue.

Three versions of the application are provided:

- A CICS application written in COBOL
- A TSO application written in COBOL
- A TSO application written in C

Preparing the sample

The Mail Manager is provided in versions that run in two environments. The preparation you must carry out before you run the application depends on the environment you want to use.

A user can access mail queues and nickname queues from both TSO and CICS so long as their sign-on user IDs are the same on each system.

Before you can send messages to another queue manager, you must set up a message channel to that queue manager. To do this, use the channel control function of MQSeries, described in the *MQSeries Intercommunication* book.

Preparing the sample for the TSO environment

Follow these steps:

1. Prepare the sample as described in “Preparing sample applications for the TSO environment” on page 427.
2. Tailor the CLIST provided for the sample to define:
 - The location of the panels
 - The location of the message file
 - The location of the load modules
 - The name of the queue manager you want to use with the application

A separate CLIST is provided for each language version of the sample:

For the COBOL version: CSQ4RVD1

For the C version: CSQ4RCD1

3. Ensure that the queues used by the application are available on the queue manager. (The queues are defined in CSQ4CVD.)

Note: VS COBOL II does not support multitasking with ISPF. This means that you cannot use the Mail Manager sample application on both sides of a split screen. If you do, the results are unpredictable.

Running the sample

To start the sample in the TSO environment, execute your tailored version of the CLIST from the TSO command processor within ISPF.

To start the sample in the CICS Transaction Server for OS/390 environment, run transaction MAIL. If you have not already signed-on to CICS, the application prompts you to enter a user ID to which it can send your mail.

When you start the application, it opens your mail queue. If this queue does not already exist, the application creates one for you. Mail queues have names of the form CSQ4SAMP.MAILMGR.*userid*, where *userid* depends on the environment:

In TSO

The user’s TSO ID

In CICS

The user’s CICS sign-on or the user ID entered by the user when prompted when the Mail Manager started

All parts of the queue names that the Mail Manager uses must be uppercase.

The application then presents a menu panel that has options for:

- Read incoming mail
- Send mail
- Create nickname

The menu panel also shows you how many messages are waiting on your mail queue. Each of the menu options displays a further panel:

Read incoming mail

The Mail Manager displays a list of the messages that are on your mail queue. (Note that only the first 99 messages on the queue are displayed.)

Mail Manager sample

For an example of this panel, see Figure 45 on page 449. When you select a message from this list, the contents of the message are displayed (see Figure 46 on page 450).

Send mail

A panel prompts you to enter:

- The name of the user to whom you want to send a message
- The name of the queue manager that owns their mail queue
- The text of your message

In the user name field you can enter either a user ID or a nickname that you created using the Mail Manager. You can leave the queue manager name field blank if the user's mail queue is owned by the same queue manager that you are using, and you must leave it blank if you entered a nickname in the user name field:

- If you specify only a user name, the program first assumes that the name is a nickname, and sends the message to the object defined by that name. If there is no such nickname, the program attempts to send the message to a local queue of that name.
- If you specify both a user name and a queue manager name, the program sends the message to the mail queue that is defined by those two names.

For example, if you want to send a message to user JONESM on remote queue manager QM12, you could send them a message in either of two ways:

- Use both fields to specify user JONESM at queue manager QM12.
- Define a nickname (for example, MARY) for that user and send them a message by putting MARY in the user name field and nothing in the queue manager name field.

Create nickname

You can define an easy-to-remember name that you can use when you send a message to another user who you contact frequently. You are prompted to enter the user ID of the other user and the name of the queue manager that owns their mail queue.

Nicknames are queues that have names of the form `CSQ4SAMP.MAILMGR.userid.nickname`, where *userid* is your own user ID and *nickname* is the nickname that you want to use. With names structured in this way, users can each have their own set of nicknames.

The type of queue that the program creates depends on how you fill in the fields of the Create Nickname panel:

- If you specify only a user name, or the queue manager name is the same as that of the queue manager to which the Mail Manager is connected, the program creates an alias queue.
- If you specify both a user name and a queue manager name (and the queue manager is not the one to which the Mail Manager is connected), the program creates a local definition of a remote queue. The program does not check the existence of the queue to which this definition resolves, or even that the remote queue manager exists.

For example, if your own user ID is SMITHK and you create a nickname called MARY for user JONESM (who uses the remote queue manager QM12), the nickname program creates a local definition of a remote queue named `CSQ4SAMP.MAILMGR.SMITHK.MARY`. This definition resolves to

Mary's mail queue, which is CSQ4SAMP.MAILMGR.JONESM at queue manager QM12. If you are using queue manager QM12 yourself, the program instead creates an alias queue of the same name (CSQ4SAMP.MAILMGR.SMITHK.MARY).

The C version of the TSO application makes greater use of ISPF's message-handling capabilities than does the COBOL version. You may notice that different error messages are displayed by the C and COBOL versions.

Design of the sample

The following sections describe each of the programs that comprise the Mail Manager sample application. The relationships between the programs and the panels that the application uses is shown in Figure 43 for the TSO version, and Figure 44 on page 448 for the CICS Transaction Server for OS/390 version.

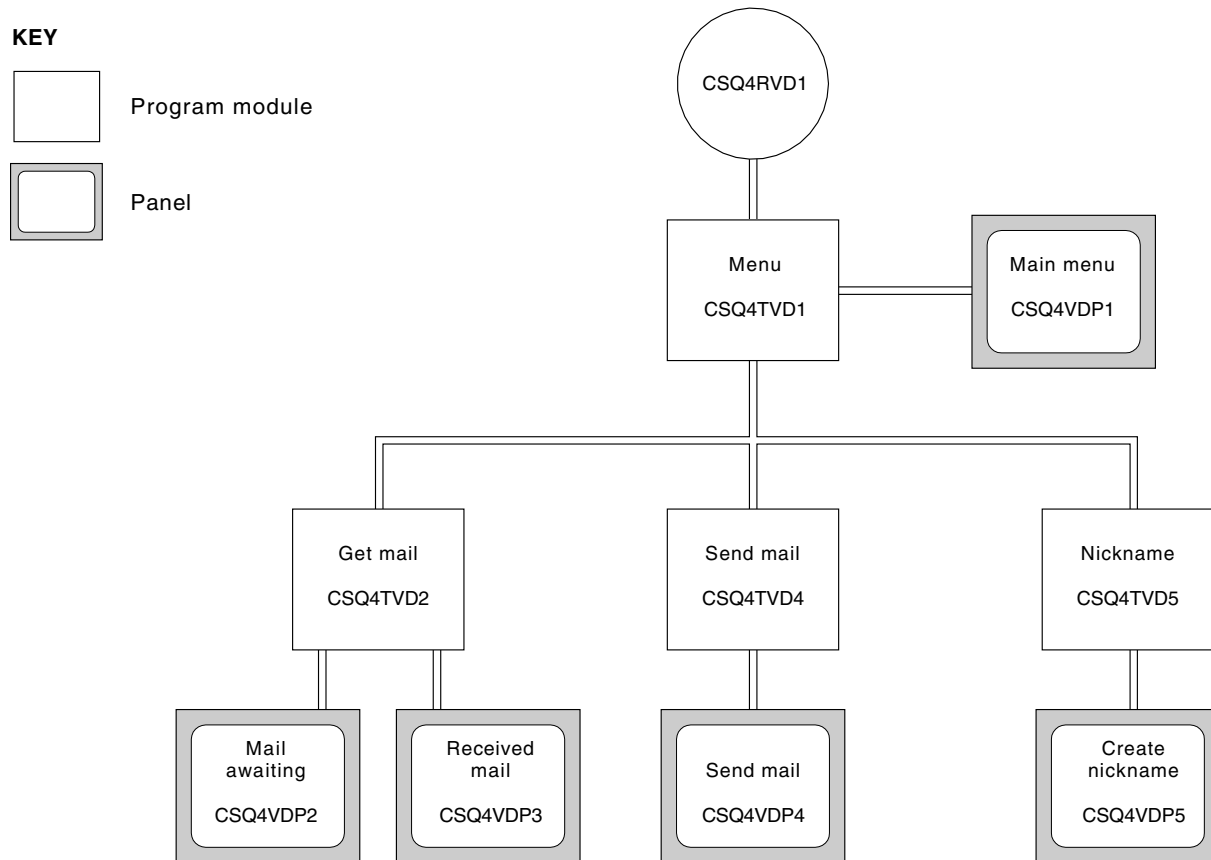


Figure 43. Programs and panels for the TSO versions of the Mail Manager. This figure shows the names for the COBOL version.

Mail Manager sample

KEY

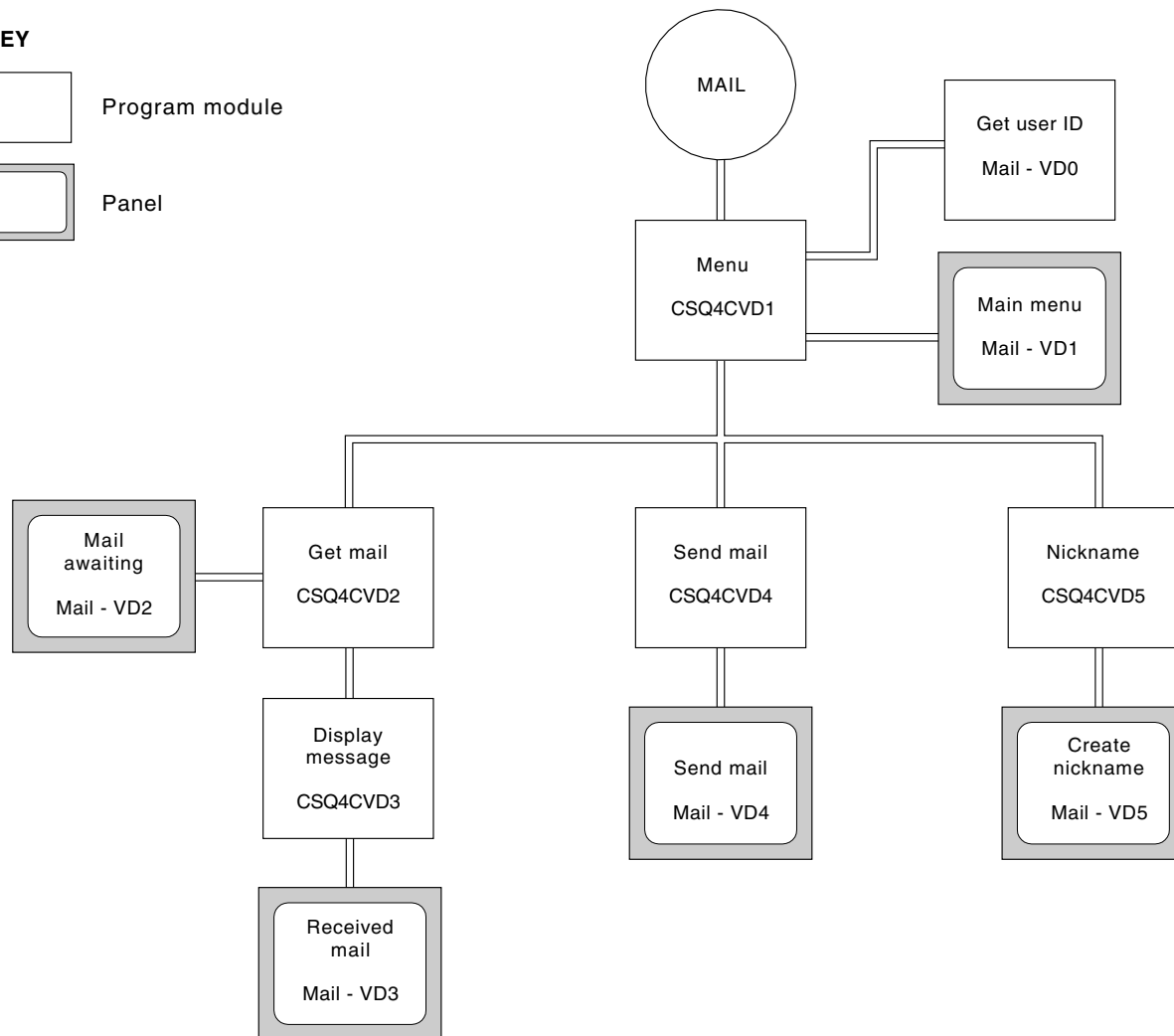
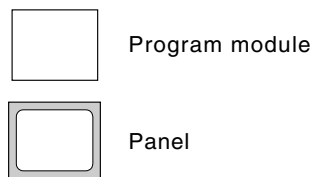


Figure 44. Programs and panels for the CICS version of the Mail Manager

Menu program

In the TSO environment, the menu program is invoked by the CLIST. In the CICS environment, the program is invoked by transaction MAIL.

The menu program is the initial program in the suite. It displays the menu and invokes the other programs when they are selected from the menu.

The program first obtains the user's ID:

- In the CICS version of the program, if the user has signed on to CICS, the user ID is obtained by using the CICS command ASSIGN USERID. If the user has not signed on, the program displays the sign-on panel (CSQ4VD0) to prompt the user to enter a user ID. There is no security processing within this program—the user can give *any* user ID.
- In the TSO version, the user's ID is obtained from TSO in the CLIST. It is passed to the menu program as a variable in the ISPF shared pool.

After the program has obtained the user ID, it checks to ensure that the user has a mail queue (CSQ4SAMP.MAILMGR.userid). If a mail queue does not exist, the program creates one by putting a message on the system-command input queue.

Mail Manager sample

The message contains the MQSeries for OS/390 command DEFINE QLOCAL. The object definition that this command uses sets the maximum depth of the queue to 9999 messages.

The program also creates a temporary dynamic queue to handle replies from the system-command input queue. To do this, the program uses the MQOPEN call, specifying the SYSTEM.DEFAULT.MODEL.QUEUE as the template for the dynamic queue. The queue manager creates the temporary dynamic queue with a name that has the prefix CSQ4SAMP; the remainder of the name is generated by the queue manager.

The program then opens the user's mail queue and finds the number of messages on the queue by inquiring about the current depth of the queue. To do this, the program uses the MQINQ call, specifying the MQIA_CURRENT_Q_DEPTH selector.

The program then performs a loop that displays the menu and processes the selection that the user makes. The loop is stopped when the user presses the PF3 key. When a valid selection is made, the appropriate program is started; otherwise an error message is displayed.

Get-mail and display-message programs

In the TSO versions of the application, the get-mail and display-message functions are performed by the same program. In the CICS version of the application, these functions are performed by separate programs.

The Mail Awaiting panel (see Figure 45 for an example) shows all the messages that are on the user's mail queue. To create this list, the program uses the MQGET call to browse all the messages on the queue, saving information about each one. In addition to the information displayed, the program records the *MsgId* and *CorrelId* of each message.

```
----- MQSeries for OS/390 Sample Programs ----- ROW 16 OF 29
COMMAND ==>                                     Scroll   ==> PAGE
                                                USERID - NTSFV02
                                                QMGR   - VC4

                Mail Manager System
                Mail Awaiting

        Msg      Mail      Date      Time
        No      From      Sent      Sent
16
    16      Deleted
    17      JOHNJ      01/06/1993  12:52:02
    18      JOHNJ      01/06/1993  12:52:02
    19      JOHNJ      01/06/1993  12:52:03
    20      JOHNJ      01/06/1993  12:52:03
    21      JOHNJ      01/06/1993  12:52:03
    22      JOHNJ      01/06/1993  12:52:04
    23      JOHNJ      01/06/1993  12:52:04
    24      JOHNJ      01/06/1993  12:52:04
    25      JOHNJ      01/06/1993  12:52:05
    26      JOHNJ      01/06/1993  12:52:05
    27      JOHNJ      01/06/1993  12:52:05
    28      JOHNJ      01/06/1993  12:52:06
    29      JOHNJ      01/06/1993  12:52:06
```

Figure 45. Example of a panel showing a list of waiting messages

From the Mail Awaiting panel the user can select one message and display the contents of the message (see Figure 46 on page 450 for an example). The program

Mail Manager sample

uses the MQGET call to remove this message from the queue, using the *MsgId* and *CorrelId* that the program noted when it browsed all the messages. This MQGET call is performed using the MQGMO_SYNCPOINT option. The program displays the contents of the message, then declares a syncpoint: this commits the MQGET call, so the message now no longer exists.

```
----- MQSeries for OS/390 Sample Programs -----
COMMAND ==>

                                Mail Manager System      USERID - NTSFV02
                                Received Mail             QMGR   - VC4

Mail sent from JOHNJ   at VC4

Sent on the 01/06/1993 at 12:52:02
----- Message -----
HELLO FROM JOHNJ

```

Figure 46. Example of a panel showing the contents of a message

An obvious extension to the function provided by the Mail Manager is to give the user the option to leave the message on the queue after viewing its contents. To do this, you would have to back out the MQGET call that removes the message from the queue, after displaying the message.

Send-mail program

When the user has completed the Send Mail panel, the send-mail program puts the message on the receiver's mail queue. To do this, the program uses the MQPUT1 call. The destination of the message depends on how the user has filled the fields in the Send Mail panel:

- If the user has specified only a user name, the program first assumes that the name is a nickname, and sends the message to the object defined by that name. If there is no such nickname, the program attempts to send the message to a local queue of that name.
- If the user has specified both a user name and a queue manager name, the program sends the message to the mail queue that is defined by those two names.

The program does not accept blank messages, and it removes leading blanks from each line of the message text.

If the MQPUT1 call is successful, the program displays a message that shows the user name and queue manager name to which the message was put. If the call is unsuccessful, the program checks specifically for the reason codes that indicate the queue or the queue manager do not exist; these are MQRC_UNKNOWN_OBJECT_NAME and MQRC_UNKNOWN_OBJECT_Q_MGR.

The program displays its own error message for each of these errors; for other errors, the program displays the completion and reason codes returned by the call.

Nickname program

When the user defines a nickname, the program creates a queue that has the nickname as part of its name. The program does this by putting a message on the system-command input queue. The message contains the MQSeries for OS/390 command DEFINE QALIAS or DEFINE QREMOTE. The type of queue that the program creates depends on how the user has filled the fields of the Create Nickname panel:

- If the user has specified only a user name, or the queue manager name is the same as that of the queue manager to which the Mail Manager is connected, the program creates an alias queue.
- If the user has specified both a user name and a queue manager name, (and the queue manager is not the one to which the Mail Manager is connected), the program creates a local definition of a remote queue. The program does not check the existence of the queue to which this definition resolves, or even that the remote queue manager exists.

The program also creates a temporary dynamic queue to handle replies from the system-command input queue.

If the queue manager cannot create the nickname queue for a reason that the program expects (for example, the queue already exists), the program displays its own error message. If the queue manager cannot create the queue for a reason that the program does not expect, the program displays up to two of the error messages that are returned to the program by the command server.

Note: For each nickname, the nickname program creates only an alias queue or a local definition of a remote queue. The local queues to which these queue names resolve are created only when the user ID that is contained in the nickname is used to start the Mail Manager application.

The Credit Check sample

The Credit Check sample application is a suite of programs that demonstrates how to use many of the features provided by MQSeries for OS/390. It shows how the many component programs of an application can pass messages to each other using message queuing techniques.

The sample can run as a stand-alone CICS application. However, to demonstrate how to design a message queuing application that uses the facilities provided by both the CICS and IMS environments, one module is also supplied as an IMS batch message processing program. This extension to the sample is described in "The IMS extension to the Credit Check sample" on page 462.

You can also run the sample on more than one queue manager, and send messages between each instance of the application. To do this, see "The Credit Check sample with multiple queue managers" on page 462.

The CICS programs are delivered in C and COBOL. The single IMS program is delivered only in C. The supplied data sets are shown in Table 46 on page 430 and Table 47 on page 432.

Credit Check sample

The application demonstrates a method of assessing the risk when bank customers ask for loans. The application shows how a bank could work in two ways to process loan requests:

- When dealing directly with a customer, bank staff want immediate access to account and credit-risk information.
- When dealing with written applications, bank staff can submit a series of requests for account and credit-risk information, and deal with the replies at a later time.

The financial and security details in the application have been kept simple so that the message queuing techniques are clear.

Preparing and running the Credit Check sample

To prepare and run the Credit Check sample, perform the following steps:

1. Create the VSAM data set that holds information about some example accounts. Do this by editing and running the JCL supplied in data set CSQ4FILE.
2. Perform the steps in “Preparing the sample applications for the CICS environment” on page 428. (The additional steps you must perform if you want to use the IMS extension to the sample are described in “The IMS extension to the Credit Check sample” on page 462.)
3. Start the CKTI trigger monitor (supplied with MQSeries for OS/390) against queue CSQ4SAMP.INITIATION.QUEUE, using the CICS transaction CKQC.
4. To start the application, start your CICS system and use the transaction MVB1.
5. Select **Immediate** or **Batch** inquiry from the first panel.

The immediate and batch inquiry panels are similar—Figure 47 shows the Immediate Inquiry panel.

```
CSQ4VB2          MQSeries for OS/390 Sample Programs
                  Credit Check - Immediate Inquiry

Specify details of the request, then press Enter.
  Name . . . . . _____
  Social security number  _ _ _ _
  Bank account name . .  _____
  Account number . . . .  _____
  Amount requested . . .  012345
Response from CHECKING ACCOUNT for name : _____
  Account information not found
  Credit worthiness index - NOT KNOWN
..
..
..
..
..
..
..
..
..
MESSAGE LINE
F1=Help F3=Exit F5=Make another inquiry
```

Figure 47. Immediate Inquiry panel for the Credit Check sample application

6. Enter an account number and loan amount in the appropriate fields. See “Entering information in the inquiry panels” on page 453 for guidance on what information you should enter in these fields.

Entering information in the inquiry panels

The Credit Check sample application checks that the data you enter in the 'Amount requested' field of the inquiry panels is in the form of integers.

If you enter one of the following account numbers, the application finds the appropriate account name, average account balance, and credit worthiness index in the VSAM data set CSQ4BAQ:

2222222222
3111234329
3256478962
3333333333
3501676212
3696879656
4444444444
5555555555
6666666666
7777777777

You can enter any, or no, information in the other fields. The application retains any information that you do enter and returns the same information in the reports that it generates.

Design of the sample

This section describes the design of each of the programs that comprise the Credit Check sample application. For a discussion of some of the techniques that were considered during the design of the application, see "Design considerations" on page 460.

Figure 48 on page 454 shows the programs that make up the application, and also the queues that these programs serve. In this figure, the prefix CSQ4SAMP has been omitted from all the queue names to make the figure easier to understand.

Credit Check sample

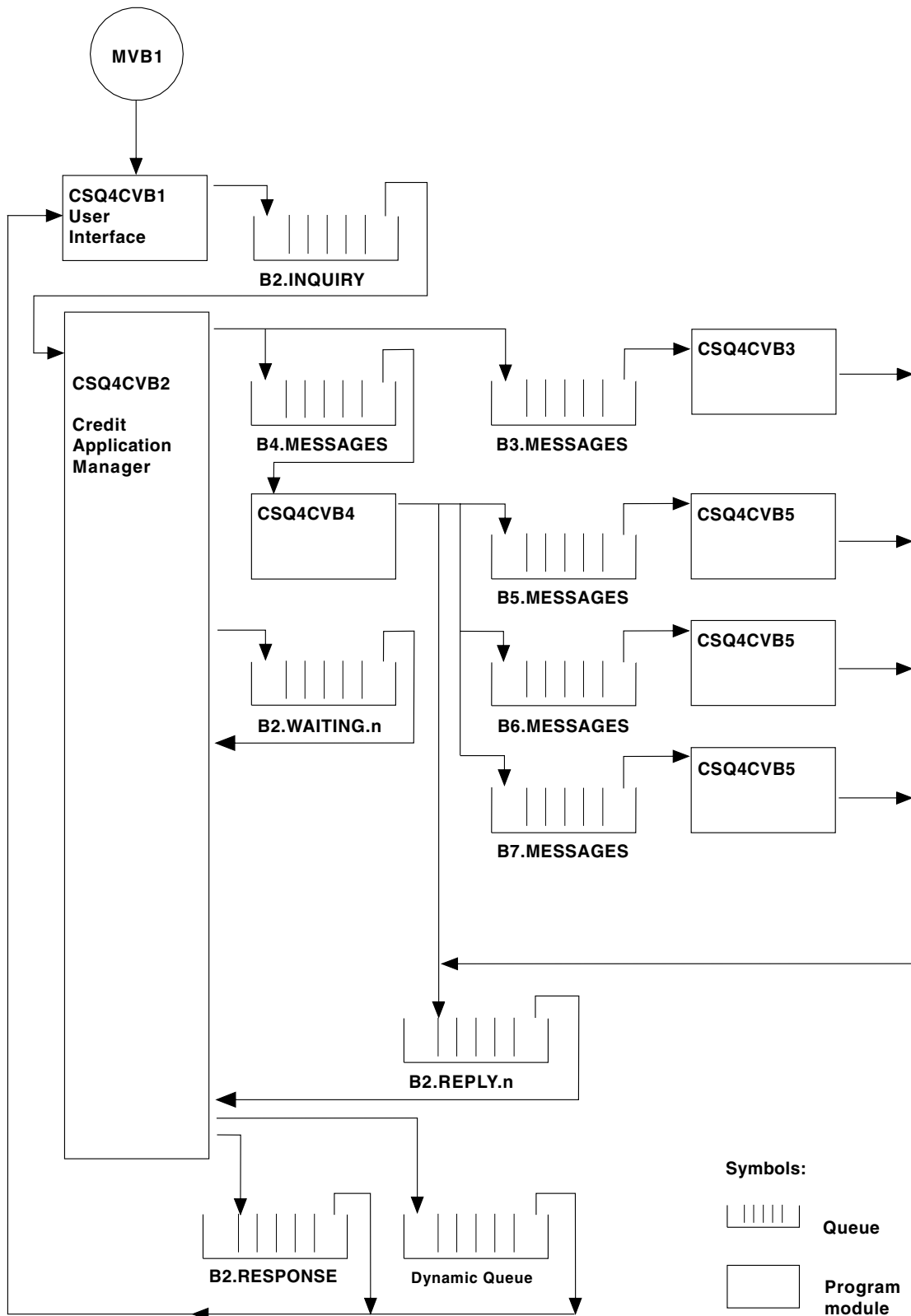


Figure 48. Programs and queues for the Credit Check sample application (COBOL programs only). In the sample application, the queue names shown in this figure have the prefix 'CSQ4SAMP.'

User-interface program (CSQ4CVB1)

When you start the conversational-mode CICS transaction MVB1, this starts the user-interface program for the application. This program puts inquiry messages on queue CSQ4SAMP.B2.INQUIRY and gets replies to those inquiries from a reply-to queue that it specifies when it makes the inquiry. From the user interface you can submit either immediate or batch inquiries:

- For immediate inquiries, the program creates a temporary dynamic queue that it uses as a reply-to queue. This means that each inquiry has its own reply-to queue.
- For batch inquiries, the user-interface program gets replies from the queue CSQ4SAMP.B2.RESPONSE. For simplicity, the program gets replies for all its inquiries from this one reply-to queue. It is easy to see that a bank might want to use a separate reply-to queue for each user of MVB1, so that they could each see replies to only those inquiries they had initiated.

Important differences between the properties of messages used in the application when in batch and immediate mode are:

- For batch working, the messages have a low priority, so they are processed after any loan requests that are entered in immediate mode. Also, the messages are persistent, so they are recovered if the application or the queue manager has to restart.
- For immediate working, the messages have a high priority, so they are processed before any loan requests that are entered in batch mode. Also, messages are not persistent so they are discarded if the application or the queue manager has to restart.

However, in all cases, the properties of loan request messages are propagated throughout the application. So, for example, all messages that result from a high-priority request will also have a high priority.

Credit application manager (CSQ4CVB2)

The Credit Application Manager (CAM) program performs most of the processing for the Credit Check application.

The CAM is started by the CKTI trigger monitor (supplied with MQSeries for OS/390) when a trigger event occurs on either queue CSQ4SAMP.B2.INQUIRY or queue CSQ4SAMP.B2.REPLY.*n*, where *n* is an integer that identifies one of a set of reply queues. The trigger message contains data that includes the name of the queue on which the trigger event occurred.

The CAM uses queues with names of the form CSQ4SAMP.B2.WAITING.*n* to store information about inquiries it is processing. The queues are named so that they are each paired with a reply-to queue; for example, queue CSQ4SAMP.B2.WAITING.3 contains the input data for a particular inquiry, and queue CSQ4SAMP.B2.REPLY.3 contains a set of reply messages (from programs that query databases) all relating to that same inquiry. To understand the reasons behind this design, see “Separate inquiry and reply queues in the CAM” on page 460.

Start-up logic: If the trigger event occurs on queue CSQ4SAMP.B2.INQUIRY, the CAM opens the queue for shared access. It then tries to open each reply queue until a free one is found. If it cannot find a free reply queue, the CAM logs the fact and terminates normally.

If the trigger event occurs on queue CSQ4SAMP.B2.REPLY.*n*, the CAM opens the queue for exclusive access. If the return code reports that the object is already in

Credit Check sample

use, the CAM terminates normally. If any other error occurs, the CAM logs the error and terminates. The CAM opens the corresponding waiting queue and the inquiry queue, then starts getting and processing messages. From the waiting queue, the CAM recovers details of partially-completed inquiries.

For the sake of simplicity in this sample, the names of the queues used are held in the program. In a business environment, the queue names would probably be held in a file accessed by the program.

Getting a message: The CAM first attempts to get a message from the inquiry queue using the MQGET call with the MQGMO_SET_SIGNAL option. If a message is available immediately, the message is processed; if no message is available, a signal is set.

The CAM then attempts to get a message from the reply queue, again using the MQGET call with the same option. If a message is available immediately, the message is processed; otherwise a signal is set.

When both signals are set, the program waits until one of the signals is posted. If a signal is posted to indicate a message is available, the message is retrieved and processed. If the signal expires or the queue manager is terminating, the program terminates.

Processing the message retrieved: A message retrieved by the CAM may be one of four types:

- An inquiry message
- A reply message
- A propagation message
- An unexpected or unwanted message

The CAM processes these messages as follows:

Inquiry message

Inquiry messages come from the user-interface program. It creates an inquiry message for each loan request.

For all loan requests, the CAM requests the average balance of the customer's checking account. It does this by putting a request message on alias queue CSQ4SAMP.B2.OUTPUT.ALIAS. This queue name resolves to queue CSQ4SAMP.B3.MESSAGES, which is processed by the checking-account program, CSQ4CVB3. When the CAM puts a message on this alias queue, it specifies the appropriate CSQ4SAMP.B2.REPLY.n queue for the reply-to queue. An alias queue is used here so that program CSQ4CVB3 can easily be replaced by another program that processes a base queue of a different name. To do this, you simply redefine the alias queue so that its name resolves to the new queue. Also, you could assign differing access authorities to the alias queue and to the base queue.

If a user requests a loan that is larger than 10000 units, the CAM initiates checks on other databases as well. It does this by putting a request message on queue CSQ4SAMP.B4.MESSAGES, which is processed by the distribution program, CSQ4CVB4. The process serving this queue propagates the message to queues served by programs that have access to other records such as credit card history, savings accounts, and mortgage payments. The data from these programs is returned to the reply-to queue specified in the put operation. Additionally, a propagation message is sent to the reply-to queue by this program to specify how many propagation messages have been sent.

In a business environment, the distribution program would probably reformat the data provided to match the format required by each of the other types of bank account.

Any of the queues referred to here can be on a remote system.

For each inquiry message, the CAM initiates an entry in the memory-resident Inquiry Record Table (IRT). This record contains:

- The *MsgId* of the inquiry message
- In the ReplyExp field, the number of responses expected (equal to the number of messages sent)
- In the ReplyRec field, the number of replies received (zero at this stage)
- In the PropsOut field, an indication of whether a propagation message is expected

The CAM copies the inquiry message on to the waiting queue with:

- *Priority* set to 3
- *CorrelId* set to the *MsgId* of the inquiry message
- The other message-descriptor fields set to those of the inquiry message

Propagation message

A propagation message contains the number of queues to which the distribution program has forwarded the inquiry. The message is processed as follows:

1. Add to the ReplyExp field of the appropriate record in the IRT the number of messages sent. This information is in the message.
2. Increment by 1 the ReplyRec field of the record in the IRT.
3. Decrement by 1 the PropsOut field of the record in the IRT.
4. Copy the message on to the waiting queue. The CAM sets the *Priority* to 2 and the other fields of the message descriptor to those of the propagation message.

Reply message

A reply message contains the response to one of the requests to the checking-account program or to one of the agency-query programs. Reply messages are processed as follows:

1. Increment by 1 the ReplyRec field of the record in the IRT.
2. Copy the message on to the waiting queue with *Priority* set to 1 and the other fields of the message descriptor set to those of the reply message.
3. If ReplyRec = ReplyExp, and PropsOut = 0, set the MsgComplete flag.

Other messages

The application does not expect other messages. However, the application might receive messages broadcast by the system, or reply messages with unknown *CorrelIds*.

The CAM puts these messages on queue CSQ4SAMP.DEAD.QUEUE, where they can be examined. If this put operation fails, the message is lost and the program continues. For more information on the design of this part of the program, see "How the sample handles unexpected messages" on page 460.

Sending an answer: When the CAM has received all the replies it is expecting for an inquiry, it processes the replies and creates a single response message. It consolidates into one message all the data from all reply messages that have the

Credit Check sample

same *CorrelId*. This response is put on the reply-to queue specified in the original loan request. The response message is put within the same unit of work that contains the retrieval of the final reply message. This is to simplify recovery by ensuring that there is never a completed message on queue CSQ4SAMP.B2.WAITING.n.

Recovery of partially-completed inquiries: The CAM copies on to queue CSQ4SAMP.B2.WAITING.n all the messages that it receives. It sets the fields of the message descriptor like this:

- *Priority* is determined by the type of message:
 - For request messages, priority = 3
 - For datagrams, priority = 2
 - For reply messages, priority = 1
- *CorrelId* is set to the *MsgId* of the loan request message
- Other MQMD fields are copied from those of the received message

When an inquiry has been completed, the messages for a specific inquiry are removed from the waiting queue during answer processing. Therefore, at any time, the waiting queue contains all messages relevant to in-progress inquiries. These messages are used to recover details of in-progress inquiries if the program has to restart. The different priorities are set so that inquiry messages are recovered before propagations or reply messages.

Checking-account program (CSQ4CVB3)

The checking-account program is started by a trigger event on queue CSQ4SAMP.B3.MESSAGES. After it has opened the queue, this program gets a message from the queue using the MQGET call with the wait option, and with the wait interval set to 30 seconds.

The program searches VSAM data set CSQ4BAQ for the account number in the loan request message. It retrieves the corresponding account name, average balance, and credit worthiness index, or notes that the account number is not in the data set.

The program then puts a reply message (using the MQPUT1 call) on the reply-to queue named in the loan request message. For this reply message, the program:

- Copies the *CorrelId* of the loan request message
- Uses the MQPMO_PASS_IDENTITY_CONTEXT option

The program continues to get messages from the queue until the wait interval expires.

Distribution program (CSQ4CVB4)

The distribution program is started by a trigger event on queue CSQ4SAMP.B4.MESSAGES. To simulate the distribution of the loan request to other agencies that have access to records such as credit card history, savings accounts, and mortgage payments, the program puts a copy of the same message on all the queues in the namelist CSQ4SAMP.B4.NAMELIST. There are three of these queues, with names of the form CSQ4SAMP.Bn.MESSAGES, where *n* is 5, 6, or 7. In a business application, the agencies could be at separate locations, so these queues could be remote queues. If you want to modify the sample application to show this, see “The Credit Check sample with multiple queue managers” on page 462.

The distribution program performs the following steps:

1. From the namelist, gets the names of the queues the program is to use. The program does this by using the MQINQ call to inquire about the attributes of the namelist object.
2. Opens these queues and also CSQ4SAMP.B4.MESSAGES.
3. Performs the following loop until there are no more messages on queue CSQ4SAMP.B4.MESSAGES:
 - a. Get a message using the MQGET call with the wait option, and with the wait interval set to 30 seconds.
 - b. Put a message on each queue listed in the namelist, specifying the name of the appropriate CSQ4SAMP.B2.REPLY.n queue for the reply-to queue. The program copies the *CorrelId* of the loan request message to these copy messages, and it uses the MQPMO_PASS_IDENTITY_CONTEXT option on the MQPUT call.
 - c. Send a datagram message to queue CSQ4SAMP.B2.REPLY.n to show how many messages it has successfully put.
 - d. Declare a syncpoint.

Agency-query program (CSQ4CVB5/CSQ4CCB5)

The agency-query program is supplied as both a COBOL program and a C program. Both programs have the same design. This shows that programs of different types can easily coexist within an MQSeries application, and that the program modules that comprise such an application can easily be replaced.

An instance of the program is started by a trigger event on any of these queues:

- For the COBOL program (CSQ4CVB5):
 - CSQ4SAMP.B5.MESSAGES
 - CSQ4SAMP.B6.MESSAGES
 - CSQ4SAMP.B7.MESSAGES
- For the C program (CSQ4CCB5), queue CSQ4SAMP.B8.MESSAGES

Note: If you want to use the C program, you must alter the definition of the namelist CSQ4SAMP.B4.NAMELIST to replace the queue CSQ4SAMP.B7.MESSAGES with CSQ4SAMP.B8.MESSAGES. To do this, you can use any one of:

- The MQSeries for OS/390 operations and control panels
- The ALTER NAMELIST command (described in the *MQSeries MQSC Command Reference* manual)
- The CSQUTIL utility (described in the *MQSeries for OS/390 System Administration Guide*)

After it has opened the appropriate queue, this program gets a message from the queue using the MQGET call with the wait option, and with the wait interval set to 30 seconds.

The program simulates the search of an agency's database by searching the VSAM data set CSQ4BAQ for the account number that was passed in the loan request message. It then builds a reply that includes the name of the queue it is serving and a credit-worthiness index. To simplify the processing, the credit-worthiness index is selected at random.

When putting the reply message, the program uses the MQPUT1 call and:

- Copies the *CorrelId* of the loan request message

Credit Check sample

- Uses the MQPMO_PASS_IDENTITY_CONTEXT option

The program sends the reply message to the reply-to queue named in the loan request message. (The name of the queue manager that owns the reply-to queue is also specified in the loan request message.)

Design considerations

This section discusses:

- Why the CAM uses separate inquiry and reply queues
- How the sample handles errors
- How the sample handles unexpected messages
- How the sample uses syncpoints
- How the sample uses message context information

Separate inquiry and reply queues in the CAM

The application could use a single queue for both inquiries and replies, but it was designed to use separate queues for the following reasons:

- When the program is handling the maximum number of inquiries, further inquiries can be left on the queue. If a single queue were being used, these would have to be taken off the queue and stored elsewhere.
- Other instances of the CAM could be started automatically to service the same inquiry queue if message traffic was high enough to warrant it. But the program must track in-progress inquiries, and to do this it must get back all replies to inquiries it has initiated. If only one queue were used, the program would have to browse the messages to see if they were for this program or for another. This would make the operation much less efficient.

The application can support multiple CAMs and can recover in-progress inquiries effectively by using paired reply-to and waiting queues.

- The program can wait on multiple queues effectively by using signaling.

How the sample handles errors

The user-interface program handles errors very simply by reporting them directly to the user. The other programs do not have user interfaces, so they have to handle errors in other ways. Also, in many situations (for example, if an MQGET call fails) these other programs do not know the identity of the user of the application.

The other programs put error messages on a CICS temporary storage queue called CSQ4SAMP. You can browse this queue using the CICS-supplied transaction CEBR. The programs also write error messages to the CICS CSML log.

How the sample handles unexpected messages

When you design a message-queuing application, you must decide how to handle messages that arrive on a queue unexpectedly. The two basic choices are:

- The application must do no more work until it has processed the unexpected message. This probably means that the application must notify an operator, terminate itself, and ensure that it is not restarted automatically (it can do this by setting triggering off). This choice means that all processing for the application can be halted by a single unexpected message, and the intervention of an operator is required to restart the application.
- The application must remove the message from the queue it is serving, put the message in another location, and continue processing. The best place to put this message is on the system dead-letter queue.

If you choose the second option:

- An operator, or another program, should examine the messages that are put on the dead-letter queue to find out where the messages are coming from.
- An unexpected message is lost if it cannot be put on the dead-letter queue.
- An long unexpected message is truncated if it is longer than the limit for messages on the dead-letter queue, or longer than the buffer size in the program.

To ensure that the application smoothly handles all inquiries with minimal impact from outside activities, the Credit Check sample application uses the second option. To allow you to keep the sample separate from other applications that use the same queue manager, the Credit Check sample does not use the system dead-letter queue: instead, it uses its own dead-letter queue. This queue is named CSQ4SAMP.DEAD.QUEUE. The sample truncates any messages that are longer than the buffer area provided for the sample programs. You can use the Browse sample application to browse messages on this queue, or use the Print Message sample application to print the messages together with their message descriptors.

However, if you extend the sample to run across more than one queue manager, unexpected messages, or messages that cannot be delivered, could be put on the system dead-letter queue by the queue manager.

How the sample uses syncpoints

The programs in the Credit Check sample application declare syncpoints to ensure that:

- Only one reply message is sent in response to each expected message
- Multiple copies of unexpected messages are never put on the sample's dead-letter queue
- The CAM can recover the state of all partially-completed inquiries by getting persistent messages from its waiting queue

To achieve this, a single unit of work is used to cover the getting of a message, the processing of that message, and any subsequent put operations.

How the sample uses message context information

When the user-interface program (CSQ4CVB1) sends messages, it uses the MQPMO_DEFAULT_CONTEXT option. This means that the queue manager generates both identity and origin context information. The queue manager gets this information from the transaction that started the program (MVB1) and from the user ID that started the transaction.

When the CAM sends inquiry messages, it uses the MQPMO_PASS_IDENTITY_CONTEXT option. This means that the identity context information of the message being put is copied from the identity context of the original inquiry message. With this option, origin context information is generated by the queue manager.

When the CAM sends reply messages, it uses the MQPMO_ALTERNATE_USER_AUTHORITY option. This causes the queue manager to use an alternate user ID for its security check when the CAM opens a reply-to queue. The CAM uses the user ID of the submitter of the original inquiry message. This means that users are allowed to see replies to only those inquiries they have originated. The alternate user ID is obtained from the identity context information in the message descriptor of the original inquiry message.

Credit Check sample

When the query programs (CSQ4CVB3/4/5) send reply messages, they use the MQPMO_PASS_IDENTITY_CONTEXT option. This means that the identity context information of the message being put is copied from the identity context of the original inquiry message. With this option, origin context information is generated by the queue manager.

Note: The user ID associated with the MVB3/4/5 transactions requires access to the B2.REPLY.n queues. These user IDs may not be the same as those associated with the request being processed. To get around this possible security exposure, the query programs could use the MQPMO_ALTERNATE_USER_AUTHORITY option when putting their replies. This would mean that each individual user of MVB1 needs authority to open the B2.REPLY.n queues.

Use of message and correlation identifiers in the CAM

The application has to monitor the progress of all the “live” inquiries it is processing at any one time. To do this it uses the unique message identifier of each loan request message to associate all the information it has about each inquiry.

The CAM copies the *MsgId* of the inquiry message into the *CorrelId* of all the request messages it sends for that inquiry. The other programs in the sample (CSQ4CVB3 - 5) copy the *CorrelId* of each message they receive into the *CorrelId* of their reply message.

The Credit Check sample with multiple queue managers

You can use the Credit Check sample application to demonstrate distributed queuing by installing the sample on two queue managers and CICS systems (with each queue manager connected to a different CICS system). When the sample program is installed, and the trigger monitor (CKTI) is running on each system, you need to:

1. Set up the communication link between the two queue managers. For information on how to do this, see the *MQSeries Intercommunication* book.
2. On one queue manager, create a local definition for each of the remote queues (on the other queue manager) that you want to use. These queues can be any of CSQ4SAMP.B*n*.MESSAGES, where *n* is 3, 5, 6, or 7. (These are the queues that are served by the checking-account program and the agency-query program.) For information on how to do this, see the *MQSeries MQSC Command Reference* manual.
3. Change the definition of the namelist (CSQ4SAMP.B4.NAMELIST) so that it contains the names of the remote queues you choose to use. For information on how to do this, see the *MQSeries MQSC Command Reference* manual.

The IMS extension to the Credit Check sample

A version of the checking-account program is supplied as an IMS batch message processing (BMP) program. It is written in the C language.

The program performs the same function as the CICS version, except that to obtain the account information, the program reads an IMS database instead of a VSAM file. If you replace the CICS version of the checking-account program with the IMS version, you see no difference in the method of using the application.

To prepare and run the IMS version you must:

1. Follow the steps in “Preparing and running the Credit Check sample” on page 452.

2. Follow the steps in “Preparing the sample application for the IMS environment” on page 431.
3. Alter the definition of the alias queue CSQ4SAMP.B2.OUTPUT.ALIAS to resolve to queue CSQ4SAMP.B3.IMS.MESSAGES (instead of CSQ4SAMP.B3.MESSAGES). To do this, you can use any one of:
 - The MQSeries for OS/390 operations and control panels
 - The ALTER QALIAS command (described in the *MQSeries MQSC Command Reference* manual)

Another way of using the IMS checking-account program is to make it serve one of the queues that receives messages from the distribution program. In the delivered form of the Credit Check sample application, there are three of these queues (B5/6/7.MESSAGES), all served by the agency-query program. This program searches a VSAM data set. To compare the use of the VSAM data set and the IMS database, you could make the IMS checking-account program serve one of these queues instead. To do this, you must alter the definition of the namelist CSQ4SAMP.B4.NAMELIST to replace one of the CSQ4SAMP.Bn.MESSAGES queues with the CSQ4SAMP.B3.IMS.MESSAGES queue. You can use any one of:

- The MQSeries for OS/390 operations and control panels
- The ALTER NAMELIST command (described in the *MQSeries MQSC Command Reference* manual)

You can then run the sample from CICS transaction MVB1 as usual. The user sees no difference in operation or response. The IMS BMP stops either after receiving a stop message or after being inactive for five minutes.

Design of the IMS checking-account program (CSQ4ICB3)

This program runs as a BMP. You must start the program using its JCL before any MQSeries messages are sent to it.

The program searches an IMS database for the account number in the loan request messages. It retrieves the corresponding account name, average balance, and credit worthiness index.

The program sends the results of the database search to the reply-to queue named in the MQSeries message being processed. The message returned appends the account type and the results of the search to the message received so that the transaction building the response can confirm that the correct query is being processed. The message is in the form of three 79-character groups, as follows:

```
'Response from CHECKING ACCOUNT for name : JONES J B'
'      Opened 870530, 3-month average balance = 000012.57'
'      Credit worthiness index - BBB'
```

When running as a message-oriented BMP, the program drains the IMS message queue, then reads messages from the MQSeries for OS/390 queue and processes them. No information is received from the IMS message queue. The program reconnects to the queue manager after each checkpoint because the handles have been closed.

When running in a batch-oriented BMP, the program continues to be connected to the queue manager after each checkpoint because the handles are not closed.

The Message Handler sample

The Message Handler sample TSO application allows you to browse, forward, and delete messages on a queue. The sample is available in C and COBOL.

Preparing and running the sample

Follow these steps:

1. Prepare the sample as described in “Preparing sample applications for the TSO environment” on page 427.
2. Tailor the CLIST (CSQ4RCH1) provided for the sample to define: - The location of the panels - The location of the message file - The location of the load modules

CLIST CSQ4RCH1 may be used to run both the C and the COBOL version of the sample. The supplied version of CSQ4RCH1 runs the C version, and contains instructions on the tailoring necessary for the COBOL version.

Notes:

1. There are no sample queue definitions provided with the sample.
2. VS COBOL II does not support multitasking with ISPF, so you should not use the Message Handler sample application on both sides of a split screen. If you do, the results are unpredictable.

Using the sample

Having installed the sample and invoked it from the tailored CLIST CSQ4RCH1, the screen shown in Figure 49 is displayed.

```
----- MQSeries for OS/390 -- Samples -----  
COMMAND ===>                                     User Id : JOHNJ  
  
Enter information. Press ENTER :  
  
Queue Manager Name   : _____ :  
Queue Name           : _____ :  
  
F1=HELP   F2=SPLIT  F3=END    F4=RETURN  F5=RFIND   F6=RCHANGE  
F7=UP     F8=DOWN   F9=SWAP   F10=LEFT  F11=RIGHT  F12=RETRIEVE
```

Figure 49. Initial screen for Message Handler sample

Enter the Queue Manager and Queue name to be viewed (case sensitive) and the message list screen is displayed (see Figure 50 on page 465).

Message Handler sample

```
----- MQSeries for OS/390 -- Samples ----- Row 1 to 4 of 4
COMMAND ==>

Queue Manager   : VM03
Queue           : MQEI.IMS.BRIDGE.QUEUE

Message number  01 of 04

Msg  Put Date  Put Time  Format   User      Put Application
No   MM/DD/YYYY HH:MM:SS  Name     Identifier Type       Name
01   10/16/1998 13:51:19 MQIMS    NTSFV02   00000002 NTSFV02A
02   10/16/1998 13:55:45 MQIMS    JOHNJ     00000011 EDIT\CLASSES\BIN\PROGTS
03   10/16/1998 13:54:01 MQIMS    NTSFV02   00000002 NTSFV02B
04   10/16/1998 13:57:22 MQIMS    johnj     00000011 EDIT\CLASSES\BIN\PROGTS
***** Bottom of data *****
```

Figure 50. Message list screen for Message Handler sample

This screen shows the first 99 messages on the queue and, for each, shows the following fields:

Msg No

Message number

Put Date MM/DD/YYYY

Date the message was put on the queue (GMT)

Put Time HH:MM:SS

Time the message was put on the queue (GMT)

Format Name

MQMD.Format field

User Identifier

MQMD.UserIdentifier field

Put Application Type

MQMD.PutApplType field

Put Application Name

MQMD.PutApplName field

The total number of messages on the queue is also displayed.

From this screen a message can be chosen, by number not by cursor position, and then displayed. For an example, see Figure 51 on page 466.

Message Handler sample

```
----- MQSeries for OS/390 -- Samples ----- Row 1 to 35 of 35
COMMAND ==>

Queue Manager   : VM03
Queue           : MQEI.IMS.BRIDGE.QUEUE
Forward to Q Mgr : VM03
Forward to Queue : QL.TEST.ISCRES1

Action : _ : (D)elete (F)orward

Message Content :
-----
Message Descriptor
  StrucId       : 'MD '
  Version       : 00000001
  Report        : 00000000
  MsgType       : 00000001
  Expiry        : -00000001
  Feedback      : 00000000
  Encoding      : 00000785
  CodedCharSetId : 00000500
  Format         : 'MQIMS '
  Priority       : 00000000
  Persistence   : 00000001
  MsgId         : 'C3E2D840E5D4F0F340404040404040AF6B30F0A89B7605'X
  CorrelId      : '000000000000000000000000000000000000000000000000'X
  BackoutCount  : 00000000
  ReplyToQ      : 'QL.TEST.ISCRES1'
  ReplyToQMgr   : 'VM03'
  UserIdentifier : 'NTSFV02'
  AccountingToken :
    '06F2F5F5F3F0F100000000000000000000000000000000000000000000000'X
  ApplIdentityData :
  PutApplType   : 00000002
  PutApplName   : 'NTSFV02A'
  PutDate       : '19971016'
  PutTime       : '13511903'
  ApplOriginData :

Message Buffer : 108 byte(s)
00000000 : C9C9 C840 0000 0001 0000 0054 0000 0311 'IIH ...../'
00000010 : 0000 0000 4040 4040 4040 4040 0000 0000 '.....  .../'
00000020 : 4040 4040 4040 4040 4040 4040 4040 4040 '.....  .../'
00000030 : 4040 4040 4040 4040 4040 4040 4040 4040 '.....  .../'
00000040 : 0000 0000 0000 0000 0000 0000 0000 0000 '.....  .../'
00000050 : 40F1 C300 0018 0000 C9C1 D7D4 C4C9 F2F8 '1C.....IAPMDI28'
00000060 : 40C8 C5D3 D3D6 40E6 D6D9 D3C4 'HELLO WORLD'
***** Bottom of data *****
```

Figure 51. Chosen message is displayed

Once the message has been displayed it can be deleted, left on the queue, or forwarded to another queue. The Forward to Q Mgr and Forward to Queue fields are initialized with values from the MQMD, these can be changed prior to forwarding the message.

The sample design will only allow messages with unique MsgId / CorrelId combinations to be selected and displayed, this is because the message is retrieved using the MsgId and CorrelId as the key. If the key is not unique the sample cannot retrieve the chosen message with certainty.

Design of the sample

This section describes the design of each of the programs that comprise the Message Handler sample application.

Object validation program

This requests a valid queue and queue manager name. If you do not specify a queue manager name, the default queue manager is used, if available. Only local queues can be used; an MQINQ is issued to check the queue type and an error is reported if the queue is not local. If the queue is not opened successfully, or the MQGET call is inhibited on the queue, error messages are returned indicating the CompCode and Reason return code.

Message list program

This displays a list of messages on a queue with information about them such as the putdate, puttime and the message format. The maximum number of messages stored in the list is 99. If there are more messages on the queue than this, the current queue depth is also displayed. To choose a message for display, type the message number into the entry field (the default is 01). If your entry is invalid, you will receive an appropriate error message.

Message content program

This displays message content. The content is formatted and split into two parts:

1. the message descriptor
2. the message buffer

The message descriptor shows the contents of each field on a separate line.

The message buffer is formatted depending on its contents. If the buffer holds a dead letter header (MQDLH) or a transmission queue header (MQXQH), these are formatted and displayed before the buffer itself.

Before the buffer data is formatted, a title line shows the buffer length of the message in bytes. The maximum buffer size is 32768 bytes, and any message longer than this is truncated. The full size of the buffer is displayed along with a message indicating that only the first 32768 bytes of the message are displayed.

The buffer data is formatted in two ways:

1. After the offset into the buffer is printed, the buffer data is displayed in HEX.
2. The buffer data is then displayed again as EBCDIC values. If any EBCDIC value cannot be printed, it prints a '.' instead.

You may enter 'D' for delete, or 'F' for forward into the action field. If you choose to forward the message, the *forward-to queue* and *queue manager name* must be filled in appropriately. The defaults for these fields are read from the message descriptor ReplyToQ and ReplyToQMgr fields.

If you forward a message, any header block stored in the buffer is stripped. If the message is forwarded successfully, it is removed from the original queue. If you enter invalid actions, error messages are displayed.

An example help panel is also available called CSQ4CHP9.

Part 5. Appendixes

Appendix A. Language compilers and assemblers

Table 48 lists the language compilers and assemblers supported.

Table 48. Language compilers and assemblers

Platform	Language	Compiler/Assembler
MQSeries for AIX	C++	IBM C Set++ for AIX, V3.1 IBM C++ compiler, V3.6.4 (for AIX V4.3)
	C	IBM C for AIX, V3.1.4 IBM C Set++ for AIX, V3.1 (C bindings only) IBM C++ compiler, V3.6.4 (for AIX V4.3)
	COBOL	IBM COBOL Set for AIX, V1.1 Micro Focus COBOL Compiler for UNIX, V4.0
	PL/I	IBM PL/I Set for AIX, V1.1
MQSeries for AS/400	C++	IBM ILE C++ for AS/400 (program 5799-GDW) IBM VisualAge for C++ for AS/400 compiler (program 5769-CX4)
	C	IBM ILE C for AS/400, V4R4M0
	COBOL	IBM ILE COBOL for AS/400, V4R4M0
	RPG	IBM ILE RPG for AS/400, V4R4M0
	Java	AS/400 Developer Kit for Java (program 5769-JV1)
MQSeries for AT&T GIS UNIX	C++	AT&T C++ language system for AT&T GIS UNIX
	C	AT&T GIS High Performance C, V1.0b
MQSeries for Compaq (DIGITAL) OpenVMS	C++	DEC C++, V5.0 (VAX), V5.2 (AXP)
	C	DEC C, V5.0
	COBOL	DEC COBOL, V5.0 (VAX), V2.2 (AXP)
MQSeries for Compaq Tru64 UNIX	C++	Compaq C++ for Tru64 UNIX Version 6.2
	C	Compaq C for Tru64 UNIX
	COBOL	Micro Focus COBOL for UNIX Version 4.1B Micro Focus COBOL for UNIX Version 4.1.00G
	Java	Java Development Kit for Compaq Tru64 UNIX, Version 1.1.8
MQSeries for HP-UX	C++	ANSI C++ for HP-UX V10 and V11 HP C++, V3.1 for HP-UX V10.x IBM C++ compiler, V3.6
	C	ANSI C++ for HP-UX V10 and V11 C bundled compiler C Softbench, V5.0 HP C++, V3.1 for HP-UX V10.x HP-UX ANSI C compiler IBM C compiler, V3.6
	COBOL	COBOL Softbench, V4.0 Micro Focus COBOL compiler, V4.0 for UNIX

Compilers and assemblers

Table 48. Language compilers and assemblers (continued)

Platform	Language	Compiler/Assembler
MQSeries for OS/2 Warp	C++	IBM C++ compiler, V3.6 IBM VisualAge for C++ for OS/2, V3.0
	C	Borland C++, V2 (C bindings only) IBM C compiler, V3.6 IBM VisualAge for C++ for OS/2, V3.0 (C bindings only)
	COBOL	IBM VisualAge for COBOL for OS/2, V1.1 Micro Focus COBOL, V4.0
	PL/I	IBM PL/I for OS/2, V1.2 IBM VisualAge for PL/I for OS/2
MQSeries for OS/390	Assembler	Assembler H assembler IBM High Level Assembler/MVS assembler
	C++	IBM OS/390 C/C++, V2R4
	C	C/370, Release 2.1.0 IBM OS/390 C/C++, V2R4 IBM SAA AD/Cycle [®] C/370
	COBOL	IBM SAA AD/Cycle COBOL/370 VS COBOL II COBOL for OS/390 & VM
	PL/I	IBM SAA AD/Cycle PL/I Compiler OS PL/I Optimizing compiler
MQSeries for SINIX and DC/OSx	C	DC/OSx: C4.0 compiler, V4.0.1 SINIX: C compiler (C-DS, MIPS), V1.1
	COBOL	Micro Focus COBOL, V3.2
MQSeries for Sun Solaris	C++	SunWorkShop compiler C++, V4.2
	C	SunWorkShop compiler C, V4.2
	COBOL	Micro Focus COBOL Compiler, V4.0 for UNIX
MQSeries for Tandem NSK	C	D30 or later using WIDE memory model (32-bit integers)
	COBOL	D30 or later
	TAL	D30 or later
MQSeries for VSE/ESA	C	IBM C for VSE/ESA, V1.1
	COBOL	IBM COBOL for VSE/ESA, V1.1
	PL/I	IBM PL/I for VSE/ESA, V1.1
MQSeries for Windows, V2.0	16-bit Basic	Microsoft Visual Basic, V3.0 or V4.0
	32-bit Basic	Microsoft Visual Basic, V4.0
	16-bit C	Microsoft Visual C++, V1.5
	32-bit C	Microsoft Visual C++, V2.0
MQSeries for Windows, V2.1	Basic	Microsoft Visual Basic, V4.0
	C	Microsoft Visual C++, V4.0 Borland C

Table 48. Language compilers and assemblers (continued)

Platform	Language	Compiler/Assembler
MQSeries for Windows NT	Basic	Microsoft Visual Basic for Windows, V4.0 (16-bit) Microsoft Visual Basic for Windows, V5.0 (32-bit)
	C++	IBM C++ compiler, V3.6.4 IBM VisualAge for C++ for Windows, V3.5 IBM VisualAge for C++ Professional, V4.0 Microsoft Visual C++ for Windows 95 and NT, V4.0 & V5.0
	C	IBM C compiler, V3.6.4 IBM VisualAge for C++ for Windows, V3.5 Microsoft Visual C++ for Windows 95 and NT, V4.0 & V5.0
	COBOL	IBM VisualAge COBOL Enterprise, V2.2 IBM VisualAge COBOL for Windows NT, V2.1 Micro Focus Object COBOL for Windows NT, V3.3 or V4.0
	Java	IBM VisualAge e-business for Windows, V1.0.1 IBM VisualAge for Java Enterprise, V2.0 IBM VisualAge for Java Professional, V2.0
	PL/I	IBM PL/I for Windows, V1.2 IBM VisualAge for PL/I for Windows IBM VisualAge PL/I Enterprise, V2.1
	DOS clients	C
VM/ESA clients	Assembler	IBM Assembler
	C	IBM C for VM Release, 3.1
	COBOL	IBM VS COBOL II
	PL/I	IBM OS/PL/I, Release 2.3
	REXX	IBM VM/ESA REXX/VM
Windows 3.1 clients	C++	Microsoft Visual C++, V1.5
	C	Microsoft C, V7.0
Windows 95 and Windows 98 clients	C++	IBM VisualAge for C++ for Windows, V3.5 Microsoft Visual C++, V4.0
	C	Microsoft Visual C++, V4.0
	COBOL	Micro Focus COBOL Workbench, V4.0
Note: RPG bindings are shown for the IBM SAA AD/Cycle RPG/400 compiler.		

Compilers and assemblers

Appendix B. C language examples

The extracts in this appendix are mostly taken from the MQSeries for OS/390 sample applications. They are applicable to all platforms, except where noted.

The examples in this appendix demonstrate the following techniques:

- “Connecting to a queue manager” on page 476
- “Disconnecting from a queue manager” on page 476
- “Creating a dynamic queue” on page 477
- “Opening an existing queue” on page 478
- “Closing a queue” on page 479
- “Putting a message using MQPUT” on page 479
- “Putting a message using MQPUT1” on page 480
- “Getting a message” on page 482
- “Getting a message using the wait option” on page 483
- “Getting a message using signaling” on page 484
- “Inquiring about the attributes of an object” on page 486
- “Setting the attributes of a queue” on page 487

Connecting to a queue manager

This example demonstrates how to use the MQCONN call to connect a program to a queue manager in OS/390 batch. This extract is taken from the Browse sample application (program CSQ4BCA1) supplied with MQSeries for OS/390. For the names and locations of the sample applications on other platforms, see “Chapter 34. Sample programs (all platforms except OS/390)” on page 359.

```
#include <cmqc.h>
:
:
static char Parm1[MQ_Q_MGR_NAME_LENGTH] ;
:
:
int main(int argc, char *argv[] )
{
/*                                     */
/*   Variables for MQ calls           */
/*                                     */
MQHCONN Hconn;      /* Connection handle */
MQLONG  CompCode;   /* Completion code   */
MQLONG  Reason;     /* Qualifying reason  */
:
/* Copy the queue manager name, passed in the */
/* parm field, to Parm1                       */
strncpy(Parm1,argv[1],MQ_Q_MGR_NAME_LENGTH);
:
/*                                     */
/* Connect to the specified queue manager.     */
/* Test the output of the connect call. If the */
/* call fails, print an error message showing the */
/* completion code and reason code, then leave the */
/* program.                                     */
/*                                     */
MQCONN(Parm1,
        &Hconn,
        &CompCode,
        &Reason);
if ((CompCode != MQCC_OK) | (Reason != MQRC_NONE))
{
printf(pBuff, MESSAGE_4_E,
        ERROR_IN_MQCONN, CompCode, Reason);
PrintLine(pBuff);
RetCode = CSQ4_ERROR;
goto AbnormalExit2;
}
:
:
}
```

Disconnecting from a queue manager

This example demonstrates how to use the MQDISC call to disconnect a program from a queue manager in OS/390 batch. The variables used in this code extract are those that were set in “Connecting to a queue manager”. This extract is taken from the Browse sample application (program CSQ4BCA1) supplied with MQSeries for OS/390. For the names and locations of the sample applications on other platforms, see “Chapter 34. Sample programs (all platforms except OS/390)” on page 359.

```
:
/*                                     */
/* Disconnect from the queue manager. Test the */
/* output of the disconnect call. If the call  */
/*                                     */
```


C: Disconnecting from a queue manager

```
/* fails, print an error message showing the      */
/* completion code and reason code.              */
/*                                                */
MQDISC(&Hconn,
      &CompCode,
      &Reason);
if ((CompCode != MQCC_OK) || (Reason != MQRC_NONE))
{
    sprintf(pBuff, MESSAGE_4_E,
           ERROR_IN_MQDISC, CompCode, Reason);
    PrintLine(pBuff);
    RetCode = CSQ4_ERROR;
}
:
:
```

Creating a dynamic queue

This example demonstrates how to use the MQOPEN call to create a dynamic queue. This extract is taken from the Mail Manager sample application (program CSQ4TCD1) supplied with MQSeries for OS/390. For the names and locations of the sample applications on other platforms, see “Chapter 34. Sample programs (all platforms except OS/390)” on page 359.

```
:
MQLONG HCONN = 0; /* Connection handle */
MQHOBJ HOBJ; /* MailQ Object handle */
MQHOBJ HobjTempQ; /* TempQ Object Handle */
MQLONG CompCode; /* Completion code */
MQLONG Reason; /* Qualifying reason */
MQOD ObjDesc = {MQOD_DEFAULT};
/* Object descriptor */
MQLONG OpenOptions; /* Options control MQOPEN */
:
/*-----*/
/* Initialize the Object Descriptor (MQOD) */
/* control block. (The remaining fields */
/* are already initialized.) */
/*-----*/
strncpy( ObjDesc.ObjectName,
        SYSTEM_REPLY_MODEL,
        MQ_Q_NAME_LENGTH );
strncpy( ObjDesc.DynamicQName,
        SYSTEM_REPLY_INITIAL,
        MQ_Q_NAME_LENGTH );
OpenOptions = MQOO_INPUT_AS_Q_DEF;
/*-----*/
/* Open the model queue and, therefore, */
/* create and open a temporary dynamic */
/* queue */
/*-----*/
MQOPEN( HCONN,
        &ObjDesc,
        OpenOptions,
        &HobjTempQ,
        &CompCode,
        &Reason );
if ( CompCode == MQCC_OK ) {
:
:
}
else {
/*-----*/
/* Build an error message to report the */
/* failure of the opening of the model */
/* queue */
/*-----*/
}
```

C: Creating a dynamic queue

```
    /* queue */
    /*-----*/
    MQMErrorHandling( "OPEN TEMPQ", CompCode,
                    Reason );
    ErrorFound = TRUE;
}
return ErrorFound;
}
:
```

Opening an existing queue

This example demonstrates how to use the MQOPEN call to open a queue that has already been defined. This extract is taken from the Browse sample application (program CSQ4BCA1) supplied with MQSeries for OS/390. For the names and locations of the sample applications on other platforms, see “Chapter 34. Sample programs (all platforms except OS/390)” on page 359.

```
#include <cmqc.h>
:
static char Parm1[MQ_Q_MGR_NAME_LENGTH];
:
int main(int argc, char *argv[] )
{
/*
/*   Variables for MQ calls */
/*
MQHCONN Hconn ;          /* Connection handle */
MQLONG  CompCode;        /* Completion code */
MQLONG  Reason;          /* Qualifying reason */
MQOD    ObjDesc = { MQOD_DEFAULT };
MQLONG  OpenOptions;     /* Options that control */
/* the MQOPEN call */
MQHOBJ  Hobj;            /* Object handle */
:
/* Copy the queue name, passed in the parm field, */
/* to Parm2 strncpy(Parm2,argv[2], */
/* MQ_Q_NAME_LENGTH); */
:
/* */
/* Initialize the object descriptor (MQOD) control */
/* block. (The initialization default sets StrucId, */
/* Version, ObjectType, ObjectQMgrName, */
/* DynamicQName, and AlternateUserid fields) */
/* */
strncpy(ObjDesc.ObjectName,Parm2,MQ_Q_NAME_LENGTH);
:
/* Initialize the other fields required for the open */
/* call (Hobj is set by the MQCONN call). */
/* */
OpenOptions = MQOO_BROWSE;
:
/* */
/* Open the queue. */
/* Test the output of the open call. If the call */
/* fails, print an error message showing the */
/* completion code and reason code, then bypass */
/* processing, disconnect and leave the program. */
/* */
MQOPEN(Hconn,
        &ObjDesc,
```

```

        OpenOptions,
        &Hobj,
        &CompCode,
        &Reason);
if ((CompCode != MQCC_OK) || (Reason != MQRC_NONE))
{
    sprintf(pBuff, MESSAGE_4_E,
            ERROR_IN_MQOPEN, CompCode, Reason);
    PrintLine(pBuff);
    RetCode = CSQ4_ERROR;
    goto AbnormalExit1;    /* disconnect processing */
}
:
} /* end of main */

```

Closing a queue

This example demonstrates how to use the MQCLOSE call to close a queue. This extract is taken from the Browse sample application (program CSQ4BCA1) supplied with MQSeries for OS/390. For the names and locations of the sample applications on other platforms, see “Chapter 34. Sample programs (all platforms except OS/390)” on page 359.

```

:
/*                               */
/* Close the queue.              */
/* Test the output of the close call. If the call */
/* fails, print an error message showing the */
/* completion code and reason code.          */
/*                               */
MQCLOSE(Hconn,
        &Hobj,
        MQCO_NONE,
        &CompCode,
        &Reason);
if ((CompCode != MQCC_OK) || (Reason != MQRC_NONE))
{
    sprintf(pBuff, MESSAGE_4_E,
            ERROR_IN_MQCLOSE, CompCode, Reason);
    PrintLine(pBuff);
    RetCode = CSQ4_ERROR;
}
:

```

Putting a message using MQPUT

This example demonstrates how to use the MQPUT call to put a message on a queue. This extract is not taken from the sample applications supplied with MQSeries. For the names and locations of the sample applications, see “Chapter 34. Sample programs (all platforms except OS/390)” on page 359 and “Chapter 35. Sample programs for MQSeries for OS/390” on page 421.

```

:
qput()
{
    MQMD    MsgDesc;
    MQPMO   PutMsgOpts;
    MQLONG  CompCode;
    MQLONG  Reason;
    MQHCONN Hconn;

```

C: Putting using MQPUT

```
MQHOBJ Hobj;
char message_buffer[] = "MY MESSAGE";
/*-----*/
/* Set up PMO structure. */
/*-----*/
memset(&PutMsgOpts, '\0', sizeof(PutMsgOpts));
memcpy(PutMsgOpts.StrucId, MQPMO_STRUC_ID,
       sizeof(PutMsgOpts.StrucId));
PutMsgOpts.Version = MQPMO_VERSION_1;
PutMsgOpts.Options = MQPMO_SYNCPOINT;

/*-----*/
/* Set up MD structure. */
/*-----*/
memset(&MsgDesc, '\0', sizeof(MsgDesc));
memcpy(MsgDesc.StrucId, MQMD_STRUC_ID,
       sizeof(MsgDesc.StrucId));
MsgDesc.Version = MQMD_VERSION_1;
MsgDesc.Expiry = MQEI_UNLIMITED;
MsgDesc.Report = MQRO_NONE;
MsgDesc.MsgType = MQMT_DATAGRAM;
MsgDesc.Priority = 1;
MsgDesc.Persistence = MQPER_PERSISTENT;
memset(MsgDesc.ReplyToQ,
       '\0',
       sizeof(MsgDesc.ReplyToQ));

/*-----*/
/* Put the message. */
/*-----*/
MQPUT(Hconn, Hobj, &MsgDesc, &PutMsgOpts,
      sizeof(message_buffer), message_buffer,
      &CompCode, &Reason);

/*-----*/
/* Check completion and reason codes. */
/*-----*/
switch (CompCode)
{
    case MQCC_OK:
        break;
    case MQCC_FAILED:
        switch (Reason)
        {
            case MQRC_Q_FULL:
            case MQRC_MSG_TOO_BIG_FOR_Q:
                break;
            default:
                break; /* Perform error processing */
        }
        break;
    default:
        break; /* Perform error processing */
}
}
```

Putting a message using MQPUT1

This example demonstrates how to use the MQPUT1 call to open a queue, put a single message on the queue, then close the queue. This extract is taken from the Credit Check sample application (program CSQ4CCB5) supplied with MQSeries for OS/390. For the names and locations of the sample applications on other platforms, see “Chapter 34. Sample programs (all platforms except OS/390)” on page 359.

```

:
MQLONG Hconn;          /* Connection handle      */
MQHOBJ  Hobj_CheckQ;  /* Object handle         */
MQLONG  CompCode;     /* Completion code       */
MQLONG  Reason;       /* Qualifying reason     */
MQOD    ObjDesc = {MQOD_DEFAULT};
                /* Object descriptor     */
MQMD    MsgDesc = {MQMD_DEFAULT};
                /* Message descriptor   */
MQLONG  OpenOptions; /* Control the MQOPEN call */

MQGMO   GetMsgOpts = {MQGMO_DEFAULT};
                /* Get Message Options  */
MQLONG  MsgBuffLen; /* Length of message buffer */
CSQ4BCAQ MsgBuffer; /* Message structure     */
MQLONG  DataLen;   /* Length of message     */

MQPMO   PutMsgOpts = {MQPMO_DEFAULT};
                /* Put Message Options  */
CSQ4BQRM PutBuffer; /* Message structure     */
MQLONG  PutBuffLen = sizeof(PutBuffer);
                /* Length of message buffer */
:
void Process_Query(void)
{
    /*
    /* Build the reply message
    /*
    :
    /*
    /* Set the object descriptor, message descriptor and
    /* put message options to the values required to
    /* create the reply message.
    /*
    strncpy(ObjDesc.ObjectName, MsgDesc.ReplyToQ,
            MQ_Q_NAME_LENGTH);
    strncpy(ObjDesc.ObjectQMgrName, MsgDesc.ReplyToQMgr,
            MQ_Q_MGR_NAME_LENGTH);
    MsgDesc.MsgType = MQMT_REPLY;
    MsgDesc.Report = MQRO_NONE;
    memset(MsgDesc.ReplyToQ, ' ', MQ_Q_NAME_LENGTH);
    memset(MsgDesc.ReplyToQMgr, ' ', MQ_Q_MGR_NAME_LENGTH);
    memcpy(MsgDesc.MsgId, MQMI_NONE, sizeof(MsgDesc.MsgId));
    PutMsgOpts.Options = MQPMO_SYNCPOINT +
                        MQPMO_PASS_IDENTITY_CONTEXT;
    PutMsgOpts.Context = Hobj_CheckQ;
    PutBuffLen = sizeof(PutBuffer);
    MQPUT1(Hconn,
           &ObjDesc,
           &MsgDesc,
           &PutMsgOpts,
           PutBuffLen,
           &PutBuffer,
           &CompCode,
           &Reason);

    if (CompCode != MQCC_OK)
    {
        strncpy(TS_Operation, "MQPUT1",
                sizeof(TS_Operation));
        strncpy(TS_ObjName, ObjDesc.ObjectName,
                MQ_Q_NAME_LENGTH);
        Record_Call_Error();
        Forward_Msg_To_DLQ();
    }
}

```

C: Putting using MQPUT1

```
    }  
    return;  
  }  
  :  
  :
```

Getting a message

This example demonstrates how to use the MQGET call to remove a message from a queue. This extract is taken from the Browse sample application (program CSQ4BCA1) supplied with MQSeries for OS/390. For the names and locations of the sample applications on other platforms, see "Chapter 34. Sample programs (all platforms except OS/390)" on page 359.

```
#include "cmqc.h"  
:  
#define BUFFERLENGTH 80  
:  
int main(int argc, char *argv[] )  
{  
  /*                                     */  
  /*   Variables for MQ calls           */  
  /*                                     */  
  MQHCONN Hconn ;           /* Connection handle   */  
  MQLONG  CompCode;         /* Completion code   */  
  MQLONG  Reason;          /* Qualifying reason */  
  MQHOBJ  Hobj;            /* Object handle     */  
  MQMD    MsgDesc = { MQMD_DEFAULT }; /* Message descriptor */  
  MQLONG  DataLength ;     /* Length of the message */  
  MQCHAR  Buffer[BUFFERLENGTH+1]; /* Area for message data */  
  MQGMO   GetMsgOpts = { MQGMO_DEFAULT }; /* Options which control */  
  MQLONG  BufferLength = BUFFERLENGTH ; /* the MQGET call       */  
  MQLONG  BufferLength = BUFFERLENGTH ; /* Length of buffer     */  
  :  
  /*   No need to change the message descriptor */  
  /*   (MQMD) control block because initialization */  
  /*   default sets all the fields.             */  
  /*                                     */  
  /*   Initialize the get message options (MQGMO) */  
  /*   control block (the copy file initializes all */  
  /*   the other fields).                       */  
  /*                                     */  
  GetMsgOpts.Options = MQGMO_NO_WAIT      +  
                       MQGMO_BROWSE_FIRST +  
                       MQGMO_ACCEPT_TRUNCATED_MSG;  
  /*                                     */  
  /* Get the first message.                 */  
  /* Test for the output of the call is carried out */  
  /* in the 'for' loop.                     */  
  /*                                     */  
  MQGET(Hconn,  
        Hobj,  
        &MsgDesc,  
        &GetMsgOpts,  
        BufferLength,  
        Buffer,  
        &DataLength,  
        &CompCode,  
        &Reason);
```

```

/*                                     */
/* Process the message and get the next message, */
/* until no messages remaining.                */
:
/* If the call fails for any other reason,      */
/* print an error message showing the completion */
/* code and reason code.                       */
/*                                     */
if ( (CompCode == MQCC_FAILED) &&
      (Reason == MQRC_NO_MSG_AVAILABLE) )
{
:
}
else
{
    sprintf(pBuff, MESSAGE_4_E,
            ERROR_IN_MQGET, CompCode, Reason);
    PrintLine(pBuff);
    RetCode = CSQ4_ERROR;
}
:
} /* end of main */

```

Getting a message using the wait option

This example demonstrates how to use the wait option of the MQGET call. This code accepts truncated messages. This extract is taken from the Credit Check sample application (program CSQ4CCB5) supplied with MQSeries for OS/390. For the names and locations of the sample applications on other platforms, see “Chapter 34. Sample programs (all platforms except OS/390)” on page 359.

```

:
MQLONG  Hconn;           /* Connection handle */
MQHOBJ  Hobj_CheckQ;    /* Object handle     */
MQLONG  CompCode;       /* Completion code   */
MQLONG  Reason;         /* Qualifying reason */
MQOD    ObjDesc = {MQOD_DEFAULT};
                /* Object descriptor */
MQMD    MsgDesc = {MQMD_DEFAULT};
                /* Message descriptor */
MQLONG  OpenOptions;    /* Control the MQOPEN call */
MQGMO   GetMsgOpts = {MQGMO_DEFAULT};
                /* Get Message Options */
MQLONG  MsgBuffLen;     /* Length of message buffer */
CSQ4BCAQ MsgBuffer;     /* Message structure  */
MQLONG  DataLen;        /* Length of message  */
:
void main(void)
{
:
/*                                     */
/* Initialize options and open the queue for input */
/*                                     */
:
/*                                     */
/* Get and process messages */
/*                                     */
/*                                     */
    GetMsgOpts.Options = MQGMO_WAIT +
                        MQGMO_ACCEPT_TRUNCATED_MSG +
                        MQGMO_SYNCPOINT;
    GetMsgOpts.WaitInterval = WAIT_INTERVAL;

```

C: Get with wait option

```
MsgBuffLen = sizeof(MsgBuffer);
memcpy(MsgDesc.MsgId, MQMI_NONE,
        sizeof(MsgDesc.MsgId));
memcpy(MsgDesc.CorrelId, MQCI_NONE,
        sizeof(MsgDesc.CorrelId));
/*
/* Make the first MQGET call outside the loop
/*
MQGET(Hconn,
      Hobj_CheckQ,
      &MsgDesc,
      &GetMsgOpts,
      MsgBuffLen,
      &MsgBuffer,
      &DataLen,
      &CompCode,
      &Reason);
:
/*
/* Test the output of the MQGET call. If the call
/* failed, send an error message showing the
/* completion code and reason code, unless the
/* reason code is NO_MSG_AVAILABLE.
/*
if (Reason != MQRC_NO_MSG_AVAILABLE)
{
  strncpy(TS_Operation, "MQGET", sizeof(TS_Operation));
  strncpy(TS_ObjName, ObjDesc.ObjectName,
          MQ_Q_NAME_LENGTH);
  Record_Call_Error();
}
:
```

Getting a message using signaling

Signaling is available only with MQSeries for OS/390 and MQSeries for Windows V2.1.

This example demonstrates how to use the MQGET call to set a signal so that you are notified when a suitable message arrives on a queue. This extract is not taken from the sample applications supplied with MQSeries.

```
:
get_set_signal()
{
  MQMD   MsgDesc;
  MQGMO  GetMsgOpts;
  MQLONG CompCode;
  MQLONG Reason;
  MQHCONN Hconn;
  MQHOBJ  Hobj;
  MQLONG  BufferLength;
  MQLONG  DataLength;
  char message_buffer[100];
  long int q_ecn, work_ecn;
  short int signal_sw, endloop;
  long int mask = 255;

  /*-----*/
  /* Set up GMO structure.
  /*-----*/
  memset(&GetMsgOpts, '\0', sizeof(GetMsgOpts));
  memcpy(GetMsgOpts.StrucId, MQGMO_STRUC_ID,
         sizeof(GetMsgOpts.StrucId));
  GetMsgOpts.Version = MQGMO_VERSION_1;
```



```

GetMsgOpts.WaitInterval = 1000;
GetMsgOpts.Options      = MQGMO_SET_SIGNAL +
                          MQGMO_BROWSE_FIRST;

q_ecn                    = 0;
GetMsgOpts.Signal1     = &q_ecn;
/*-----*/
/* Set up MD structure. */
/*-----*/
memset(&MsgDesc,'\0',sizeof(MsgDesc));
memcpy(MsgDesc.StrucId, MQMD_STRUC_ID,
        sizeof(MsgDesc.StrucId));
MsgDesc.Version = MQMD_VERSION_1;
MsgDesc.Report  = MQRO_NONE;
memcpy(MsgDesc.MsgId,MQMI_NONE,
        sizeof(MsgDesc.MsgId));
memcpy(MsgDesc.CorrelId,MQCI_NONE,
        sizeof(MsgDesc.CorrelId));

/*-----*/
/* Issue the MQGET call. */
/*-----*/
BufferLength = sizeof(message_buffer);
signal_sw = 0;

MQGET(Hconn, Hobj, &MsgDesc, &GetMsgOpts,
      BufferLength, message_buffer, &DataLength,
      &CompCode, &Reason);
/*-----*/
/* Check completion and reason codes. */
/*-----*/
switch (CompCode)
{
    case (MQCC_OK):          /* Message retrieved */
        break;
    case (MQCC_WARNING):
        switch (Reason)
        {
            case (MQRC_SIGNAL_REQUEST_ACCEPTED):
                signal_sw = 1;
                break;
            default:
                break; /* Perform error processing */
        }
        break;
    case (MQCC_FAILED):
        switch (Reason)
        {
            case (MQRC_Q_MGR_NOT_AVAILABLE):
            case (MQRC_CONNECTION_BROKEN):
            case (MQRC_Q_MGR_STOPPING):
                break;
            default:
                break; /* Perform error processing. */
        }
        break;
    default:
        break; /* Perform error processing. */
}
/*-----*/
/* If the SET_SIGNAL was accepted, set up a loop to */
/* check whether a message has arrived at one second */
/* intervals. The loop ends if a message arrives or */
/* the wait interval specified in the MQGMO */
/* structure has expired. */
/* */
/* If a message arrives on the queue, another MQGET */
/* must be issued to retrieve the message. If other */
/* MQM calls have been made in the intervening */

```

C: Getting with signaling

```
/* period, this may necessitate reinitializing the */
/* MQMD and MQGMO structures. */
/* In this code, no intervening calls */
/* have been made, so the only change required to */
/* the structures is to specify MQGMO_NO_WAIT, */
/* since we now know the message is there. */
/* */
/* This code uses the EXEC CICS DELAY command to */
/* suspend the program for a second. A batch program */
/* may achieve the same effect by calling an */
/* assembler language subroutine which issues an */
/* OS/390 STIMER macro. */
/*-----*/
if (signal_sw == 1)
{
    endloop = 0;
    do
    {
        EXEC CICS DELAY FOR HOURS(0) MINUTES(0) SECONDS(1);
        work_ecb = q_ecb & mask;
        switch (work_ecb)
        {
            case (MQEC_MSG_ARRIVED):
                endloop = 1;
                mqgmo_options = MQGMO_NO_WAIT;
                MQGET(Hconn, Hobj, &MsgDesc, &GetMsgOpts,
                    BufferLength, message_buffer,
                    &DataLength, &CompCode, &Reason);
                if (CompCode != MQCC_OK)
                    ; /* Perform error processing. */
                break;
            case (MQEC_WAIT_INTERVAL_EXPIRED):
            case (MQEC_WAIT_CANCELED):
                endloop = 1;
                break;
            default:
                break;
        }
    } while (endloop == 0);
}
return;
}
```

Inquiring about the attributes of an object

This example demonstrates how to use the MQINQ call to inquire about the attributes of a queue. This extract is taken from the Queue Attributes sample application (program CSQ4CCC1) supplied with MQSeries for OS/390. For the names and locations of the sample applications on other platforms, see “Chapter 34. Sample programs (all platforms except OS/390)” on page 359.

```
#include <cmqc.h> /* MQ API header file */
:
:
#define NUMBEROFSELECTORS 2

const MQHCONN Hconn = MQHC_DEF_HCONN;
:
:
static void InquireGetAndPut(char *Message,
                            PMQHOBJ pHobj,
                            char *Object)
{
    /* Declare local variables */
    /* */
    MQLONG SelectorCount = NUMBEROFSELECTORS;
```

C: Inquiring about attributes

```

/* Number of selectors */
MQLONG IntAttrCount = NUMBEROFSELECTORS;
/* Number of int attrs */
MQLONG CharAttrLength = 0;
/* Length of char attribute buffer */
MQCHAR *CharAttrs ;
/* Character attribute buffer */
MQLONG SelectorsTable[NUMBEROFSELECTORS];
/* attribute selectors */
MQLONG IntAttrsTable[NUMBEROFSELECTORS];
/* integer attributes */
MQLONG CompCode;
/* Completion code */
MQLONG Reason;
/* Qualifying reason */
/*
/* Open the queue. If successful, do the inquire */
/* call. */
/*
/*
/* Initialize the variables for the inquire */
/* call: */
/* - Set SelectorsTable to the attributes whose */
/* status is */
/* required */
/* - All other variables are already set */
/*
SelectorsTable[0] = MQIA_INHIBIT_GET;
SelectorsTable[1] = MQIA_INHIBIT_PUT;
/*
/* Issue the inquire call */
/* Test the output of the inquire call. If the */
/* call failed, display an error message */
/* showing the completion code and reason code, */
/* otherwise display the status of the */
/* INHIBIT-GET and INHIBIT-PUT attributes */
/*
MQINQ(Hconn,
      *pHobj,
      SelectorCount,
      SelectorsTable,
      IntAttrCount,
      IntAttrsTable,
      CharAttrLength,
      CharAttrs,
      &CompCode,
      &Reason);
if (CompCode != MQCC_OK)
{
    sprintf(Message, MESSAGE_4_E,
            ERROR_IN_MQINQ, CompCode, Reason);
    SetMsg(Message);
}
else
{
    /* Process the changes */
} /* end if CompCode */
```

Setting the attributes of a queue

This example demonstrates how to use the MQSET call to change the attributes of a queue. This extract is taken from the Queue Attributes sample application (program CSQ4CCC1) supplied with MQSeries for OS/390. For the names and locations of the sample applications on other platforms, see “Chapter 34. Sample programs (all platforms except OS/390)” on page 359.

```
#include <cmqc.h> /* MQ API header file */
```

C: Setting queue attributes

```
:
#define NUMBEROFSELECTORS 2

const MQHCONN Hconn = MQHC_DEF_HCONN;

static void InhibitGetAndPut(char *Message,
                             PMQHOBJ pHobj,
                             char *Object)
{
/*          */
/*      Declare local variables          */
/*          */
MQLONG SelectorCount = NUMBEROFSELECTORS;
/*          Number of selectors          */
MQLONG IntAttrCount = NUMBEROFSELECTORS;
/*          Number of int attrs          */
MQLONG CharAttrLength = 0;
/*          Length of char attribute buffer          */
MQCHAR *CharAttrs ;
/*          Character attribute buffer          */
MQLONG SelectorsTable[NUMBEROFSELECTORS];
/*          attribute selectors          */
MQLONG IntAttrsTable[NUMBEROFSELECTORS];
/*          integer attributes          */
MQLONG CompCode;
/*          Completion code          */
MQLONG Reason;
/*          Qualifying reason          */
:
/*          */
/*      Open the queue.  If successful, do the          */
/*      inquire call.          */
/*          */
:
/*          */
/*      Initialize the variables for the set call:          */
/*      - Set SelectorsTable to the attributes to be          */
/*      set          */
/*      - Set IntAttrsTable to the required status          */
/*      - All other variables are already set          */
/*          */
SelectorsTable[0] = MQIA_INHIBIT_GET;
SelectorsTable[1] = MQIA_INHIBIT_PUT;
IntAttrsTable[0] = MQQA_GET_INHIBITED;
IntAttrsTable[1] = MQQA_PUT_INHIBITED;
:
/*          */
/*      Issue the set call.          */
/*      Test the output of the set call.  If the          */
/*      call fails, display an error message          */
/*      showing the completion code and reason          */
/*      code; otherwise move INHIBITED to the          */
/*      relevant screen map fields          */
/*          */
MQSET(Hconn,
      pHobj,
      SelectorCount,
      SelectorsTable,
      IntAttrCount,
      IntAttrsTable,
      CharAttrLength,
      CharAttrs,
      &CompCode,
      &Reason);
if (CompCode != MQCC_OK)
{
    sprintf(Message, MESSAGE_4_E,
            ERROR_IN_MQSET, CompCode, Reason);
}
```

C: Setting queue attributes

```
    SetMsg(Message);  
  }  
else  
  {  
    /* Process the changes */  
  } /* end if CompCode */
```

Appendix C. COBOL examples

The examples in this appendix are taken from the MQSeries for OS/390 sample applications. They are applicable to all platforms, except where noted.

The examples in this appendix demonstrate the following techniques:

- “Connecting to a queue manager” on page 492
- “Disconnecting from a queue manager” on page 492
- “Creating a dynamic queue” on page 493
- “Opening an existing queue” on page 494
- “Closing a queue” on page 496
- “Putting a message using MQPUT” on page 496
- “Putting a message using MQPUT1” on page 498
- “Getting a message” on page 499
- “Getting a message using the wait option” on page 501
- “Getting a message using signaling” on page 502
- “Inquiring about the attributes of an object” on page 505
- “Setting the attributes of a queue” on page 506

Connecting to a queue manager

This example demonstrates how to use the MQCONN call to connect a program to a queue manager in OS/390 batch. This extract is taken from the Browse sample application (program CSQ4BVA1) supplied with MQSeries for OS/390. For the names and locations of the sample applications on other platforms, see “Chapter 34. Sample programs (all platforms except OS/390)” on page 359.

```
* -----*
WORKING-STORAGE SECTION.
* -----*
*   W02 - Data fields derived from the PARM field
01  W02-MQM          PIC X(48) VALUE SPACES.
*   W03 - MQM API fields
01  W03-HCONN       PIC S9(9) BINARY.
01  W03-COMPCODE    PIC S9(9) BINARY.
01  W03-REASON      PIC S9(9) BINARY.
*
*   MQV contains constants (for filling in the control
*   blocks)
*   and return codes (for testing the result of a call)
*
01  W05-MQM-CONSTANTS.
COPY CMQV SUPPRESS.
   :
*   Separate into the relevant fields any data passed
*   in the PARM statement
*
UNSTRING PARM-STRING DELIMITED BY ALL ','
        INTO W02-MQM
        W02-OBJECT.
   :
*   Connect to the specified queue manager.
*
CALL 'MQCONN' USING W02-MQM
                  W03-HCONN
                  W03-COMPCODE
                  W03-REASON.
*
*   Test the output of the connect call. If the call
*   fails, print an error message showing the
*   completion code and reason code.
*
IF (W03-COMPCODE NOT = MQCC-OK) THEN
   :
   END-IF.
   :
   :
```

Disconnecting from a queue manager

This example demonstrates how to use the MQDISC call to disconnect a program from a queue manager in OS/390 batch. The variables used in this code extract are those that were set in “Connecting to a queue manager”. This extract is taken from the Browse sample application (program CSQ4BVA1) supplied with MQSeries for OS/390. For the names and locations of the sample applications on other platforms, see “Chapter 34. Sample programs (all platforms except OS/390)” on page 359.

```
   :
*
*   Disconnect from the queue manager
```


COBOL: Disconnecting from a queue manager

```
*
*   CALL 'MQDISC' USING W03-HCONN
*                           W03-COMPCODE
*                           W03-REASON.
*
*   Test the output of the disconnect call.  If the
*   call fails, print an error message showing the
*   completion code and reason code.
*
*   IF (W03-COMPCODE NOT = MQCC-OK) THEN
*   :
*       END-IF.
*   :
```

Creating a dynamic queue

This example demonstrates how to use the MQOPEN call to create a dynamic queue. This extract is taken from the Credit Check sample application (program CSQ4CVB1) supplied with MQSeries for OS/390. For the names and locations of the sample applications on other platforms, see “Chapter 34. Sample programs (all platforms except OS/390)” on page 359.

```

:
* -----*
*   WORKING-STORAGE SECTION.
* -----*
*
*   W02 - Queues processed in this program
*
*   01 W02-MODEL-QNAME      PIC X(48) VALUE
*       'CSQ4SAMP.B1.MODEL'
*   01 W02-NAME-PREFIX     PIC X(48) VALUE
*       'CSQ4SAMP.B1.*'
*   01 W02-TEMPORARY-Q     PIC X(48).
*
*   W03 - MQM API fields
*
*   01 W03-HCONN          PIC S9(9) BINARY VALUE ZERO.
*   01 W03-OPTIONS       PIC S9(9) BINARY.
*   01 W03-HOBJ          PIC S9(9) BINARY.
*   01 W03-COMPCODE      PIC S9(9) BINARY.
*   01 W03-REASON        PIC S9(9) BINARY.
*
*   API control blocks
*
*   01 MQM-OBJECT-DESCRIPTOR.
*       COPY CMQODV.
*
*   CMQV contains constants (for setting or testing
*   field values) and return codes (for testing the
*   result of a call)
*
*   01 MQM-CONSTANTS.
*       COPY CMQV SUPPRESS.
* -----*
*   PROCEDURE DIVISION.
* -----*
*   :
* -----*
*   OPEN-TEMP-RESPONSE-QUEUE SECTION.
* -----*
```

COBOL: Creating a dynamic queue

```
*
* This section creates a temporary dynamic queue
* using a model queue
*
* -----*
*
* Change three fields in the Object Descriptor (MQOD)
* control block. (MQODV initializes the other fields)
*
      MOVE MQOT-Q          TO MQOD-OBJECTTYPE.
      MOVE W02-MODEL-QNAME TO MQOD-OBJECTNAME.
      MOVE W02-NAME-PREFIX TO MQOD-DYNAMICQNAME.
*
      COMPUTE W03-OPTIONS = MQ00-INPUT-EXCLUSIVE.
*
      CALL 'MQOPEN' USING W03-HCONN
                        MQOD
                        W03-OPTIONS
                        W03-HOBJ-MODEL
                        W03-COMPCODE
                        W03-REASON.
*
      IF W03-COMPCODE NOT = MQCC-OK
          MOVE 'MQOPEN'      TO M01-MSG4-OPERATION
          MOVE W03-COMPCODE  TO M01-MSG4-COMPCODE
          MOVE W03-REASON    TO M01-MSG4-REASON
          MOVE M01-MESSAGE-4 TO M00-MESSAGE
      ELSE
          MOVE MQOD-OBJECTNAME TO W02-TEMPORARY-Q
      END-IF.
*
      OPEN-TEMP-RESPONSE-QUEUE-EXIT.
*
* Return to performing section.
*
      EXIT.
      EJECT
*
```

Opening an existing queue

This example demonstrates how to use the MQOPEN call to open an existing queue. This extract is taken from the Browse sample application (program CSQ4BVA1) supplied with MQSeries for OS/390. For the names and locations of the sample applications on other platforms, see “Chapter 34. Sample programs (all platforms except OS/390)” on page 359.

```
      :
* -----*
      WORKING-STORAGE SECTION.
* -----*
*
* W01 - Fields derived from the command area input
*
      01 W01-OBJECT          PIC X(48).
*
* W02 - MQM API fields
*
      01 W02-HCONN          PIC S9(9) BINARY VALUE ZERO.
      01 W02-OPTIONS       PIC S9(9) BINARY.
      01 W02-HOBJ          PIC S9(9) BINARY.
      01 W02-COMPCODE      PIC S9(9) BINARY.
      01 W02-REASON        PIC S9(9) BINARY.
*
```

COBOL: Opening an existing queue

```
* CMQODV defines the object descriptor (MQOD)
*
01 MQM-OBJECT-DESCRIPTOR.
   COPY CMQODV.
*
* CMQV contains constants (for setting or testing
* field values) and return codes (for testing the
* result of a call)
*
01 MQM-CONSTANTS.
   COPY CMQV SUPPRESS.
* -----*
E-OPEN-QUEUE SECTION.
* -----*
*
* This section opens the queue
*
* Initialize the Object Descriptor (MQOD) control
* block
* (The copy file initializes the remaining fields.)
*
   MOVE MQOT-Q          TO MQOD-OBJECTTYPE.
   MOVE W01-OBJECT      TO MQOD-OBJECTNAME.
*
* Initialize W02-OPTIONS to open the queue for both
* inquiring about and setting attributes
*
   COMPUTE W02-OPTIONS = MQ00-INQUIRE + MQ00-SET.
*
* Open the queue
*
   CALL 'MQOPEN' USING W02-HCONN
                      MQOD
                      W02-OPTIONS
                      W02-HOBJ
                      W02-COMPCODE
                      W02-REASON.
*
* Test the output from the open
*
* If the completion code is not OK, display a
* separate error message for each of the following
* errors:
*
* Q-MGR-NOT-AVAILABLE - MQM is not available
* CONNECTION-BROKEN   - MQM is no longer connected to CICS
* UNKNOWN-OBJECT-NAME - The queue does not exist
* NOT-AUTHORIZED      - The user is not authorized to open
*                      the queue
*
* For any other error, display an error message
* showing the completion and reason codes
*
   IF W02-COMPCODE NOT = MQCC-OK
      EVALUATE TRUE
*
         WHEN W02-REASON = MQRC-Q-MGR-NOT-AVAILABLE
            MOVE M01-MESSAGE-6 TO M00-MESSAGE
*
         WHEN W02-REASON = MQRC-CONNECTION-BROKEN
            MOVE M01-MESSAGE-6 TO M00-MESSAGE
*
         WHEN W02-REASON = MQRC-UNKNOWN-OBJECT-NAME
            MOVE M01-MESSAGE-2 TO M00-MESSAGE
*
         WHEN W02-REASON = MQRC-NOT-AUTHORIZED
            MOVE M01-MESSAGE-3 TO M00-MESSAGE
```

COBOL: Opening an existing queue

```
*
      WHEN OTHER
          MOVE 'MQOPEN'      TO M01-MSG4-OPERATION
          MOVE W02-COMPCODE TO M01-MSG4-COMPCODE
          MOVE W02-REASON   TO M01-MSG4-REASON
          MOVE M01-MESSAGE-4 TO M00-MESSAGE
      END-EVALUATE
  END-IF.
E-EXIT.
*
*   Return to performing section
*
      EXIT.
      EJECT
```

Closing a queue

This example demonstrates how to use the MQCLOSE call. The variables used in this code extract are those that were set in “Connecting to a queue manager” on page 492. This extract is taken from the Browse sample application (program CSQ4BVA1) supplied with MQSeries for OS/390. For the names and locations of the sample applications on other platforms, see “Chapter 34. Sample programs (all platforms except OS/390)” on page 359.

```
      :
*
*   Close the queue
*
      MOVE MQCO-NONE TO W03-OPTIONS.
*
      CALL 'MQCLOSE' USING W03-HCONN
                          W03-HOBJ
                          W03-OPTIONS
                          W03-COMPCODE
                          W03-REASON.
*
*   Test the output of the MQCLOSE call. If the call
*   fails, print an error message showing the
*   completion code and reason code.
*
      IF (W03-COMPCODE NOT = MQCC-OK) THEN
          MOVE 'CLOSE'      TO W04-MSG4-TYPE
          MOVE W03-COMPCODE TO W04-MSG4-COMPCODE
          MOVE W03-REASON   TO W04-MSG4-REASON
          MOVE W04-MESSAGE-4 TO W00-PRINT-DATA
          PERFORM PRINT-LINE
          MOVE W06-CSQ4-ERROR TO W00-RETURN-CODE
      END-IF.
*
```

Putting a message using MQPUT

This example demonstrates how to use the MQPUT call using context. This extract is taken from the Credit Check sample application (program CSQ4CVB1) supplied with MQSeries for OS/390. For the names and locations of the sample applications on other platforms, see “Chapter 34. Sample programs (all platforms except OS/390)” on page 359.

```
      :
* -----*
```

COBOL: Putting using MQPUT

```

WORKING-STORAGE SECTION.
* -----*
*
*   W02 - Queues processed in this program
*
01 W02-TEMPORARY-Q          PIC X(48).
*
*   W03 - MQM API fields
*
01 W03-HCONN              PIC S9(9) BINARY VALUE ZERO.
01 W03-HOBJ-INQUIRY      PIC S9(9) BINARY.
01 W03-OPTIONS           PIC S9(9) BINARY.
01 W03-BUFFLEN           PIC S9(9) BINARY.
01 W03-COMPCODE          PIC S9(9) BINARY.
01 W03-REASON            PIC S9(9) BINARY.
*
01 W03-PUT-BUFFER.
*
   05 W03-CSQ4BIIM.
   COPY CSQ4VB1.
*
*   API control blocks
*
01 MQM-MESSAGE-DESCRIPTOR.
   COPY CMQMDV.
01 MQM-PUT-MESSAGE-OPTIONS.
   COPY CMQPMOV.
*
*   MQV contains constants (for filling in the
*   control blocks) and return codes (for testing
*   the result of a call).
*
01 MQM-CONSTANTS.
   COPY CMQV SUPPRESS.
* -----*
PROCEDURE DIVISION.
* -----*
:
:
*   Open queue and build message.
:
:
*
*   Set the message descriptor and put-message options to
*   the values required to create the message.
*   Set the length of the message.
*
MOVE MQMT-REQUEST          TO MQMD-MSGTYPE.
MOVE MQCI-NONE             TO MQMD-CORRELID.
MOVE MQMI-NONE            TO MQMD-MSGID.
MOVE W02-TEMPORARY-Q      TO MQMD-REPLYTOQ.
MOVE SPACES                TO MQMD-REPLYTOQMGR.
MOVE 5                    TO MQMD-PRIORITY.
MOVE MQPER-NOT-PERSISTENT TO MQMD-PERSISTENCE.
COMPUTE MQPMO-OPTIONS     = MQPMO-NO-SYNCPPOINT +
                          MQPMO-DEFAULT-CONTEXT.
MOVE LENGTH OF CSQ4BIIM-MSG TO W03-BUFFLEN.
*
CALL 'MQPUT' USING W03-HCONN
                  W03-HOBJ-INQUIRY
                  MQMD
                  MQPMO
                  W03-BUFFLEN
                  W03-PUT-BUFFER
                  W03-COMPCODE
                  W03-REASON.
IF W03-COMPCODE NOT = MQCC-OK

```

COBOL: Putting using MQPUT

```
      :  
      END-IF.
```

Putting a message using MQPUT1

This example demonstrates how to use the MQPUT1 call. This extract is taken from the Credit Check sample application (program CSQ4CVB5) supplied with MQSeries for OS/390. For the names and locations of the sample applications on other platforms, see “Chapter 34. Sample programs (all platforms except OS/390)” on page 359.

```
      :  
* -----*  
WORKING-STORAGE SECTION.  
* -----*  
*  
*   W03 - MQM API fields  
*  
01 W03-HCONN          PIC S9(9) BINARY VALUE ZERO.  
01 W03-OPTIONS       PIC S9(9) BINARY.  
01 W03-COMPCODE      PIC S9(9) BINARY.  
01 W03-REASON        PIC S9(9) BINARY.  
01 W03-BUFFLEN       PIC S9(9) BINARY.  
*  
01 W03-PUT-BUFFER.  
   05 W03-CSQ4BQRM.  
   COPY CSQ4VB4.  
*  
*   API control blocks  
*  
01 MQM-OBJECT-DESCRIPTOR.  
   COPY CMQODV.  
01 MQM-MESSAGE-DESCRIPTOR.  
   COPY CMQMDV.  
01 MQM-PUT-MESSAGE-OPTIONS.  
   COPY CMQPMOV.  
*  
* CMQV contains constants (for filling in the  
* control blocks) and return codes (for testing  
* the result of a call).  
*  
01 MQM-MQV.  
   COPY CMQV SUPPRESS.  
* -----*  
PROCEDURE DIVISION.  
* -----*  
      :  
*   Get the request message.  
      :  
* -----*  
PROCESS-QUERY SECTION.  
* -----*  
      :  
*   Build the reply message.  
      :  
*  
* Set the object descriptor, message descriptor and  
* put-message options to the values required to create  
* the message.  
* Set the length of the message.  
*  
      MOVE MQMD-REPLYTOQ    TO MQOD-OBJECTNAME.
```

```

MOVE MQMD-REPLYTOQMGR TO MQOD-OBJECTQMGRNAME.
MOVE MQMT-REPLY      TO MQMD-MSGTYPE.
MOVE SPACES          TO MQMD-REPLYTOQ.
MOVE SPACES          TO MQMD-REPLYTOQMGR.
MOVE LOW-VALUES      TO MQMD-MSGID.
COMPUTE MQPMO-OPTIONS = MQPMO-SYNCPOINT +
                        MQPMO-PASS-IDENTITY-CONTEXT.
MOVE W03-HOBJ-CHECKQ TO MQPMO-CONTEXT.
MOVE LENGTH OF CSQ4BQRM-MSG TO W03-BUFFLEN.
*
CALL 'MQPUT1' USING W03-HCONN
                  MQOD
                  MQMD
                  MQPMO
                  W03-BUFFLEN
                  W03-PUT-BUFFER
                  W03-COMPCODE
                  W03-REASON.
IF W03-COMPCODE NOT = MQCC-OK
  MOVE 'MQPUT1'      TO M02-OPERATION
  MOVE MQOD-OBJECTNAME TO M02-OBJECTNAME
  PERFORM RECORD-CALL-ERROR
  PERFORM FORWARD-MSG-TO-DLQ
END-IF.
*

```

Getting a message

This example demonstrates how to use the MQGET call to remove a message from a queue. This extract is taken from the Credit Check sample application (program CSQ4CVB1) supplied with MQSeries for OS/390. For the names and locations of the sample applications on other platforms, see “Chapter 34. Sample programs (all platforms except OS/390)” on page 359.

```

:
* -----*
WORKING-STORAGE SECTION.
* -----*
*
*   W03 - MQM API fields
*
01 W03-HCONN          PIC S9(9) BINARY VALUE ZERO.
01 W03-HOBJ-RESPONSE PIC S9(9) BINARY.
01 W03-OPTIONS       PIC S9(9) BINARY.
01 W03-BUFFLEN       PIC S9(9) BINARY.
01 W03-DATALEN       PIC S9(9) BINARY.
01 W03-COMPCODE      PIC S9(9) BINARY.
01 W03-REASON        PIC S9(9) BINARY.
*
01 W03-GET-BUFFER.
   05 W03-CSQ4BAM.
      COPY CSQ4VB2.
*
*   API control blocks
*
01 MQM-MESSAGE-DESCRIPTOR.
   COPY CMQMDV.
01 MQM-GET-MESSAGE-OPTIONS.
   COPY CMQGMV.
*
*   MQV contains constants (for filling in the
*   control blocks) and return codes (for testing
*   the result of a call).
*

```

COBOL: Getting a message

```
01 MQM-CONSTANTS.
   COPY CMQV SUPPRESS.
* -----*
A-MAIN SECTION.
* -----*
  :
*   Open response queue.
  :
* -----*
PROCESS-RESPONSE-SCREEN SECTION.
* -----*
*                                     *
* This section gets a message from the response queue. *
*                                     *
* When a correct response is received, it is          *
* transferred to the map for display; otherwise       *
* an error message is built.                         *
*                                     *
* -----*
*
*   Set get-message options
*
  COMPUTE MQGMO-OPTIONS = MQGMO-SYNCPPOINT +
                        MQGMO-ACCEPT-TRUNCATED-MSG +
                        MQGMO-NO-WAIT.
*
* Set msgid and correlid in MQMD to nulls so that any
* message will qualify.
* Set length to available buffer length.
*
  MOVE MQMI-NONE TO MQMD-MSGID.
  MOVE MQCI-NONE TO MQMD-CORRELID.
  MOVE LENGTH OF W03-GET-BUFFER TO W03-BUFFLEN.
*
  CALL 'MQGET' USING W03-HCONN
                    W03-HOBJ-RESPONSE
                    MQMD
                    MQGMO
                    W03-BUFFLEN
                    W03-GET-BUFFER
                    W03-DATALEN
                    W03-COMPCODE
                    W03-REASON.
  EVALUATE TRUE
    WHEN W03-COMPCODE NOT = MQCC-FAILED
      :
      :
*     Process the message
      :
      :
    WHEN (W03-COMPCODE = MQCC-FAILED AND
          W03-REASON = MQRC-NO-MSG-AVAILABLE)
      MOVE M01-MESSAGE-9 TO M00-MESSAGE
      PERFORM CLEAR-RESPONSE-SCREEN
*
    WHEN OTHER
      MOVE 'MQGET ' TO M01-MSG4-OPERATION
      MOVE W03-COMPCODE TO M01-MSG4-COMPCODE
      MOVE W03-REASON TO M01-MSG4-REASON
      MOVE M01-MESSAGE-4 TO M00-MESSAGE
      PERFORM CLEAR-RESPONSE-SCREEN
  END-EVALUATE.
```


Getting a message using the wait option

This example demonstrates how to use the MQGET call with the wait option and accepting truncated messages. This extract is taken from the Credit Check sample application (program CSQ4CVB5) supplied with MQSeries for OS/390. For the names and locations of the sample applications on other platforms, see "Chapter 34. Sample programs (all platforms except OS/390)" on page 359.

```

      :
* -----*
WORKING-STORAGE SECTION.
* -----*
*
*   W00 - General work fields
*
01 W00-WAIT-INTERVAL   PIC S9(09) BINARY VALUE 30000.
*
*   W03 - MQM API fields
*
01 W03-HCONN          PIC S9(9) BINARY VALUE ZERO.
01 W03-OPTIONS        PIC S9(9) BINARY.
01 W03-HOBJ-CHECKQ    PIC S9(9) BINARY.
01 W03-COMPCODE       PIC S9(9) BINARY.
01 W03-REASON         PIC S9(9) BINARY.
01 W03-DATALEN        PIC S9(9) BINARY.
01 W03-BUFFLEN        PIC S9(9) BINARY.
*
01 W03-MSG-BUFFER.
   05 W03-CSQ4BCAQ.
   COPY CSQ4VB3.
*
*   API control blocks
*
01 MQM-MESSAGE-DESCRIPTOR.
   COPY CMQMDV.
01 MQM-GET-MESSAGE-OPTIONS.
   COPY CMQGMV.
*
*   CMQV contains constants (for filling in the
*   control blocks) and return codes (for testing
*   the result of a call).
*
01 MQM-MQV.
   COPY CMQV SUPPRESS.
* -----*
PROCEDURE DIVISION.
* -----*
      :
*   Open input queue.
      :
*
*   Get and process messages.
*
      COMPUTE MQGMO-OPTIONS = MQGMO-WAIT +
                              MQGMO-ACCEPT-TRUNCATED-MSG +
                              MQGMO-SYNCPOINT.
      MOVE LENGTH OF W03-MSG-BUFFER TO W03-BUFFLEN.
      MOVE W00-WAIT-INTERVAL TO MQGMO-WAITINTERVAL.
      MOVE MQMI-NONE TO MQMD-MSGID.
      MOVE MQCI-NONE TO MQMD-CORRELID.
*
*   Make the first MQGET call outside the loop.
*

```


COBOL: Getting with signaling

```

*
*   05 W03-CSQ4BIIM REDEFINES W03-CSQ4BQRM.
*   COPY CSQ4VB1.
*
*   05 W03-CSQ4BPGM REDEFINES W03-CSQ4BIIM.
*   COPY CSQ4VB5.
*   :
*   API control blocks
*
*   01 MQM-MESSAGE-DESCRIPTOR.
*   COPY CMQMDV.
*   01 MQM-GET-MESSAGE-OPTIONS.
*   COPY CMQGM0V.
*   :
*   MQV contains constants (for filling in the
*   control blocks) and return codes (for testing
*   the result of a call).
*
*   01 MQM-MQV.
*   COPY CMQV SUPPRESS.
* -----*
* LINKAGE SECTION.
* -----*
*   01 L01-ECB-ADDR-LIST.
*       05 L01-ECB-ADDR1          POINTER.
*       05 L01-ECB-ADDR2          POINTER.
*
*   01 L02-ECBS.
*       05 L02-INQUIRY-ECB1       PIC S9(09) BINARY.
*       05 L02-REPLY-ECB2        PIC S9(09) BINARY.
*   01 REDEFINES L02-ECBS.
*       05                        PIC X(02).
*       05 L02-INQUIRY-ECB1-CC   PIC S9(04) BINARY.
*       05                        PIC X(02).
*       05 L02-REPLY-ECB2-CC    PIC S9(04) BINARY.
*
* -----*
* PROCEDURE DIVISION.
* -----*
*   :
*   * Initialize variables, open queues, set signal on
*   * inquiry queue.
*   :
* -----*
* PROCESS-SIGNAL-ACCEPTED SECTION.
* -----*
*   * This section gets a message with signal.  If a
*   * message is received, process it.  If the signal
*   * is set or is already set, the program goes into
*   * an operating system wait.
*   * Otherwise an error is reported and call error set.
* -----*
*
*   PERFORM REPLYQ-GETSIGNAL.
*
*   EVALUATE TRUE
*       WHEN (W03-COMPCODE = MQCC-OK AND
*            W03-REASON = MQRC-NONE)
*           PERFORM PROCESS-REPLYQ-MESSAGE
*
*       WHEN (W03-COMPCODE = MQCC-WARNING AND
*            W03-REASON = MQRC-SIGNAL-REQUEST-ACCEPTED)
*           OR
*           (W03-COMPCODE = MQCC-FAILED AND
*            W03-REASON = MQRC-SIGNAL-OUTSTANDING)
*           PERFORM EXTERNAL-WAIT

```

COBOL: Getting with signaling

```
*
      WHEN OTHER
          MOVE 'MQGET SIGNAL' TO M02-OPERATION
          MOVE MQOD-OBJECTNAME TO M02-OBJECTNAME
          PERFORM RECORD-CALL-ERROR
          MOVE W06-CALL-ERROR TO W06-CALL-STATUS
      END-EVALUATE.
*
PROCESS-SIGNAL-ACCEPTED-EXIT.
*   Return to performing section
      EXIT.
      EJECT
*
* -----*
EXTERNAL-WAIT SECTION.
* -----*
* This section performs an external CICS wait on two *
* ECBs until at least one is posted. It then calls *
* the sections to handle the posted ECB. *
* -----*
      EXEC CICS WAIT EXTERNAL
          ECBLIST(W04-ECB-ADDR-LIST-PTR)
          NUMEVENTS(2)
      END-EXEC.
*
* At least one ECB must have been posted to get to this
* point. Test which ECB has been posted and perform
* the appropriate section.
*
      IF L02-INQUIRY-ECB1 NOT = 0
          PERFORM TEST-INQUIRYQ-ECB
      ELSE
          PERFORM TEST-REPLYQ-ECB
      END-IF.
*
EXTERNAL-WAIT-EXIT.
*
      Return to performing section.
*
      EXIT.
      EJECT
      :
* -----*
REPLYQ-GETSIGNAL SECTION.
* -----*
* This section performs an MQGET call (in syncpoint with *
* signal) on the reply queue. The signal field in the *
* MQGMO is set to the address of the ECB. *
* Response handling is done by the performing section. *
* *
* -----*
*
      COMPUTE MQGMO-OPTIONS          = MQGMO-SYNCPOINT +
          MQGMO-SET-SIGNAL.
      MOVE W00-WAIT-INTERVAL        TO MQGMO-WAITINTERVAL.
      MOVE LENGTH OF W03-GET-BUFFER TO W03-BUFFLEN.
*
      MOVE ZEROS                    TO L02-REPLY-ECB2.
      SET MQGMO-SIGNAL1 TO ADDRESS OF L02-REPLY-ECB2.
*
* Set msgid and correlid to nulls so that any message
* will qualify.
*
      MOVE MQMI-NONE TO MQMD-MSGID.
      MOVE MQCI-NONE TO MQMD-CORRELID.
```

```

*
  CALL 'MQGET' USING W03-HCONN
                    W03-HOBJ-REPLYQ
                    MQMD
                    MQGMO
                    W03-BUFFLEN
                    W03-GET-BUFFER
                    W03-DATALEN
                    W03-COMPCODE
                    W03-REASON.
*
  REPLYQ-GETSIGNAL-EXIT.
*
  Return to performing section.
*
  EXIT.
  EJECT
*
  :
```

Inquiring about the attributes of an object

This example demonstrates how to use the MQINQ call to inquire about the attributes of a queue. This extract is taken from the Queue Attributes sample application (program CSQ4CVC1) supplied with MQSeries for OS/390. For the names and locations of the sample applications on other platforms, see “Chapter 34. Sample programs (all platforms except OS/390)” on page 359.

```

  :
```

```

* -----*
  WORKING-STORAGE SECTION.
* -----*
*
*   W02 - MQM API fields
*
  01 W02-SELECTORCOUNT    PIC S9(9) BINARY VALUE 2.
  01 W02-INTATTRCOUNT    PIC S9(9) BINARY VALUE 2.
  01 W02-CHARATTRLENGTH   PIC S9(9) BINARY VALUE ZERO.
  01 W02-CHARATTRS        PIC X      VALUE LOW-VALUES.
  01 W02-HCONN             PIC S9(9) BINARY VALUE ZERO.
  01 W02-HOBJ              PIC S9(9) BINARY.
  01 W02-COMPCODE          PIC S9(9) BINARY.
  01 W02-REASON            PIC S9(9) BINARY.
  01 W02-SELECTORS-TABLE.
     05 W02-SELECTORS     PIC S9(9) BINARY OCCURS 2 TIMES
  01 W02-INTATTRS-TABLE.
     05 W02-INTATTRS     PIC S9(9) BINARY OCCURS 2 TIMES
*
*   CMQODV defines the object descriptor (MQOD).
*
  01 MQM-OBJECT-DESCRIPTOR.
     COPY CMQODV.
*
*   CMQV contains constants (for setting or testing field
*   values) and return codes (for testing the result of a
*   call).
*
  01 MQM-CONSTANTS.
     COPY CMQV SUPPRESS.
* -----*
  PROCEDURE DIVISION.
* -----*
*
  Get the queue name and open the queue.
```

COBOL: Inquiring about attributes

```
*
*   :
*
*   Initialize the variables for the inquiry call:
*   - Set W02-SELECTORS-TABLE to the attributes whose
*   status is required
*   - All other variables are already set
*
*   MOVE MQIA-INHIBIT-GET TO W02-SELECTORS(1).
*   MOVE MQIA-INHIBIT-PUT TO W02-SELECTORS(2).
*
*   Inquire about the attributes.
*
*   CALL 'MQINQ' USING W02-HCONN,
*                       W02-HOBJ,
*                       W02-SELECTORCOUNT,
*                       W02-SELECTORS-TABLE,
*                       W02-INTATTRCOUNT,
*                       W02-INTATTRS-TABLE,
*                       W02-CHARATTRLENGTH,
*                       W02-CHARATTRS,
*                       W02-COMPCODE,
*                       W02-REASON.
*
*   Test the output from the inquiry:
*
*   - If the completion code is not OK, display an error
*   message showing the completion and reason codes
*
*   - Otherwise, move the correct attribute status into
*   the relevant screen map fields
*
*   IF W02-COMPCODE NOT = MQCC-OK
*       MOVE 'MQINQ'      TO M01-MSG4-OPERATION
*       MOVE W02-COMPCODE TO M01-MSG4-COMPCODE
*       MOVE W02-REASON   TO M01-MSG4-REASON
*       MOVE M01-MESSAGE-4 TO M00-MESSAGE
*
*   ELSE
*       Process the changes.
*   :
*       END-IF.
*   :
```

Setting the attributes of a queue

This example demonstrates how to use the MQSET call to change the attributes of a queue. This extract is taken from the Queue Attributes sample application (program CSQ4CVC1) supplied with MQSeries for OS/390. For the names and locations of the sample applications on other platforms, see “Chapter 34. Sample programs (all platforms except OS/390)” on page 359.

```
*
*   :
*   -----*
*   WORKING-STORAGE SECTION.
*   -----*
*
*   W02 - MQM API fields
*
*   01 W02-SELECTORCOUNT    PIC S9(9) BINARY VALUE 2.
*   01 W02-INTATTRCOUNT    PIC S9(9) BINARY VALUE 2.
*   01 W02-CHARATTRLENGTH   PIC S9(9) BINARY VALUE ZERO.
*   01 W02-CHARATTRS        PIC X      VALUE LOW-VALUES.
```

COBOL: Setting queue attributes

```
01 W02-HCONN          PIC S9(9) BINARY VALUE ZERO.
01 W02-HOBJ           PIC S9(9) BINARY.
01 W02-COMPCODE       PIC S9(9) BINARY.
01 W02-REASON         PIC S9(9) BINARY.
01 W02-SELECTORS-TABLE.
   05 W02-SELECTORS   PIC S9(9) BINARY OCCURS 2 TIMES.
01 W02-INTATTRS-TABLE.
   05 W02-INTATTRS   PIC S9(9) BINARY OCCURS 2 TIMES.
*
* CMQODV defines the object descriptor (MQOD).
*
01 MQM-OBJECT-DESCRIPTOR.
   COPY CMQODV.
*
* CMQV contains constants (for setting or testing
* field values) and return codes (for testing the
* result of a call).
*
01 MQM-CONSTANTS.
   COPY CMQV SUPPRESS.
* -----*
PROCEDURE DIVISION.
* -----*
*
*   Get the queue name and open the queue.
*
*   :
*
*
* Initialize the variables required for the set call:
* - Set W02-SELECTORS-TABLE to the attributes to be set
* - Set W02-INTATTRS-TABLE to the required status
* - All other variables are already set
*
   MOVE MQIA-INHIBIT-GET TO W02-SELECTORS(1).
   MOVE MQIA-INHIBIT-PUT TO W02-SELECTORS(2).
   MOVE MQQA-GET-INHIBITED TO W02-INTATTRS(1).
   MOVE MQQA-PUT-INHIBITED TO W02-INTATTRS(2).
*
* Set the attributes.
*
   CALL 'MQSET' USING W02-HCONN,
                     W02-HOBJ,
                     W02-SELECTORCOUNT,
                     W02-SELECTORS-TABLE,
                     W02-INTATTRCOUNT,
                     W02-INTATTRS-TABLE,
                     W02-CHARATTRLENGTH,
                     W02-CHARATTRS,
                     W02-COMPCODE,
                     W02-REASON.
*
* Test the output from the call:
*
* - If the completion code is not OK, display an error
* message showing the completion and reason codes
*
* - Otherwise, move 'INHIBITED' into the relevant
* screen map fields
*
   IF W02-COMPCODE NOT = MQCC-OK
      MOVE 'MQSET' TO M01-MSG4-OPERATION
      MOVE W02-COMPCODE TO M01-MSG4-COMPCODE
      MOVE W02-REASON TO M01-MSG4-REASON
      MOVE M01-MESSAGE-4 TO M00-MESSAGE
   ELSE
*

```

COBOL: Setting queue attributes

```
*          Process the changes.  
  :  
  END-IF.
```

Appendix D. System/390 assembler-language examples

The extracts in this appendix are mostly taken from the MQSeries for OS/390 sample applications.

The examples in this appendix demonstrate the following techniques:

- “Connecting to a queue manager” on page 510
- “Disconnecting from a queue manager” on page 511
- “Creating a dynamic queue” on page 511
- “Opening an existing queue” on page 513
- “Closing a queue” on page 514
- “Putting a message using MQPUT” on page 514
- “Putting a message using MQPUT1” on page 516
- “Getting a message” on page 517
- “Getting a message using the wait option” on page 519
- “Getting a message using signaling” on page 520
- “Inquiring about and setting the attributes of a queue” on page 522

Connecting to a queue manager

This example demonstrates how to use the MQCONN call to connect a program to a queue manager in OS/390 batch. This extract is taken from the Browse sample program (CSQ4BAA1) supplied with MQSeries for OS/390.

```

      :
WORKAREA DSECT
*
PARMLIST CALL ,(0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0),VL,MF=L
*
COMPCODE DS    F           Completion code
REASON   DS    F           Reason code
HCONN    DS    F           Connection handle
          ORG
PARMADDR DS    F           Address of parm field
PARMLEN  DS    H           Length of parm field
*
MQMNAME  DS    CL48        Queue manager name
*
*
*****
* SECTION NAME : MAINPARM *
*****
MAINPARM DS    0H
          MVI   MQMNAME,X'40'
          MVC   MQMNAME+1(L'MQMNAME-1),MQMNAME
*
* Space out first byte and initialize
*
*
* Code to address and verify parameters passed omitted
*
*
PARM1MVE DS    0H
          SR    R1,R3           Length of data
          LA    R4,MQMNAME     Address for target
          BCTR  R1,R0           Reduce for execute
          EX    R1,MOVEPARM     Move the data
*
*****
* EXECUTES *
*****
MOVEPARM MVC    0(*-*,R4),0(R3)
*
          EJECT
*****
* SECTION NAME : MAINCONN *
*****
*
*
MAINCONN DS    0H
          XC    HCONN,HCONN     Null connection handle
*
          CALL  MQCONN,          X
              (MQMNAME,         X
              HCONN,            X
              COMPCODE,         X
              REASON),          X
              MF=(E,PARMLIST),VL
*
          LA    R0,MQCC_OK       Expected compcode
          C     R0,COMPCODE      As expected?
          BER   R6               Yes .. return to caller

```

Assembler: Connecting to a queue manager

```
*
      MVC   INF4_TYP,=CL10'CONNECT  '
      BAL   R7,ERRCODE              Translate error
      LA    R0,8                    Set exit code
      ST    R0,EXITCODE             to 8
      B     ENDPROG                 End the program
*
```

Disconnecting from a queue manager

This example demonstrates how to use the MQDISC call to disconnect a program from a queue manager in OS/390 batch. This extract is not taken from the sample applications supplied with MQSeries.

```

      :
*
*       ISSUE MQI DISC REQUEST USING REENTRANT FORM
*       OF CALL MACRO
*
*       HCONN WAS SET BY A PREVIOUS MQCONN REQUEST
*       R5 = WORK REGISTER
*
DISC   DS    0H
      CALL  MQDISC,                   X
          (HCONN,                     X
           COMPCODE,                  X
           REASON),                   X
          VL,MF=(E,CALLLST)
*
      LA   R5,MQCC_OK
      C   R5,COMPCODE
      BNE BADCALL
      :
BADCALL DS    0H
      :
*
*           CONSTANTS
*
      CMQA
*
*       WORKING STORAGE (RE-ENTRANT)
*
WEG3   DSECT
*
CALLLST CALL , (0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0),VL,MF=L
*
HCONN  DS    F
COMPCODE DS  F
REASON DS    F
*
*
LEG3   EQU   *-WKEG3
      END
```

Creating a dynamic queue

This example demonstrates how to use the MQOPEN call to create a dynamic queue. This extract is not taken from the sample applications supplied with MQSeries.

```

      :
*
```

Assembler: Creating a dynamic queue

```

*      R5 = WORK REGISTER.
*
OPEN    DS    0H
*
MVC    WOD_AREA,MQOD_AREA INITIALIZE WORKING VERSION OF
*                               MQOD WITH DEFAULTS
MVC    WOD_OBJECTNAME,MOD_Q    COPY IN THE MODEL Q NAME
MVC    WOD_DYNAMICQNAME,DYN_Q  COPY IN THE DYNAMIC Q NAME
L      R5,=AL4(MQOO_OUTPUT)    OPEN FOR OUTPUT AND
A      R5,=AL4(MQOO_INQUIRE)  INQUIRE
ST     R5,OPTIONS
*
* ISSUE MQI OPEN REQUEST USING REENTRANT
* FORM OF CALL MACRO
*
          CALL MQOPEN,          X
          (HCONN,              X
           WOD,                 X
           OPTIONS,            X
           HOBJ,               X
           COMPCODE,          X
           REASON),VL,MF=(E,CALLST)
*
LA     R5,MQCC_OK              CHECK THE COMPLETION CODE
C     R5,COMPCODE              FROM THE REQUEST AND BRANCH
BNE   BADCALL                 TO ERROR ROUTINE IF NOT MQCC_OK
*
MVC    TEMP_Q,WOD_OBJECTNAME  SAVE NAME OF TEMPORARY Q
*                               CREATED BY OPEN OF MODEL Q
*
*
*
BADCALL DS    0H
*
*
*   CONSTANTS:
*
MOD_Q  DC    CL48'QUERY.REPLY.MODEL'  MODEL QUEUE NAME
DYN_Q  DC    CL48'QUERY.TEMPQ.*'      DYNAMIC QUEUE NAME
*
          CMQODA DSECT=NO,LIST=YES  CONSTANT VERSION OF MQOD
          CMQA                               MQI VALUE EQUATES
*
*   WORKING STORAGE
*
          DFHEISTG
HCONN  DS    F                  CONNECTION HANDLE
OPTIONS DS    F                  OPEN OPTIONS
HOBJ   DS    F                  OBJECT HANDLE
COMPCODE DS    F                MQI COMPLETION CODE
REASON  DS    F                MQI REASON CODE
TEMP_Q  DS    CL(MQ_Q_NAME_LENGTH)  SAVED QNAME AFTER OPEN
*
WOD     CMQODA DSECT=NO,LIST=YES  WORKING VERSION OF MQOD
*
CALLST  CALL  ,(0,0,0,0,0,0,0,0,0,0),VL,MF=L  LIST FORM
                                                OF CALL
                                                MACRO
*
*
*
          END

```

Opening an existing queue

This example demonstrates how to use the MQOPEN call to open a queue that has already been defined. It shows how to specify two options. This extract is not taken from the sample applications supplied with MQSeries.

```

      :
*
*   R5 = WORK REGISTER.
*
OPEN   DS   0H
*
MVC   WOD_AREA,MQOD_AREA  INITIALIZE WORKING VERSION OF
*                               MQOD WITH DEFAULTS
MVC   WOD_OBJECTNAME,Q_NAME  SPECIFY Q NAME TO OPEN
LA    R5,MQOO_INPUT_EXCLUSIVE  OPEN FOR MQGET CALLS
*
      ST   R5,OPTIONS
*
* ISSUE MQI OPEN REQUEST USING REENTRANT FORM
* OF CALL MACRO
*
      CALL MQOPEN,                X
          (HCONN,                  X
           WOD,                     X
           OPTIONS,                 X
           HOBJ,                    X
           COMPCODE,                X
           REASON),VL,MF=(E,CALLLST)
*
LA    R5,MQCC_OK          CHECK THE COMPLETION CODE
C     R5,COMPCODE        FROM THE REQUEST AND BRANCH
BNE  BADCALL            TO ERROR ROUTINE IF NOT MQCC_OK
*
      :
BADCALL DS  0H
      :
*
*
*   CONSTANTS:
*
Q_NAME DC  CL48'REQUEST.QUEUE'  NAME OF QUEUE TO OPEN
*
      CMQODA DSECT=NO,LIST=YES  CONSTANT VERSION OF MQOD
      CMQA                               MQI VALUE EQUATES
*
*   WORKING STORAGE
*
      DFHEISTG
HCONN DS F          CONNECTION HANDLE
OPTIONS DS F        OPEN OPTIONS
HOBJ DS F           OBJECT HANDLE
COMPCODE DS F       MQI COMPLETION CODE
REASON DS F         MQI REASON CODE
*
WOD CMQODA DSECT=NO,LIST=YES  WORKING VERSION OF MQOD
*
CALLLST CALL ,(0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0),VL,MF=L  LIST FORM
*                                               OF CALL
*                                               MACRO
*
      :
      END

```

Assembler: Closing a queue

Closing a queue

This example demonstrates how to use the MQCLOSE call to close a queue. This extract is not taken from the sample applications supplied with MQSeries.

```

      :
*
* ISSUE MQI CLOSE REQUEST USING REENTRANT FROM OF
* CALL MACRO
*
*       HCONN WAS SET BY A PREVIOUS MQCONN REQUEST
*       HOBJ WAS SET BY A PREVIOUS MQOPEN REQUEST
*       R5 = WORK REGISTER
*
CLOSE  DS    0H
       LA    R5,MQCO_NONE      NO SPECIAL CLOSE OPTIONS
       ST    R5,OPTIONS        ARE REQUIRED.
*
       CALL  MQCLOSE,          X
              (HCONN,          X
              HOBJ,            X
              OPTIONS,         X
              COMPCODE,        X
              REASON),         X
              VL,MF=(E,CALLLST)
*
       LA    R5,MQCC_OK
       C     R5,COMPCODE
       BNE   BADCALL
*
      :
BADCALL DS    0H
      :
*           CONSTANTS
*
*       CMQA
*
*       WORKING STORAGE (REENTRANT)
*
WEG4   DSECT
*
CALLLST CALL , (0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0),VL,MF=L
*
HCONN  DS    F
HOBJ   DS    F
OPTIONS DS    F
COMPCODE DS  F
REASON DS    F
*
*
LEG4   EQU   *-WKEG4
      END
```

Putting a message using MQPUT

This example demonstrates how to use the MQPUT call to put a message on a queue. This extract is not taken from the sample applications supplied with MQSeries.

```

      :
*       CONNECT TO QUEUE MANAGER
```

Assembler: Putting using MQPUT

```

*
CONN    DS    0H
      :
*
*      OPEN A QUEUE
*
OPEN    DS    0H
      :
*
*      R4,R5,R6,R7 = WORK REGISTER.
*
PUT     DS    0H
      LA    R4,MQMD           SET UP ADDRESSES AND
      LA    R5,MQMD_LENGTH   LENGTH FOR USE BY MVCL
      LA    R6,WMD           INSTRUCTION, AS MQMD IS
      LA    R7,WMD_LENGTH    OVER 256 BYES LONG.
      MVCL  R6,R4           INITIALIZE WORKING VERSION
*                               OF MESSAGE DESCRIPTOR
*
      MVC  WPMO_AREA,MQPMO_AREA  INITIALIZE WORKING MQPMO
*
      LA    R5,BUFFER_LEN     RETRIEVE THE BUFFER LENGTH
      ST    R5,BUFFLEN        AND SAVE IT FOR MQM USE
*
      MVC  BUFFER,TEST_MSG    SET THE MESSAGE TO BE PUT
*
*      ISSUE MQI PUT REQUEST USING REENTRANT FORM
*      OF CALL MACRO
*
*      HCONN WAS SET BY PREVIOUS MQCONN REQUEST
*      HOBJ WAS SET BY PREVIOUS MQOPEN REQUEST
*
      CALL MQPUT,              X
          (HCONN,              X
           HOBJ,                X
           WMD,                  X
           WPMO,                  X
           BUFFLEN,              X
           BUFFER,                X
           COMPCODE,              X
           REASON),VL,MF=(E,CALLLST)
*
      LA    R5,MQCC_OK
      C     R5,COMPCODE
      BNE  BADCALL
*
      :
BADCALL DS    0H
      :
*
*      CONSTANTS
*
      CMQMDA DSECT=NO,LIST=YES,PERSISTENCE=MQPER_PERSISTENT
      CMQPMOA DSECT=NO,LIST=YES
      CMQA
      TEST_MSG DC CL80'THIS IS A TEST MESSAGE'
*
*      WORKING STORAGE DSECT
*
WORKSTG DSECT
*
COMPCODE DS F
REASON   DS F
BUFFLEN  DS F
OPTIONS  DS F

```

Assembler: Putting using MQPUT

```
HCONN    DS F
HOBJ     DS F
*
BUFFER   DS CL80
BUFFER_LEN EQU *-BUFFER
*
WMD      CMQMDA DSECT=NO,LIST=NO
WPMO     CMQPMOA DSECT=NO,LIST=NO
*
CALLLST  CALL ,(0,0,0,0,0,0,0,0,0,0,0),VL,MF=L
*
      :
      END
```

Putting a message using MQPUT1

This example demonstrates how to use the MQPUT1 call to open a queue, put a single message on the queue, then close the queue. This extract is not taken from the sample applications supplied with MQSeries.

```
      :
*
*   CONNECT TO QUEUE MANAGER
*
CONN     DS 0H
      :
*
*   R4,R5,R6,R7 = WORK REGISTER.
*
PUT      DS 0H
*
MVC WOD_AREA,MQOD_AREA      INITIALIZE WORKING VERSION OF
*                               MQOD WITH DEFAULTS
MVC WOD_OBJECTNAME,Q_NAME  SPECIFY Q NAME FOR PUT1
*
LA R4,MQMD                  SET UP ADDRESSES AND
LA R5,MQMD_LENGTH          LENGTH FOR USE BY MVCL
LA R6,WMD                   INSTRUCTION, AS MQMD IS
LA R7,WMD_LENGTH           OVER 256 BYES LONG.
MVCL R6,R4                  INITIALIZE WORKING VERSION
*                               OF MESSAGE DESCRIPTOR
*
MVC WPMO_AREA,MQPMO_AREA    INITIALIZE WORKING MQPMO
*
LA R5,BUFFER_LEN           RETRIEVE THE BUFFER LENGTH
ST R5,BUFFLEN              AND SAVE IT FOR MQM USE
*
MVC BUFFER,TEST_MSG        SET THE MESSAGE TO BE PUT
*
*   ISSUE MQI PUT REQUEST USING REENTRANT FORM OF CALL MACRO
*
*   HCONN WAS SET BY PREVIOUS MQCONN REQUEST
*   HOBJ WAS SET BY PREVIOUS MQOPEN REQUEST
*
CALL MQPUT1,                X
      (HCONN,                X
      LMQOD,                  X
      LMQMD,                  X
      LMQPMO,                 X
      BUFFERLENGTH,           X
      BUFFER,                  X
      COMPCODE,               X
```


Assembler: Putting using MQPUT1

```
                REASON),VL,MF=(E,CALLLST)
*
*           LA  R5,MQCC_OK
*           C   R5,COMP CODE
*           BNE BADCALL
*
*           :
BADCALL DS 0H
*           :
*
*           CONSTANTS
*
*           CMQMDA DSECT=NO,LIST=YES,PERSISTENCE=MQPER_PERSISTENT
*           CMQPMOA DSECT=NO,LIST=YES
*           CMQODA DSECT=NO,LIST=YES
*           CMQA
*
*           TEST_MSG DC CL80'THIS IS ANOTHER TEST MESSAGE'
*           Q_NAME   DC CL48'TEST.QUEUE.NAME'
*
*           WORKING STORAGE DSECT
*
*           WORKSTG DSECT
*
*           COMPCODE DS F
*           REASON   DS F
*           BUFFLEN  DS F
*           OPTIONS  DS F
*           HCONN    DS F
*           HOBJ     DS F
*
*           BUFFER   DS CL80
*           BUFFER_LEN EQU *-BUFFER
*
*           WOD  CMQODA DSECT=NO,LIST=YES   WORKING VERSION OF MQOD
*           WMD  CMQMDA DSECT=NO,LIST=NO
*           WPMO CMQPMOA DSECT=NO,LIST=NO
*
*           CALLLST CALL ,(0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0),VL,MF=L
*
*           :
*           END
```

Getting a message

This example demonstrates how to use the MQGET call to remove a message from a queue. This extract is not taken from the sample applications supplied with MQSeries.

```
           :
*
*           CONNECT TO QUEUE MANAGER
*
*           CONN   DS 0H
*           :
*
*           OPEN A QUEUE FOR GET
*
*           OPEN   DS 0H
*           :
*
*           R4,R5,R6,R7 = WORK REGISTER.
*
```

Assembler: Getting a message

```

GET DS 0H
LA R4,MQMD SET UP ADDRESSES AND
LA R5,MQMD_LENGTH LENGTH FOR USE BY MVCL
LA R6,WMD INSTRUCTION, AS MQMD IS
LA R7,WMD_LENGTH OVER 256 BYES LONG.
MVCL R6,R4 INITIALIZE WORKING VERSION
* OF MESSAGE DESCRIPTOR
*
MVC WGMO_AREA,MQGMO_AREA INITIALIZE WORKING MQGMO
*
LA R5,BUFFER_LEN RETRIEVE THE BUFFER LENGTH
ST R5,BUFFLEN AND SAVE IT FOR MQM USE
*
*
* ISSUE MQI GET REQUEST USING REENTRANT FORM OF CALL MACRO
*
* HCONN WAS SET BY PREVIOUS MQCONN REQUEST
* HOBJ WAS SET BY PREVIOUS MQOPEN REQUEST
*
CALL MQGET, X
(HCONN, X
HOBJ, X
WMD, X
WGMO, X
BUFFLEN, X
BUFFER, X
DATALEN, X
COMPCODE, X
REASON), X
VL,MF=(E,CALLLST)
*
LA R5,MQCC_OK
C R5,COMPCODE
BNE BADCALL
*
:
BADCALL DS 0H
:
*
* CONSTANTS
*
CMQMDA DSECT=NO,LIST=YES
CMQGMOA DSECT=NO,LIST=YES
CMQA
*
* WORKING STORAGE DSECT
*
WORKSTG DSECT
*
COMPCODE DS F
REASON DS F
BUFFLEN DS F
DATALEN DS F
OPTIONS DS F
HCONN DS F
HOBJ DS F
*
BUFFER DS CL80
BUFFER_LEN EQU *-BUFFER
*
WMD CMQMDA DSECT=NO,LIST=NO
WGMO CMQGMOA DSECT=NO,LIST=NO
*
CALLLST CALL ,(0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0),VL,MF=L
*

```

```

:
END

```

Getting a message using the wait option

This example demonstrates how to use the wait option of the MQGET call. This code accepts truncated messages. This extract is not taken from the sample applications supplied with MQSeries.

```

:
*   CONNECT TO QUEUE MANAGER
CONN DS 0H
:
*   OPEN A QUEUE FOR GET
OPEN DS 0H
:
*   R4,R5,R6,R7 = WORK REGISTER.
GET DS 0H
LA R4,MQMD          SET UP ADDRESSES AND
LA R5,MQMD_LENGTH  LENGTH FOR USE BY MVCL
LA R6,WMD           INSTRUCTION, AS MQMD IS
LA R7,WMD_LENGTH   OVER 256 BYES LONG.
MVCL R6,R4         INITIALIZE WORKING VERSION
*                   OF MESSAGE DESCRIPTOR
*
*   MVC WGMO_AREA,MQGMO_AREA  INITIALIZE WORKING MQGMO
L R5,=AL4(MQGMO_WAIT)
A R5,=AL4(MQGMO_ACCEPT_TRUNCATED_MSG)
ST R5,WGMO_OPTIONS
MVC WGMO_WAITINTERVAL,TWO_MINUTES  WAIT UP TO TWO
*                                     MINUTES BEFORE
*                                     FAILING THE
*                                     CALL
*
*   LA R5,BUFFER_LEN  RETRIEVE THE BUFFER LENGTH
ST R5,BUFFLEN        AND SAVE IT FOR MQM USE
*
*   ISSUE MQI GET REQUEST USING REENTRANT FORM OF CALL MACRO
*
*   HCONN WAS SET BY PREVIOUS MQCONN REQUEST
*   HOBJ WAS SET BY PREVIOUS MQOPEN REQUEST
*
*   CALL MQGET,          X
*   (HCONN,             X
*   HOBJ,               X
*   WMD,               X
*   WGMO,              X
*   BUFFLEN,          X
*   BUFFER,           X
*   DATALEN,        X
*   COMPCODE,        X
*   REASON),          X
*   VL,MF=(E,CALLST)
*
*   LA R5,MQCC_OK      DID THE MQGET REQUEST
C R5,COMPCODE         WORK OK?
BE GETOK             YES, SO GO AND PROCESS.
LA R5,MQCC_WARNING    NO, SO CHECK FOR A WARNING.
C R5,COMPCODE         IS THIS A WARNING?
BE CHECK_W           YES, SO CHECK THE REASON.
*
*   LA R5,MQRC_NO_MSG_AVAILABLE  IT MUST BE AN ERROR.
IS IT DUE TO AN EMPTY

```

Assembler: Getting with wait option

```
      C   R5,REASON          QUEUE?
      BE  NOMSG             YES, SO HANDLE THE ERROR
      B   BADCALL          NO, SO GO TO ERROR ROUTINE
*
CHECK_W DS 0H
      LA  R5,MQRC_TRUNCATED_MSG_ACCEPTED IS THIS A
                                           TRUNCATED
                                           MESSAGE?
      C   R5,REASON
      BE  GETOK             YES, SO GO AND PROCESS.
      B   BADCALL          NO, SOME OTHER WARNING
*
NOMSG  DS 0H
      :
      :
GETOK  DS 0H
      :
      :
BADCALL DS 0H
      :
*
*   CONSTANTS
*
      CMQMDA DSECT=NO,LIST=YES
      CMQGMOA DSECT=NO,LIST=YES
      CMQA
*
TWO_MINUTES DC F'120000'   GET WAIT INTERVAL
*
*   WORKING STORAGE DSECT
*
WORKSTG DSECT
*
COMPCODE DS F
REASON   DS F
BUFFLEN  DS F
DATALEN  DS F
OPTIONS  DS F
HCONN    DS F
HOBJ     DS F
*
BUFFER   DS CL80
BUFFER_LEN EQU *-BUFFER
*
WMD      CMQMDA DSECT=NO,LIST=NO
WGM0     CMQGMOA DSECT=NO,LIST=NO
*
CALLLST CALL , (0,0,0,0,0,0,0,0,0,0,0,0),VL,MF=L
*
      :
      :
      END
```

Getting a message using signaling

This example demonstrates how to use the MQGET call to set a signal so that you are notified when a suitable message arrives on a queue. This extract is not taken from the sample applications supplied with MQSeries.

```
      :
*
*   CONNECT TO QUEUE MANAGER
*
CONN   DS 0H
      :
*
*   OPEN A QUEUE FOR GET
```

Assembler: Getting using signaling

```

*
OPEN    DS    0H
      :
*
*      R4,R5,R6,R7 = WORK REGISTER.
*
GET     DS    0H
      LA    R4,MQMD           SET UP ADDRESSES AND
      LA    R5,MQMD_LENGTH   LENGTH FOR USE BY MVCL
      LA    R6,WMD           INSTRUCTION, AS MQMD IS
      LA    R7,WMD_LENGTH   OVER 256 BYES LONG.
      MVCL R6,R4           INITIALIZE WORKING VERSION
                          OF MESSAGE DESCRIPTOR
*
*
MVC     WGMO_AREA,MQGMO_AREA INITIALIZE WORKING MQGMO
      LA    R5,MQGMO_SET_SIGNAL
      ST    R5,WGMO_OPTIONS
      MVC   WGMO_WAITINTERVAL,FIVE_MINUTES  WAIT UP TO FIVE
                          MINUTES BEFORE
                          FAILING THE CALL
*
*
XC      SIG_ECB,SIG_ECB     CLEAR THE ECB
      LA    R5,SIG_ECB     GET THE ADDRESS OF THE ECB
      ST    R5,WGMO_SIGNAL1 AND PUT IT IN THE WORKING
                          MQGMO
*
*
      LA    R5,BUFFER_LEN   RETRIEVE THE BUFFER LENGTH
      ST    R5,BUFFLEN     AND SAVE IT FOR MQM USE
*
*
      ISSUE MQI GET REQUEST USING REENTRANT FORM OF CALL MACRO
*
*      HCONN WAS SET BY PREVIOUS MQCONN REQUEST
*      HOBJ WAS SET BY PREVIOUS MQOPEN REQUEST
*
      CALL  MQGET,          X
            (HCONN,        X
            HOBJ,          X
            WMD,           X
            WGMO,          X
            BUFFLEN,       X
            BUFFER,        X
            DATALEN,      X
            COMPCODE,      X
            REASON),       X
            VL,MF=(E,CALLST)
*
      LA    R5,MQCC_OK      DID THE MQGET REQUEST
      C     R5,COMPCODE     WORK OK?
      BE    GETOK           YES, SO GO AND PROCESS.
      LA    R5,MQCC_WARNING NO, SO CHECK FOR A WARNING.
      C     R5,COMPCODE     IS THIS A WARNING?
      BE    CHECK_W        YES, SO CHECK THE REASON.
      B     BADCALL        NO, SO GO TO ERROR ROUTINE
*
CHECK_W DS    0H
      LA    R5,MQRC_SIGNAL_REQUEST_ACCEPTED
      C     R5,REASON      SIGNAL REQUEST SIGNAL SET?
      BNE  BADCALL        NO, SOME ERROR OCCURRED
      B     DOWORK        YES, SO DO SOMETHING
*
                          ELSE
*
CHECKSIG DS    0H
      CLC  SIG_ECB+1(3),=AL3(MQEC_MSG_ARRIVED)
                          IS A MESSAGE AVAILABLE?

```

Assembler: Getting using signaling

```

        BE GET                YES, SO GO AND GET IT
*
        CLC SIG_ECB+1(3),=AL3(MQEC_WAIT_INTERVAL_EXPIRED)
                                HAVE WE WAITED LONG ENOUGH?
        BE NOMSG              YES, SO SAY NO MSG AVAILABLE
        B  BADCALL            IF IT'S ANYTHING ELSE
*                                GO TO ERROR ROUTINE.
*
DOWORK  DS  0H
        :
        TM SIG_ECB,X'40'      HAS THE SIGNAL ECB BEEN POSTED?
        BO CHECKSIG          YES, SO GO AND CHECK WHY
        B  DOWORK            NO, SO GO AND DO MORE WORK
*
NOMSG   DS  0H
        :
GETOK   DS  0H
        :
BADCALL DS  0H
        :
*
*      CONSTANTS
*
        CMQMDA DSECT=NO,LIST=YES
        CMQMOA DSECT=NO,LIST=YES
        CMQA
*
FIVE_MINUTES DC F'300000'      GET SIGNAL INTERVAL
*
*      WORKING STORAGE DSECT
*
WORKSTG DSECT
*
COMPCODE DS F
REASON   DS F
BUFFLEN  DS F
DATALEN  DS F
OPTIONS  DS F
HCONN    DS F
HOBJ     DS F
SIG_ECB  DS F
*
BUFFER   DS CL80
BUFFER_LEN EQU *-BUFFER
*
WMD      CMQMDA DSECT=NO,LIST=NO
WGMO     CMQMOA DSECT=NO,LIST=NO
*
CALLLST  CALL ,(0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0),VL,MF=L
*
        :
        END
```

Inquiring about and setting the attributes of a queue

This example demonstrates how to use the MQINQ call to inquire about the attributes of a queue and to use the MQSET call to change the attributes of a queue. This extract is taken from the Queue Attributes sample application (program CSQ4CAC1) supplied with MQSeries for OS/390.

```

        :
DFHEISTG DSECT
```

Assembler: Inquiring about and setting attributes

```

:
OBJDESC CMQODA LIST=YES Working object descriptor
*
SELECTORCOUNT DS F Number of selectors
INTATTRCOUNT DS F Number of integer attributes
CHARATTRLENGTH DS F char attributes length
CHARATTRS DS C Area for char attributes
*
OPTIONS DS F Command options
HCONN DS F Handle of connection
HOBJ DS F Handle of object
COMPCODE DS F Completion code
REASON DS F Reason code
SELECTOR DS 2F Array of selectors
INTATTRS DS 2F Array of integer attributes
:
OBJECT DS CL(MQ_Q_NAME_LENGTH) Name of queue
:
CALLLIST CALL ,(0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0),VL,MF=L
*****
* PROGRAM EXECUTION STARTS HERE *
:
CSQ4CAC1 DFHEIENT CODEREG=(R3),DATAREG=(R13)
:
* Initialize the variables for the set call
*
SR R0,R0 Clear register zero
ST R0,CHARATTRLENGTH Set char length to zero
LA R0,2 Load to set
ST R0,SELECTORCOUNT selectors add
ST R0,INTATTRCOUNT integer attributes
*
LA R0,MQIA_INHIBIT_GET Load q attribute selector
ST R0,SELECTOR+0 Place in field
LA R0,MQIA_INHIBIT_PUT Load q attribute selector
ST R0,SELECTOR+4 Place in field
*
UPDTEST DS 0H
CLC ACTION,CINHIB Are we inhibiting?
BE UPDINHBT Yes branch to section
*
CLC ACTION,CALLOW Are we allowing?
BE UPDALLOW Yes branch to section
*
MVC M00_MSG,M01_MSG1 Invalid request
BR R6 Return to caller
*
UPDINHBT DS 0H
MVC UPDTYPE,CINHIBIT Indicate action type
LA R0,MQQA_GET_INHIBITED Load attribute value
ST R0,INTATTRS+0 Place in field
LA R0,MQQA_PUT_INHIBITED Load attribute value
ST R0,INTATTRS+4 Place in field
B UPDCALL Go and do call
*
UPDALLOW DS 0H
MVC UPDTYPE,CALLOWED Indicate action type
LA R0,MQQA_GET_ALLOWED Load attribute value
ST R0,INTATTRS+0 Place in field
LA R0,MQQA_PUT_ALLOWED Load attribute value
ST R0,INTATTRS+4 Place in field
B UPDCALL Go and do call
*
UPDCALL DS 0H
CALL MQSET, C
(HCONN, C

```

Assembler: Inquiring about and setting attributes

```

                                HOBJ,                C
                                SELECTORCOUNT,      C
                                SELECTOR,            C
                                INTATTRCOUNT,       C
                                INTATTRS,           C
                                CHARATTRLENGTH,      C
                                CHARATTRS,          C
                                COMPCODE,           C
                                REASON),            C
                                VL,MF=(E,CALLLIST)
*
*   LA  R0,MQCC_OK    Load expected compcode
*   C   R0,COMPCODE  Was set successful?
*
*   :
* SECTION NAME : INQUIRE
* FUNCTION     : Inquires on the objects attributes
* CALLED BY    : PROCESS
* CALLS        : OPEN, CLOSE, CODES
* RETURN       : To Register 6
INQUIRE DS 0H
*
*   :
*
* Initialize the variables for the inquire call
*
*   SR  R0,R0        Clear register zero
*   ST  R0,CHARATTRLENGTH Set char length to zero
*   LA  R0,2         Load to set
*   ST  R0,SELECTORCOUNT selectors add
*   ST  R0,INTATTRCOUNT integer attributes
*
*   LA  R0,MQIA_INHIBIT_GET Load attribute value
*   ST  R0,SELECTOR+0      Place in field
*   LA  R0,MQIA_INHIBIT_PUT Load attribute value
*   ST  R0,SELECTOR+4      Place in field
*   CALL MQINQ,
*       (HCONN,
*        HOBJ,
*        SELECTORCOUNT,
*        SELECTOR,
*        INTATTRCOUNT,
*        INTATTRS,
*        CHARATTRLENGTH,
*        CHARATTRS,
*        COMPCODE,
*        REASON),
*       VL,MF=(E,CALLLIST)
*   LA  R0,MQCC_OK    Load expected compcode
*   C   R0,COMPCODE  Was inquire successful?
*
*   :

```

Appendix E. PL/I examples

The use of PL/I is supported by MQSeries for AIX, OS/2 Warp, OS/390, VSE/ESA, and Windows NT only.

The examples demonstrate the following techniques:

- “Connecting to a queue manager” on page 526
- “Disconnecting from a queue manager” on page 526
- “Creating a dynamic queue” on page 527
- “Opening an existing queue” on page 528
- “Closing a queue” on page 529
- “Putting a message using MQPUT” on page 529
- “Putting a message using MQPUT1” on page 531
- “Getting a message” on page 532
- “Getting a message using the wait option” on page 533
- “Getting a message using signaling” on page 534
- “Inquiring about the attributes of an object” on page 537
- “Setting the attributes of a queue” on page 538

Connecting to a queue manager

This example demonstrates how to use the MQCONN call to connect a program to a queue manager in OS/390 batch. This extract is not taken from the sample applications supplied with MQSeries.

```
%INCLUDE SYSLIB(CMQP);
%INCLUDE SYSLIB(CMQEPP);
:
/*****
/* STRUCTURE BASED ON PARAMETER INPUT AREA (PARAM) */
*****/
DCL 1 INPUT_PARAM      BASED(ADDR(PARAM)),
    2 PARAM_LENGTH    FIXED BIN(15),
    2 PARAM_MQMNAME    CHAR(48);
:
/*****
/* WORKING STORAGE DECLARATIONS */
*****/
DCL MQMNAME            CHAR(48);
DCL COMPCODE          BINARY FIXED (31);
DCL REASON            BINARY FIXED (31);
DCL HCONN             BINARY FIXED (31);
:
/*****
/* COPY QUEUE MANAGER NAME PARAMETER */
/* TO LOCAL STORAGE */
*****/
MQMNAME = ' ';
MQMNAME = SUBSTR(PARAM_MQMNAME,1,PARAM_LENGTH);
:
/*****
/* CONNECT FROM THE QUEUE MANAGER */
*****/
CALL MQCONN (MQMNAME, /* MQM SYSTEM NAME */
            HCONN, /* CONNECTION HANDLE */
            COMPCODE, /* COMPLETION CODE */
            REASON); /* REASON CODE */
:
/*****
/* TEST THE COMPLETION CODE OF THE CONNECT CALL. */
/* IF THE CALL HAS FAILED ISSUE AN ERROR MESSAGE */
/* SHOWING THE COMPLETION CODE AND THE REASON CODE. */
*****/
IF COMPCODE /= MQCC_OK
  THEN DO;
:
      CALL ERROR_ROUTINE;
END;
```

Disconnecting from a queue manager

This example demonstrates how to use the MQDISC call to disconnect a program from a queue manager in OS/390 batch. This extract is not taken from the sample applications supplied with MQSeries.

```
%INCLUDE SYSLIB(CMQP);
%INCLUDE SYSLIB(CMQEPP);
:
/*****
/* WORKING STORAGE DECLARATIONS */
*****/
```

PL/I: Disconnecting from a queue manager

```
DCL COMPCODE          BINARY FIXED (31);
DCL REASON            BINARY FIXED (31);
DCL HCONN             BINARY FIXED (31);
:
/*****
/* DISCONNECT FROM THE QUEUE MANAGER          */
/*****
CALL MQDISC (HCONN,      /* CONNECTION HANDLE    */
             COMPCODE,   /* COMPLETION CODE   */
             REASON);    /* REASON CODE       */

/*****
/* TEST THE COMPLETION CODE OF THE DISCONNECT CALL.          */
/* IF THE CALL HAS FAILED ISSUE AN ERROR MESSAGE             */
/* SHOWING THE COMPLETION CODE AND THE REASON CODE.          */
/*****
IF COMPCODE /= MQCC_OK
THEN DO;
:
CALL ERROR_ROUTINE;
END;
```

Creating a dynamic queue

This example demonstrates how to use the MQOPEN call to create a dynamic queue. This extract is not taken from the sample applications supplied with MQSeries.

```
%INCLUDE SYSLIB(CMQP);
%INCLUDE SYSLIB(CMQEPP);
:
/*****
/* WORKING STORAGE DECLARATIONS          */
/*****
DCL COMPCODE          BINARY FIXED (31);
DCL REASON            BINARY FIXED (31);
DCL HCONN             BINARY FIXED (31);
DCL HOBJ              BINARY FIXED (31);
DCL OPTIONS           BINARY FIXED (31);
:
DCL MODEL_QUEUE_NAME CHAR(48) INIT('PL1.REPLY.MODEL');
DCL DYNAMIC_NAME_PREFIX CHAR(48) INIT('PL1.TEMPQ.*');
DCL DYNAMIC_QUEUE_NAME CHAR(48) INIT(' ');
:
/*****
/* LOCAL COPY OF OBJECT DESCRIPTOR          */
/*****
DCL 1 LMQOD LIKE MQOD;
:
/*****
/* SET UP OBJECT DESCRIPTOR FOR OPEN OF REPLY QUEUE          */
/*****
LMQOD.OBJECTTYPE =MQOT_Q;
LMQOD.OBJECTNAME = MODEL_QUEUE_NAME;
LMQOD.DYNAMICQNAME = DYNAMIC_NAME_PREFIX;
OPTIONS = MQOO_INPUT_EXCLUSIVE;

CALL MQOPEN (HCONN,
             LMQOD,
             OPTIONS,
             HOBJ,
             COMPCODE,
```

PL/I: Creating a dynamic queue

```
REASON);

/*****
/* TEST THE COMPLETION CODE OF THE OPEN CALL.          */
/* IF THE CALL HAS FAILED ISSUE AN ERROR MESSAGE      */
/* SHOWING THE COMPLETION CODE AND THE REASON CODE.    */
/* IF THE CALL HAS SUCCEEDED THEN EXTRACT THE NAME OF */
/* THE NEWLY CREATED DYNAMIC QUEUE FROM THE OBJECT    */
/* DESCRIPTOR.                                        */
*****/
    IF COMPCODE = MQCC_OK
        THEN DO;

    :
        CALL ERROR_ROUTINE;
    END;
    ELSE
        DYNAMIC_QUEUE_NAME = LMQOD_OBJECTNAME;
```

Opening an existing queue

This example demonstrates how to use the MQOPEN call to open an existing queue. This extract is not taken from the sample applications supplied with MQSeries.

```
%INCLUDE SYSLIB(CMQP);
%INCLUDE SYSLIB(CMQEPP);
:
/*****
/* WORKING STORAGE DECLARATIONS                      */
*****/
DCL COMPCODE      BINARY FIXED (31);
DCL REASON        BINARY FIXED (31);
DCL HCONN         BINARY FIXED (31);
DCL HOBJ          BINARY FIXED (31);
DCL OPTIONS       BINARY FIXED (31);
:
DCL QUEUE_NAME    CHAR(48) INIT('PL1.LOCAL.QUEUE');
:
/*****
/* LOCAL COPY OF OBJECT DESCRIPTOR                   */
*****/
DCL 1 LMQOD LIKE MQOD;
:
/*****
/* SET UP OBJECT DESCRIPTOR FOR OPEN OF REPLY QUEUE */
*****/
LMQOD.OBJECTTYPE = MQOT_Q;
LMQOD.OBJECTNAME = QUEUE_NAME;
OPTIONS = MQOO_INPUT_EXCLUSIVE;

CALL MQOPEN (HCONN,
             LMQOD,
             OPTIONS,
             HOBJ,
             COMPCODE,
             REASON);

/*****
/* TEST THE COMPLETION CODE OF THE OPEN CALL.          */
/* IF THE CALL HAS FAILED ISSUE AN ERROR MESSAGE      */
/* SHOWING THE COMPLETION CODE AND THE REASON CODE.    */
*****/
    IF COMPCODE = MQCC_OK
```

```

THEN DO;
:
    CALL ERROR_ROUTINE;
END;

```

Closing a queue

This example demonstrates how to use the MQCLOSE call. This extract is not taken from the sample applications supplied with MQSeries.

```

%INCLUDE SYSLIB(CMQP);
%INCLUDE SYSLIB(CMQEPP);
:
/*****
/* WORKING STORAGE DECLARATIONS */
*****/
DCL COMPCODE          BINARY FIXED (31);
DCL REASON            BINARY FIXED (31);
DCL HCONN             BINARY FIXED (31);
DCL HOBJ              BINARY FIXED (31);
DCL OPTIONS           BINARY FIXED (31);
:
/*****
/* SET CLOSE OPTIONS */
*****/
OPTIONS=MQCO_NONE;

/*****
/* CLOSE QUEUE */
*****/
CALL MQCLOSE (HCONN, /* CONNECTION HANDLE */
              HOBJ,  /* OBJECT HANDLE */
              OPTIONS, /* CLOSE OPTIONS */
              COMPCODE, /* COMPLETION CODE */
              REASON); /* REASON CODE */

/*****
/* TEST THE COMPLETION CODE OF THE CLOSE CALL. */
/* IF THE CALL HAS FAILED ISSUE AN ERROR MESSAGE */
/* SHOWING THE COMPLETION CODE AND THE REASON CODE. */
*****/
IF COMPCODE ^= MQCC_OK
THEN DO;
:
    CALL ERROR_ROUTINE;
END;

```

Putting a message using MQPUT

This example demonstrates how to use the MQPUT call using context. This extract is not taken from the sample applications supplied with MQSeries.

```

%INCLUDE SYSLIB(CMQP);
%INCLUDE SYSLIB(CMQEPP);
:
/*****
/* WORKING STORAGE DECLARATIONS */
*****/
DCL COMPCODE          BINARY FIXED (31);
DCL REASON            BINARY FIXED (31);

```

PL/I: Putting using MQPUT

```

DCL HCONN          BINARY FIXED (31);
DCL HOBJ           BINARY FIXED (31);
DCL OPTIONS        BINARY FIXED (31);
DCL BUFFLEN        BINARY FIXED (31);
DCL BUFFER          CHAR(80);
:
DCL PL1_TEST_MESSAGE CHAR(80)
INIT('***** THIS IS A TEST MESSAGE *****');
:
*****/
/* LOCAL COPY OF MESSAGE DESCRIPTOR          */
/* AND PUT MESSAGE OPTIONS                    */
*****/
DCL 1 LMQMD LIKE MQMD;
DCL 1 LMQPMO LIKE MQPMO;
:
/*****/
/* SET UP MESSAGE DESCRIPTOR                  */
/*****/
LMQMD.MSGTYPE = MQMT_DATAGRAM;
LMQMD.PRIORITY = 1;
LMQMD.PERSISTENCE = MQPER_PERSISTENT;
LMQMD.REPLYTOQ = ' ';
LMQMD.REPLYTOQMGR = ' ';
LMQMD.MSGID = MQMI_NONE;
LMQMD.CORRELID = MQCI_NONE;

/*****/
/* SET UP PUT MESSAGE OPTIONS                  */
/*****/
LMQPMO.OPTIONS = MQPMO_NO_SYNCPOINT;

/*****/
/* SET UP LENGTH OF MESSAGE BUFFER AND THE MESSAGE */
/*****/
BUFFLEN = LENGTH(BUFFER);
BUFFER = PL1_TEST_MESSAGE;
/*****/
/*
*/
/* HCONN WAS SET BY PREVIOUS MQCONN REQUEST.      */
/* HOBJ WAS SET BY PREVIOUS MQOPEN REQUEST.      */
/*
*/
/*****/
CALL MQPUT (HCONN,
           HOBJ,
           LMQMD,
           LMQPMO,
           BUFFLEN,
           BUFFER,
           COMPCODE,
           REASON);

/*****/
/* TEST THE COMPLETION CODE OF THE PUT CALL.      */
/* IF THE CALL HAS FAILED ISSUE AN ERROR MESSAGE  */
/* SHOWING THE COMPLETION CODE AND THE REASON CODE. */
/*****/
IF COMPCODE /= MQCC_OK
THEN DO;

:
CALL ERROR_ROUTINE;
END;

```

Putting a message using MQPUT1

This example demonstrates how to use the MQPUT1 call. This extract is not taken from the sample applications supplied with MQSeries.

```

%INCLUDE SYSLIB(CMQEPP);
%INCLUDE SYSLIB(CMQP);
:
/*****
/* WORKING STORAGE DECLARATIONS */
*****/
DCL COMPCODE          BINARY FIXED (31);
DCL REASON            BINARY FIXED (31);
DCL HCONN             BINARY FIXED (31);
DCL OPTIONS           BINARY FIXED (31);
DCL BUFFLEN           BINARY FIXED (31);
DCL BUFFER            CHAR(80);
:
DCL REPLY_TO_QUEUE    CHAR(48) INIT('PL1.REPLY.QUEUE');
DCL QUEUE_NAME        CHAR(48) INIT('PL1.LOCAL.QUEUE');
DCL PL1_TEST_MESSAGE  CHAR(80)
    INIT('***** THIS IS ANOTHER TEST MESSAGE *****');
:
/*****
/* LOCAL COPY OF OBJECT DESCRIPTOR, MESSAGE DESCRIPTOR */
/* AND PUT MESSAGE OPTIONS */
*****/
DCL 1 LMQOD LIKE MQOD;
DCL 1 LMQMD LIKE MQMD;
DCL 1 LMQPMO LIKE MQPMO;
:
/*****
/* SET UP OBJECT DESCRIPTOR AS REQUIRED. */
*****/
LMQOD.OBJECTTYPE = MQOT_Q;
LMQOD.OBJECTNAME = QUEUE_NAME;

/*****
/* SET UP MESSAGE DESCRIPTOR AS REQUIRED. */
*****/
LMQMD.MSGTYPE = MQMT_REQUEST;
LMQMD.PRIORITY = 5;
LMQMD.PERSISTENCE = MQPER_PERSISTENT;
LMQMD.REPLYTOQ = REPLY_TO_QUEUE;
LMQMD.REPLYTOQMGR = 'I';
LMQMD.MSGID = MQMI_NONE;
LMQMD.CORRELID = MQCI_NONE;

/*****
/* SET UP PUT MESSAGE OPTIONS AS REQUIRED */
*****/
LMQPMO.OPTIONS = MQPMO_NO_SYNCPOINT;

/*****
/* SET UP LENGTH OF MESSAGE BUFFER AND THE MESSAGE */
*****/
BUFFLEN = LENGTH(BUFFER);
BUFFER = PL1_TEST_MESSAGE;

CALL MQPUT1 (HCONN,
             LMQOD,
             LMQMD,
             LMQPMO,
             BUFFLEN,
             BUFFER,

```

PL/I: Putting using MQPUT1

```
                COMPCODE,
                REASON);

/*****
/* TEST THE COMPLETION CODE OF THE PUT1 CALL.          */
/* IF THE CALL HAS FAILED ISSUE AN ERROR MESSAGE SHOWING */
/* THE COMPLETION CODE AND THE REASON CODE.          */
*****/
        IF COMPCODE ^= MQCC_OK
            THEN DO;

        :

                CALL ERROR_ROUTINE;
        END;
```

Getting a message

This example demonstrates how to use the MQGET call to remove a message from a queue. This extract is not taken from the sample applications supplied with MQSeries.

```
        %INCLUDE SYSLIB(CMQP);
        %INCLUDE SYSLIB(CMQEPP);
        :

/*****
/* WORKING STORAGE DECLARATIONS          */
*****/
        DCL COMPCODE          BINARY FIXED (31);
        DCL REASON            BINARY FIXED (31);
        DCL HCONN             BINARY FIXED (31);
        DCL HOBJ              BINARY FIXED (31);
        DCL BUFFLEN           BINARY FIXED (31);
        DCL DATALEN          BINARY FIXED (31);
        DCL BUFFER            CHAR(80);

        :

/*****
/* LOCAL COPY OF MESSAGE DESCRIPTOR AND  */
/* GET MESSAGE OPTIONS                   */
*****/
        DCL 1 LMQMD LIKE MQMD;
        DCL 1 LMQGMO LIKE MQGMO;

        :

/*****
/* SET UP MESSAGE DESCRIPTOR AS REQUIRED.  */
/* MSGID AND CORRELID IN MQMD SET TO NULLS SO FIRST */
/* AVAILABLE MESSAGE WILL BE RETRIEVED.  */
*****/
        LMQMD.MSGID = MQMI_NONE;
        LMQMD.CORRELID = MQCI_NONE;

/*****
/* SET UP GET MESSAGE OPTIONS AS REQUIRED.  */
*****/
        LMQGMO.OPTIONS = MQGMO_NO_SYNCPOINT;

/*****
/* SET UP LENGTH OF MESSAGE BUFFER.      */
*****/
        BUFFLEN = LENGTH(BUFFER);

/*****
/*
/* HCONN WAS SET BY PREVIOUS MQCONN REQUEST.
/* HOBJ WAS SET BY PREVIOUS MQOPEN REQUEST.
*****/
```



```

/*                                                                    */
/*****/

    CALL MQGET (HCONN,
                HOBJ,
                LMQMD,
                LMQGMO,
                BUFFERLEN,
                BUFFER,
                DATALEN,
                COMPCODE,
                REASON);

/*****/
/* TEST THE COMPLETION CODE OF THE GET CALL.                          */
/* IF THE CALL HAS FAILED ISSUE AN ERROR MESSAGE                       */
/* SHOWING THE COMPLETION CODE AND THE REASON CODE.                    */
/*****/
    IF COMPCODE /= MQCC_OK
    THEN DO;
        :
        :
        :
        CALL ERROR_ROUTINE;
    END;

```

Getting a message using the wait option

This example demonstrates how to use the MQGET call with the wait option and accepting truncated messages. This extract is not taken from the sample applications supplied with MQSeries.

```

%INCLUDE SYSLIB(CMQP);
%INCLUDE SYSLIB(CMQEPP);
:
/*****/
/* WORKING STORAGE DECLARATIONS                                       */
/*****/
    DCL COMPCODE                BINARY FIXED (31);
    DCL REASON                  BINARY FIXED (31);
    DCL HCONN                   BINARY FIXED (31);
    DCL HOBJ                     BINARY FIXED (31);
    DCL BUFFLEN                 BINARY FIXED (31);
    DCL DATALEN               BINARY FIXED (31);
    DCL BUFFER                  CHAR(80);

    :

/*****/
/* LOCAL COPY OF MESSAGE DESCRIPTOR AND GET MESSAGE                   */
/* OPTIONS                                                              */
/*****/
    DCL 1 LMQMD LIKE MQMD;
    DCL 1 LMQGMO LIKE MQGMO;

    :

/*****/
/* SET UP MESSAGE DESCRIPTOR AS REQUIRED.                               */
/* MSGID AND CORRELID IN MQMD SET TO NULLS SO FIRST                  */
/* AVAILABLE MESSAGE WILL BE RETRIEVED.                               */
/*****/
    LMQMD.MSGID = MQMI_NONE;
    LMQMD.CORRELID = MQCI_NONE;

/*****/
/* SET UP GET MESSAGE OPTIONS AS REQUIRED.                              */
/*

```

PL/I: Getting using the wait option

```
/* WAIT INTERVAL SET TO ONE MINUTE. */
/*****
  LMQGMO.OPTIONS = MQGMO_WAIT +
                  MQGMO_ACCEPT_TRUNCATED_MSG +
                  MQGMO_NO_SYNCPOINT;
  LMQGMO.WAITINTERVAL=60000;

/*****
/* SET UP LENGTH OF MESSAGE BUFFER. */
/*****
  BUFFLEN = LENGTH(BUFFER);

/*****
/*
/* HCONN WAS SET BY PREVIOUS MQCONN REQUEST. */
/* HOBJ WAS SET BY PREVIOUS MQOPEN REQUEST. */
/*
/*****

  CALL MQGET (HCONN,
             HOBJ,
             LMQMD,
             LMQGMO,
             BUFFERLEN,
             BUFFER,
             DATALEN,
             COMPCODE,
             REASON);

/*****
/* TEST THE COMPLETION CODE OF THE GET CALL. */
/* TAKE APPROPRIATE ACTION BASED ON COMPLETION CODE AND */
/* REASON CODE. */
/*****

  SELECT(COMPCODE);
    WHEN (MQCC_OK) DO; /* GET WAS SUCCESSFUL */
    :
    END;
    WHEN (MQCC_WARNING) DO;
      IF REASON = MQRC_TRUNCATED_MSG_ACCEPTED
      THEN DO; /* GET WAS SUCCESSFUL */
    :
    END;
    ELSE DO;
    :
    CALL ERROR_ROUTINE;
    END;
    WHEN (MQCC_FAILED) DO;
    :
    CALL ERROR_ROUTINE;
    END;
    OTHERWISE;
  END;
```

Getting a message using signaling

Signaling is available only with MQSeries for OS/390.

PL/I: Getting using signaling

This example demonstrates how to use the MQGET call with signaling. This extract is not taken from the sample applications supplied with MQSeries.

```
%INCLUDE SYSLIB(CMQP);
%INCLUDE SYSLIB(CMQEPP);
:
/*****/
/* WORKING STORAGE DECLARATIONS */
/*****/
DCL COMPCODE          BINARY FIXED (31);
DCL REASON            BINARY FIXED (31);
DCL HCONN             BINARY FIXED (31);
DCL HOBJ              BINARY FIXED (31);
DCL DATALEN         BINARY FIXED (31);
DCL BUFFLEN          BINARY FIXED (31);
DCL BUFFER            CHAR(80);

:
DCL ECB_FIXED          FIXED BIN(31);
DCL 1 ECB_OVERLAY BASED(ADDR(ECB_FIXED)),
    3 ECB_WAIT BIT,
    3 ECB_POSTED BIT,
    3 ECB_FLAG3_8 BIT(6),
    3 ECB_CODE PIC'999';

:
/*****/
/* LOCAL COPY OF MESSAGE DESCRIPTOR AND GET MESSAGE */
/* OPTIONS */
/*****/
DCL 1 LMQMD LIKE MQMD;
DCL 1 LMQGMO LIKE MQGMO;

:
/*****/
/* CLEAR ECB FIELD. */
/*****/
ECB_FIXED = 0;

:
/*****/
/* SET UP MESSAGE DESCRIPTOR AS REQUIRED. */
/* MSGID AND CORRELLID IN MQMD SET TO NULLS SO FIRST */
/* AVAILABLE MESSAGE WILL BE RETRIEVED. */
/*****/
LMQMD.MSGID = MQMI_NONE;
LMQMD.CORRELLID = MQCI_NONE;
/*****/
/* SET UP GET MESSAGE OPTIONS AS REQUIRED. */
/* WAIT INTERVAL SET TO ONE MINUTE. */
/*****/
LMQGMO.OPTIONS = MQGMO_SET_SIGNAL +
                MQGMO_NO_SYNCPOINT;
LMQGMO.WAITINTERVAL=60000;
LMQGMO.SIGNAL1 = ADDR(ECB_FIXED);

/*****/
/* SET UP LENGTH OF MESSAGE BUFFER. */
/* CALL MESSGE RETRIEVAL ROUTINE. */
/*****/
BUFFLEN = LENGTH(BUFFER);
CALL GET_MSG;

/*****/
/* TEST THE COMPLETION CODE OF THE GET CALL. */
/*****/
```

PL/I: Getting using signaling

```
/* TAKE APPROPRIATE ACTION BASED ON COMPLETION CODE AND */
/* REASON CODE. */
/*****

SELECT;
  WHEN ((COMPCODE = MQCC_OK) &
        (REASON = MQCC_NONE)) DO

  :
  :
  CALL MSG_ROUTINE;
  :
  :
  END;
  WHEN ((COMPCODE = MQCC_WARNING) &
        (REASON = MQRC_SIGNAL_REQUEST_ACCEPTED)) DO;

  :
  :
  CALL DO_WORK;
  :
  :
  END;
  WHEN ((COMPCODE = MQCC_FAILED) &
        (REASON = MQRC_SIGNAL_OUTSTANDING)) DO;

  :
  :
  CALL DO_WORK;
  :
  :
  END;
  OTHERWISE DO;          /* FAILURE CASE */
/*****
/* ISSUE AN ERROR MESSAGE SHOWING THE COMPLETION CODE */
/* AND THE REASON CODE. */
/*****

  :
  :
  CALL ERROR_ROUTINE;
  :
  :
  END;
  END;
  :
  :
DO_WORK: PROC;
  :
  :
  IF ECB_POSTED
  THEN DO;
    SELECT(ECB_CODE);
    WHEN(MQEC_MSG_ARRIVED) DO;

  :
  :
  CALL GET_MSG;
  :
  :
  END;
  WHEN(MQEC_WAIT_INTERVAL_EXPIRED) DO;

  :
  :
  CALL NO_MSG;
  :
  :
  END;
  OTHERWISE DO;          /* FAILURE CASE */
```

```

/*****
/* ISSUE AN ERROR MESSAGE SHOWING THE COMPLETION CODE */
/* AND THE REASON CODE. */
/*****

:
        CALL ERROR_ROUTINE;

:
        END;

        END;

        END;

:
END DO_WORK;

GET_MSG: PROC;

/*****
/*
/* HCONN WAS SET BY PREVIOUS MQCONN REQUEST. */
/* HOBJ WAS SET BY PREVIOUS MQOPEN REQUEST. */
/* MD AND GMO SET UP AS REQUIRED. */
/*
/*****

        CALL MQGET (HCONN,
                    HOBJ,
                    LMQMD,
                    LMQGMO,
                    BUFFLEN,
                    BUFFER,
                    DATALEN,
                    COMPCODE,
                    REASON);

END GET_MSG;

NO_MSG: PROC;

:
END NO_MSG;

```

Inquiring about the attributes of an object

This example demonstrates how to use the MQINQ call to inquire about the attributes of a queue. This extract is not taken from the sample applications supplied with MQSeries.

```

        %INCLUDE SYSLIB(CMQP);
        %INCLUDE SYSLIB(CMQEPP);
:
/*****
/* WORKING STORAGE DECLARATIONS */
/*****
        DCL COMPCODE          BINARY FIXED (31);
        DCL REASON           BINARY FIXED (31);
        DCL HCONN            BINARY FIXED (31);
        DCL HOBJ             BINARY FIXED (31);
        DCL OPTIONS          BINARY FIXED (31);
        DCL SELECTORCOUNT   BINARY FIXED (31);

```

PL/I: Inquiring about object attributes

```
DCL INTATTRCOUNT          BINARY FIXED (31);
DCL 1 SELECTOR_TABLE,
    3 SELECTORS(5)         BINARY FIXED (31);
DCL 1 INTATTR_TABLE,
    3 INTATTRS(5)         BINARY FIXED (31);
DCL CHARATTRLLENGTH       BINARY FIXED (31);
DCL CHARATTRS             CHAR(100);

:

/*****
/* SET VARIABLES FOR INQUIRE CALL          */
/* INQUIRE ON THE CURRENT QUEUE DEPTH     */
*****/

SELECTORS(01) = MQIA_CURRENT_Q_DEPTH;

SELECTORCOUNT = 1;
INTATTRCOUNT  = 1;

CHARATTRLLENGTH = 0;
/*****
/*
/* HCONN WAS SET BY PREVIOUS MQCONN REQUEST.
/* HOBJ WAS SET BY PREVIOUS MQOPEN REQUEST.
/*
*****/
CALL MQINQ (HCONN,
           HOBJ,
           SELECTORCOUNT,
           SELECTORS,
           INTATTRCOUNT,
           INTATTRS,
           CHARATTRLLENGTH,
           CHARATTRS,
           COMPCODE,
           REASON);
/*****
/* TEST THE COMPLETION CODE OF THE INQUIRE CALL.
/* IF THE CALL HAS FAILED ISSUE AN ERROR MESSAGE SHOWING
/* THE COMPLETION CODE AND THE REASON CODE.
*****/
IF COMPCODE = MQCC_OK
  THEN DO;

:
      CALL ERROR_ROUTINE;
END;
```

Setting the attributes of a queue

This example demonstrates how to use the MQSET call to change the attributes of a queue. This extract is not taken from the sample applications supplied with MQSeries.

```
%INCLUDE SYSLIB(CMQP);
%INCLUDE SYSLIB(CMQEPP);
:
/*****
/* WORKING STORAGE DECLARATIONS          */
*****/
DCL COMPCODE          BINARY FIXED (31);
DCL REASON            BINARY FIXED (31);
DCL HCONN             BINARY FIXED (31);
DCL HOBJ              BINARY FIXED (31);
```

PL/I: Setting queue attributes

```

DCL OPTIONS                BINARY FIXED (31);
DCL SELECTORCOUNT        BINARY FIXED (31);
DCL INTATTRCOUNT        BINARY FIXED (31);
DCL 1 SELECTOR_TABLE,
   3 SELECTORS(5)          BINARY FIXED (31);
DCL 1 INTATTR_TABLE,
   3 INTATTRS(5)          BINARY FIXED (31);
DCL CHARATTRLENGTH        BINARY FIXED (31);
DCL CHARATTRS              CHAR(100);

:

/*****/
/* SET VARIABLES FOR SET CALL                */
/* SET GET AND PUT INHIBITED                */
/*****/

SELECTORS(01) = MQIA_INHIBIT_GET;
SELECTORS(02) = MQIA_INHIBIT_PUT;

INTATTRS(01) = MQQA_GET_INHIBITED;
INTATTRS(02) = MQQA_PUT_INHIBITED;

SELECTORCOUNT = 2;
INTATTRCOUNT  = 2;

CHARATTRLENGTH = 0;

/*****/
/*                                          */
/* HCONN WAS SET BY PREVIOUS MQCONN REQUEST. */
/* HOBJ WAS SET BY PREVIOUS MQOPEN REQUEST.  */
/*                                          */
/*****/
CALL MQSET (HCONN,
            HOBJ,
            SELECTORCOUNT,
            SELECTORS,
            INTATTRCOUNT,
            INTATTRS,
            CHARATTRLENGTH,
            CHARATTRS,
            COMPCODE,
            REASON);

/*****/
/* TEST THE COMPLETION CODE OF THE SET CALL. */
/* IF THE CALL HAS FAILED ISSUE AN ERROR MESSAGE SHOWING */
/* THE COMPLETION CODE AND THE REASON CODE.          */
/*****/
IF COMPCODE = MQCC_OK
  THEN DO;

:
CALL ERROR_ROUTINE;
END;

```

Appendix F. MQSeries data definition files

MQSeries provides data definition files to assist you with the writing of your applications. Data definition files are also known as:

Language	Data definitions
C	Include files or header files
Visual Basic	Module files
COBOL	Copy files
Assembler	Macros
PL/I	Include files

See “Appendix A. Language compilers and assemblers” on page 471 for the compilers that are supported and suitable for use with these data definition files.

The data definition files to assist with the writing of channel exits are described in the *MQSeries Intercommunication* book.

The data definition files to assist with the writing of installable services exits are described in the *MQSeries Programmable System Management* book.

For data definition files supported on C++, see the *MQSeries Using C++* book.

For data definition files supported on RPG, see the *MQSeries for AS/400, V5.1 Application Programming Reference (ILE RPG)* book.

The names of the data definition files have the prefix CMQ, and a suffix that is determined by the programming language:

Suffix	Language
a	Assembler language
b	Visual Basic
c	C
l	COBOL (without initialized values)
p	PL/I
v	COBOL (with default values set)

Installation library

The name **thlqual** is the high-level qualifier of the installation library on OS/390.

This chapter introduces MQSeries data definition files, under these headings:

- “C language include files” on page 542
- “Visual Basic module files” on page 542
- “COBOL copy files” on page 543
- “System/390 assembler-language macros” on page 546
- “PL/I include files” on page 547

C language include files

C language include files

The MQSeries C include files are listed in Table 49. They are installed in the following directories or libraries:

Platform	Installation directory or library
AIX	/usr/mqm/inc/
AS/400	QMOM/H
Compaq (DIGITAL) OpenVMS	/mqm/inc/
Other UNIX platforms	/opt/mqm/inc/
OS/2 and Windows NT	\mqm\tools\c\include
Windows V2.0	\MQW\INCLUDE
Windows V2.1	\Program Files\MQSeries for Windows\Lib
OS/390	thlqual.SCSQC370
Tandem NSK	\$volume.zmqslib
VSE/ESA	PRD2.MQSERIES

Note: For UNIX platforms (not including Digital OpenVMS), the include files are symbolically linked into /usr/include.

For more information on the structure of directories, see the *MQSeries System Administration* book for MQSeries for AIX, AS/400, HP-UX, OS/2, Sun Solaris, and Windows NT; for other platforms, see the appropriate *System Management Guide*.

Table 49. C include files for MQSeries

File name	Contents
<cmqc.h>	Call prototypes, data types, structures, return codes, and constants
<cmqcf.h> (1, 2)	Definitions for programmable commands
<cmqxc.h>(2)	Definitions for channel exits and data-conversion exits
<cmqzc.h>(2, 3)	Definitions for installable services exits
Notes: The files are protected against multiple declaration, so you can include them many times. 1. MQSeries for Windows does not provide this include file. 2. MQSeries for VSE/ESA does not provide this include file. 3. MQSeries for OS/390 and MQSeries for Windows do not provide this include file. 4. On Tandem NSK filenames cannot contain a period (.) so the header filenames are <cmqch> and so on.	

Visual Basic module files

MQSeries for Windows Version 2.0 provides two Visual Basic module files. They are listed in Table 50 and installed in \MQW\INCLUDE.

Table 50. Visual Basic module files for MQSeries for Windows V2.0

File name	Contents
CMQB3.BAS	Call declarations, data types, and named constants for the 16-bit MQI.(1)
CMQB4.BAS	Call declarations, data types, and named constants for both the 16-bit and 32-bit MQI.(2)

Table 50. Visual Basic module files for MQSeries for Windows V2.0 (continued)

File name	Contents
Notes:	
1. Use this with Microsoft Visual Basic Version 3.	
2. Use this with Microsoft Visual Basic Version 4.	

MQSeries for Windows Version 2.1 provides two Visual Basic module files. They are listed in Table 51 and installed in \Program Files\MQSeries for Windows\Lib.

Table 51. Visual Basic module files for MQSeries for Windows V2.1

File name	Contents
CMQB.BAS	Call declarations, data types, and named constants for the main MQI.
CMQB4.BAS	Call declarations, data types, and named constants for the channel exits.
Note: In a default installation, the form files (.BAS) are supplied in the \Program Files\MQSeries for Windows\Include subdirectory.	

MQSeries for Windows NT, V5.1 provides four Visual Basic module files. They are listed in Table 52 and installed in \Program Files\MQSeries\Tools\Samples\VB\Include.

Table 52. Visual Basic module files for MQSeries for Windows NT, V5.1

File name	Contents
CMQB.BAS	Call declarations, data types, and named constants for the main MQI.
CMQBB.BAS	Call declarations, data types, and named constants for MQAI support.
CMQCFB.BAS	Call declarations, data types, and named constants for PCF support.
CMQXB.BAS	Call declarations, data types, and named constants for the channel exits.

COBOL copy files

For COBOL, MQSeries provides separate copy files containing the named constants, and two copy files for each of the structures. There are two copy files for each structure because each is provided both with and without initial values:

- In the WORKING-STORAGE SECTION of a COBOL program, use the files that initialize the structure fields to default values. These structures are defined in the copy files that have names suffixed with the letter "V" (values).
- In the LINKAGE SECTION of a COBOL program, use the structures without initial values. These structures are defined in copy files that have names suffixed with the letter "L" (linkage).

Copy files containing data and interface definitions for MQSeries for AS/400 Version 5 Release 1 are provided for ILE COBOL programs using prototyped calls to the MQI. The files exist in QMQM/QCBLLESRC with member names that have a suffix of 'L' (for structures without initial values) or a suffix of 'V' (for structures with initial values).

The MQSeries interfaces for OPM COBOL have been created to maintain compatibility with previous releases, BUT DO NOT provide support for new

COBOL copy files

function. Consequently, the copy files provided for OPM COBOL programs are at the MQSeries for AS/400 Version 4 Release 2.1 level.

The MQSeries COBOL copy files are listed in Table 53. They are installed in the following directories:

Platform	Installation directory or library
AIX	/usr/mqm/inc/
Compaq (DIGITAL) OpenVMS	/mqm/inc/
Other UNIX platforms	/opt/mqm/inc/
OS/2 and Windows NT	\mqm\tools\cobol\copybook (for Micro Focus COBOL) \mqm\tools\cobol\copybook\VAcobol (for IBM VisualAge COBOL)
OS/390	thlqual.SCSQCOBC
Tandem NSK	\$volume.zmqslib
VSE/ESA	PRD2.MQSERIES

Notes:

1. For AS/400, they are supplied in the library QMQM:
2. For OPM, they are supplied as members of the file QLBLSRC.
3. For ILE, they are supplied as members of the file QCBLLSRC.
4. For Tandem NSK, all the sections are contained in one ENSCRIBE file CMPCOBOL.

Table 53. COBOL copy files

File name (with initial values)	File name (without initial values)	Contents
CMQBOV (not AS/400)	CMQBOL (not AS/400)	Begin options structure (MQBO)
CMQCFV (OS/390 only)	not applicable	Additional named constants for events and PCF commands
CMQCIHV	CMQCIHL	CICS information header structure
CMQCNOV	CMQCNOL	Connect options structure (MQCNO)
CMQDHV	CMQDHL	Distribution header structure (MQDH)
CMQDLHV	CMQDLHL	Dead-letter (undelivered-message) header structure (MQDLH)
CMQDXPV	CMQDXPL	Data-conversion exit parameter structure (MQDXP)
CMQGMOV	CMQGMOL	Get-message options structure (MQGMO)
CMQIIHV	CMQIIHL	IMS header structure (MQIIH)
CMQMDEV	CMQMDEL	Message descriptor extension structure (MQMDE)
CMQMDV	CMQMDL	Message descriptor structure (MQMD)
CMQODV	CMQODL	Object descriptor structure (MQOD)
CMQORV	CMQORL	Object record structure (MQOR)
CMQPMOV	CMQPMOL	Put-message options structure (MQPMO)
CMQRFHV	CMQRFHL	Rules and formatting header structure

Table 53. COBOL copy files (continued)

File name (with initial values)	File name (without initial values)	Contents
CMQRFH2V	CMQRFH2L	Rules and formatting header 2 structure
CMQRRV	CMQRRL	Response record structure (MQRR)
CMQTMCV	CMQTMCL	Trigger-message structure (character format)
CMQTM2V	CMQTM2L	Trigger-message structure (character format) (MQTMC)
CMQTMV	CMQTML	Trigger-message structure (MQTM)
CMQV	not applicable	Named constants for the MQI
CMQWIHV	CMQWIHL	Work-information header structure
CMQXQHV	CMQXQHL	Transmission-queue header structure (MQXQH)
CMQXV	not applicable	Named constants for exits

Include in your program only those files you need. Do this with one or more COPY statements after a level-01 declaration. This means you can include multiple versions of the structures in a program if necessary. However, note that CMQV is a large file.

Here is an example of COBOL code for including the CMQMDV copy file:

```
01 MQM-MESSAGE-DESCRIPTOR.
   COPY CMQMDV.
```

Each structure declaration begins with a level-10 item; this means you can declare several instances of the structure by coding the level-01 declaration followed by a COPY statement to copy in the remainder of the structure declaration. To refer to the appropriate instance, use the IN keyword.

Here is an example of COBOL code for including two instances of CMQMDV:

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

The structures should be aligned on 4-byte boundaries. If you use the COPY statement to include a structure following an item that is not the level-01 item, try to ensure that the structure is a multiple of 4-bytes from the start of the level-01 item. If you do not do this, you may get a reduction in the performance of your application.

The structures are described in the *MQSeries Application Programming Reference* manual. The descriptions of the field in the structures show the names of fields without a prefix. In COBOL programs you must prefix the field names with the name of the structure followed by a hyphen, as shown in the COBOL declarations. The fields in the structure copy files are prefixed this way.

COBOL copy files

The field names in the declarations in the structure copy files are in uppercase. You can use mixed case or lowercase instead. For example, the field *StrucId* of the MQGMO structure is shown as MQGMO-STRUCID in the COBOL declaration and in the copy file.

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

System/390 assembler-language macros

MQSeries for OS/390 provides two assembler-language macros containing the named constants, and one macro to generate each structure. They are listed in Table 54 and installed in `thlqual.SCSQMACS`.

Table 54. System/390 assembler-language macros

Macro	Contents
CMQA	Values of the return codes for the API calls Constants for filling in the option fields Constants for each object attribute, used by the MQINQ and MQSET calls
CMQCFA	Additional named constants for events and PCF commands
CMQCIHA	CICS information-header structure
CMQDLHA	Definition of the MQDLH structure
CMQDXPA	Definition of the MQDXP structure
CMQGMOA	Definition of the MQGMO structure
CMQIIHA	Definition of the MQIIH structure
CMQMDA	Definition of the MQMD structure
CMQODA	Definition of the MQOD structure
CMQPMOA	Definition of the MQPMO structure
CMQRFHA	Definition of the MQRFH structure
CMQRFH2	Definition of the MQRFH2 structure
CMQTMA	Definition of the MQTM structure
CMQTM2A	Definition of the MQTMC2 structure
CMQWIHA	Work-information header structure
CMQXA	Constants for exits
CMQXPA	Definition of the MQXP structure
CMQXQHA	Definition of the MQXQH structure

These macros are called using code like this:

```
MY_MQMD CMQMDA EXPIRY=0,MSGTYPE=MQMT_DATAGRAM
```

PL/I include files

MQSeries for OS/390, AIX, OS/2 Warp, and Windows NT provide include files that contain all the definitions you need when you write MQSeries applications in PL/I. They are listed in Table 55. They are installed in the following directories:

Platform

Installation directory or library

AIX /usr/mqm/inc/

OS/2 \mqm\tools\pli\include

Windows NT

\Program Files\MQSeries\Tools\PLI\Include

OS/390

thlqual.SCSQPLIC

VSE/ESA

PRD2.MQSERIES

Table 55. PL/I include files

Include file	Contents
CMQCFP(1)	Definitions for programmable commands
CMQEPP	Entry point definitions for the API calls.
CMQP	Definitions of all the constants and return codes, data types and structures, and constants to initialize the structures.
CMQXP(1)	Definitions for channel exits and data-conversion exits on OS/390. Named constants related to PCF on AIX, OS/2 Warp, and Windows NT.
Note:	
1. MQSeries for VSE/ESA does not provide this include file.	

Include these files in your program if you are going to link the MQSeries stub to your program (see “Preparing your program to run” on page 311). Include only CMQP if you intend to link the MQSeries calls dynamically (see “Dynamically calling the MQSeries stub” on page 315). Dynamic linking can be performed for batch and IMS programs only.

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 documents General-use Programming Interface and Associated Guidance Information and Product-sensitive Programming Interface and Associated Guidance Information provided by MQSeries for AIX, V5.1, MQSeries for AS/400, V5.1, MQSeries for AT&T GIS UNIX, V2.2, MQSeries for Compaq (DIGITAL) OpenVMS, V2.2.1.1, MQSeries for Compaq Tru64 UNIX, V5.1, MQSeries for HP-UX, V5.1, MQSeries for OS/2 Warp, V5.1, MQSeries for OS/390, V5.2, MQSeries for SINIX and DC/OSx, V2.2, MQSeries for Sun Solaris, V5.1, MQSeries for Tandem NonStop Kernel, V2.2.0.1, MQSeries for VSE/ESA, V2.1, MQSeries for Windows, V2.0, MQSeries for Windows V2.1, and MQSeries for Windows NT, V5.1.

General-use programming interfaces allow the customer to write programs that obtain the services of these products.

Programming interface information

General-use Programming Interface and Associated Guidance Information is identified where it occurs, by an introductory statement to a chapter or section.

Product-sensitive programming interfaces allow the customer installation to perform tasks such as diagnosing, modifying, monitoring, repairing, tailoring, or tuning of these products. Use of such interfaces creates dependencies on the detailed design or implementation of the IBM software product. Product-sensitive programming interfaces should be used only for these specialized purposes. Because of their dependencies on detailed design and implementation, it is to be expected that programs written to such interfaces may need to be changed in order to run with new product releases or versions, or as a result of service.

Product-sensitive Programming Interface and Associated Guidance Information is identified where it occurs, by an introductory statement to a chapter or section.

Trademarks

The following terms are trademarks of International Business Machines Corporation in the United States, other countries, or both:

AD/Cycle	AIX	AS/400
BookManager	C/370	C/400
CICS	CICS/ESA	COBOL/400
Common User Access	DB2	First Failure Support Technology
FFST	IBM	IBMLink
IMS	IMS/ESA	Integrated Language Environment
Language Environment	MQSeries	MVS/ESA
OS/2	OS/390	OS/400
OpenEdition	RACF	SAA
RPG/400	SP2	SupportPac
System/390	TXSeries	VM/ESA
VSE/ESA	VisualAge	

| Intel is a registered trademark of Intel Corporation in the United States, other
| countries, or both.

| ActiveX, Microsoft, Visual Basic, Visual C++, Windows, and Windows NT are
| registered trademarks of Microsoft Corporation in the United States, other
| countries, or both.

Java is a registered trademark of Sun Microsystems, Inc. in the United States, other
countries, or both.

| UNIX is a registered trademark of The Open Group in the United States and other
| countries.

Other company, product, and service names may be trademarks or service marks
of others.

Glossary of terms and abbreviations

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

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

A

abend reason code. A 4-byte hexadecimal code that uniquely identifies a problem with MQSeries for 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.

affinity. An association between objects that have some relationship or dependency upon each other.

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-defined format. In message queuing, application data in a message, which has a meaning defined by the user application. Contrast with *built-in format*.

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.

Glossary

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*.

built-in format. In message queuing, application data in a message, which has a meaning defined by the queue manager. Synonymous with *in-built format*. Contrast with *application-defined format*.

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.

CICS transaction. In CICS, a unit of application processing, usually comprising one or more units of work.

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

CL. Control Language.

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

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

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

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

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

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

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

command processor. The MQSeries component that processes commands.

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

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

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

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

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

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

context. Information about the origin of a message.

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

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

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

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

controlled shutdown. See *quiesced shutdown*.

CPF. Command prefix.

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

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

D

DAE. Dump analysis and elimination.

data 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.

data-conversion service. A service that converts application data to the character set and encoding that are required by applications on other platforms.

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.

Glossary

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.

distribution list. A list of queues to which a message can be put using a single MQPUT or MQPUT1 statement.

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

FIFO. First-in-first-out.

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

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*.

format. In message queuing, a term used to identify the nature of application data in a message. See also *built-in format* and *application-defined format*.

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

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

FRR. Functional recovery routine.

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

G

GCPC. Generalized command preprocessor.

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

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

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

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

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

GTF. Generalized Trace Facility.

H

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

I

ILE. Integrated Language Environment.

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

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

in-built format. See *built-in format*.

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

Glossary

another queue manager. In addition, it is used for queue-manager aliasing and reply-to-queue aliasing.

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

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

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

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

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

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

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

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

M

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

MCA. Message channel agent.

MCI. Message channel interface.

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

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

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

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

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

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

message format service (MFS). In IMS, an editing facility that allows application programs to deal with simple logical messages, instead of device-dependent data, thus simplifying the application development process. See *message input descriptor* and *message output descriptor*.

message group. A group of logical messages. Logical grouping of messages allows applications to group messages that are similar and to ensure the sequence of the messages.

message input descriptor (MID). In IMS, the MFS control block that describes the format of the data presented to the application program. Contrast with *message output descriptor*.

message output descriptor (MOD). In IMS, the MFS control block that describes the format of the output data produced by the application program. Contrast with *message input descriptor*.

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 segment. One of a number of segments of a message that is too large either for the application or for the queue manager to handle.

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*.

MFS. Message format service.

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

MQAI. MQSeries Administration Interface.

MQI. Message queue interface.

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

MQSC. MQSeries commands.

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

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

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

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

N

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

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

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

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

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

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

NSI. Name service interface.

NTFS. New Technology File System.

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

O

OAM. Object authority manager.

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

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

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

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

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

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

OPM. Original Program Model.

Glossary

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

OTMA. Open Transaction Manager Access.

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

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

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

P

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

PCF. Programmable command format.

PCF command. See *programmable command format*.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Contrast with *MQSC*.

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

PTF. Program temporary fix.

Q

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

queue manager. A system program that provides queuing services to applications. It provides an application programming interface so that programs

can access messages on the queues that the queue manager owns. See also *local queue manager* and *remote queue manager*. An MQSeries object that defines the attributes of a particular queue manager.

queue manager event. An event that indicates:

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

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

queuing. See *message queuing*.

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

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

R

RBA. Relative byte address.

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

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

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

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

reference message. A message that refers to a piece of data that is to be transmitted. The reference message is handled by message exit programs, which attach and

detach the data from the message so allowing the data to be transmitted without having to be stored on any queues.

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.

Glossary

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.

| **shared inbound channel.** In MQSeries for OS/390, a
| channel that was started by a listener using the group
| port. The channel definition of a shared channel can be
| stored either on page set zero (private) or in the shared
| repository (global).

| **shared outbound channel.** In MQSeries for OS/390, a
| channel that moves messages from a shared
| transmission queue. The channel definition of a shared
| channel can be stored either on page set zero (private)
| or in the shared repository (global).

| **shared queue.** In MQSeries for OS/390, a type of local
| queue. The messages on the queue are stored in the
| *coupling facility* and can be accessed by one or more
| queue managers in a *queue-sharing group*. The definition
| of the queue is stored in the *shared repository*.

shared repository. In MQSeries for OS/390, a shared DB2 database that is used to hold object definitions that have been defined globally.

shutdown. See *immediate shutdown*, *preemptive shutdown*, and *quiesced shutdown*.

signaling. In MQSeries for OS/390 and MQSeries for Windows 2.1, a feature that allows the operating system to notify a program when an expected message arrives on a queue.

single logging. A method of recording MQSeries for OS/390 activity where each change is recorded on one data set only. Contrast with *dual logging*.

single-phase backout. A method in which an action in progress must not be allowed to finish, and all changes that are part of that action must be undone.

single-phase commit. A method in which a program can commit updates to a queue without coordinating those updates with updates the program has made to resources controlled by another resource manager. Contrast with *two-phase commit*.

SIT. System initialization table.

stanza. A group of lines in a configuration file that assigns a value to a parameter modifying the behavior of a queue manager, client, or channel. In MQSeries on UNIX systems, MQSeries for OS/2 Warp, and MQSeries for Windows NT, a configuration (.ini) file may contain a number of stanzas.

storage class. In MQSeries for OS/390, a storage class defines the page set that is to hold the messages for a particular queue. The storage class is specified when the queue is defined.

store and forward. The temporary storing of packets, messages, or frames in a data network before they are retransmitted toward their destination.

subsystem. In OS/390, a group of modules that provides function that is dependent on OS/390. For example, MQSeries for OS/390 is an OS/390 subsystem.

supervisor call (SVC). An OS/390 instruction that interrupts a running program and passes control to the supervisor so that it can perform the specific service indicated by the instruction.

SVC. Supervisor call.

switch profile. In MQSeries for OS/390, a RACF profile used when MQSeries starts up or when a refresh security command is issued. Each switch profile that MQSeries detects turns off checking for the specified resource.

symptom string. Diagnostic information displayed in a structured format designed for searching the IBM software support database.

synchronous messaging. A method of communication between programs in which programs place messages on message queues. With synchronous messaging, the sending program waits for a reply to its message before resuming its own processing. Contrast with *asynchronous messaging*.

syncpoint. An intermediate or end point during processing of a transaction at which the transaction's protected resources are consistent. At a syncpoint, changes to the resources can safely be committed, or they can be backed out to the previous syncpoint.

System Authorization Facility (SAF). An OS/390 facility through which MQSeries for OS/390 communicates with an external security manager such as RACF.

system.command.input queue. A local queue on which application programs can put MQSeries commands. The commands are retrieved from the queue by the command server, which validates them and passes them to the command processor to be run.

system control commands. Commands used to manipulate platform-specific entities such as buffer pools, storage classes, and page sets.

system diagnostic work area (SDWA). Data recorded in a SYS1.LOGREC entry, which describes a program or hardware error.

system initialization table (SIT). A table containing parameters used by CICS on start up.

SYS1.LOGREC. A service aid containing information about program and hardware errors.

T

target library high-level qualifier (thlqual). High-level qualifier for OS/390 target data set names.

task control block (TCB). An OS/390 control block used to communicate information about tasks within an address space that are connected to an OS/390 subsystem such as MQSeries for OS/390 or CICS.

task switching. The overlapping of I/O operations and processing between several tasks. In MQSeries for OS/390, the task switcher optimizes performance by allowing some MQI calls to be executed under subtasks rather than under the main CICS TCB.

TCB. Task control block.

TCP/IP. Transmission Control Protocol/Internet Protocol.

Glossary

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*.

teraspace. In MQSeries for AS/400, a form of shared memory introduced in OS/400 V4R4.

termination notification. A pending event that is activated when a CICS subsystem successfully connects to MQSeries for OS/390.

thlqual. Target library high-level qualifier.

thread. In MQSeries, the lowest level of parallel execution available on an operating system platform.

time-independent messaging. See *asynchronous messaging*.

TMI. Trigger monitor interface.

trace. In MQSeries, a facility for recording MQSeries activity. The destinations for trace entries can include GTF and the system management facility (SMF).

tranid. See *transaction identifier*.

transaction. See *unit of work* and *CICS transaction*.

transaction identifier. In CICS, a name that is specified when the transaction is defined, and that is used to invoke the transaction.

transaction manager. A software unit that coordinates the activities of resource managers by managing global transactions and coordinating the decision to commit them or roll them back. V5.1 of MQSeries for AIX, Compaq Tru64 UNIX, HP-UX, OS/2 Warp, Sun Solaris, and Windows NT is a transaction manager.

Transmission Control Protocol/Internet Protocol (TCP/IP). A suite of communication protocols that support peer-to-peer connectivity functions for both local and wide area networks.

transmission program. See *message channel agent*.

transmission queue. A local queue on which prepared messages destined for a remote queue manager are temporarily stored.

trigger event. An event (such as a message arriving on a queue) that causes a queue manager to create a trigger message on an initiation queue.

triggering. In MQSeries, a facility allowing a queue manager to start an application automatically when predetermined conditions on a queue are satisfied.

trigger message. A message containing information about the program that a trigger monitor is to start.

trigger monitor. A continuously-running application serving one or more initiation queues. When a trigger message arrives on an initiation queue, the trigger monitor retrieves the message. It uses the information in the trigger message to start a process that serves the queue on which a trigger event occurred.

trigger monitor interface (TMI). The MQSeries interface to which customer- or vendor-written trigger monitor programs must conform. A part of the MQSeries Framework.

two-phase commit. A protocol for the coordination of changes to recoverable resources when more than one resource manager is used by a single transaction. Contrast with *single-phase commit*.

U

UIS. User identifier service.

undelivered-message queue. See *dead-letter queue*.

undo/redo record. A log record used in recovery. The redo part of the record describes a change to be made to an MQSeries object. The undo part describes how to back out the change if the work is not committed.

unit of recovery. A recoverable sequence of operations within a single resource manager. Contrast with *unit of work*.

unit of work. A recoverable sequence of operations performed by an application between two points of consistency. A unit of work begins when a transaction starts or after a user-requested syncpoint. It ends either at a user-requested syncpoint or at the end of a transaction. Contrast with *unit of recovery*.

user identifier service (UIS). In MQSeries for OS/2 Warp, the facility that allows MQI applications to associate a user ID, other than the default user ID, with MQSeries messages.

utility. In MQSeries, a supplied set of programs that provide the system operator or system administrator with facilities in addition to those provided by the MQSeries commands. Some utilities invoke more than one function.

X

X/Open XA. The X/Open Distributed Transaction Processing XA interface. A proposed standard for distributed transaction communication. The standard specifies a bidirectional interface between resource managers that provide access to shared resources within transactions, and between a transaction service that monitors and resolves transactions.

XCF. Cross Systems Coupling Facility.

Bibliography

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

MQSeries cross-platform publications

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

- MQSeries for AIX, V5.1
- MQSeries for AS/400, V5.1
- MQSeries for AT&T GIS UNIX, V2.2
- MQSeries for Compaq (DIGITAL) OpenVMS, V2.2.1.1
- MQSeries for Compaq Tru64 UNIX, V5.1
- MQSeries for HP-UX, V5.1
- MQSeries for OS/2 Warp, V5.1
- MQSeries for OS/390, V5.2
- MQSeries for SINIX and DC/OSx, V2.2
- MQSeries for Sun Solaris, V5.1
- MQSeries for Sun Solaris, Intel Platform Edition, V5.1
- MQSeries for Tandem NonStop Kernel, V2.2.0.1
- MQSeries for VSE/ESA, V2.1
- MQSeries for Windows, V2.0
- MQSeries for Windows, V2.1
- MQSeries for Windows NT, V5.1

The MQSeries cross-platform publications are:

- *MQSeries Brochure*, G511-1908
- *An Introduction to Messaging and Queuing*, GC33-0805
- *MQSeries Intercommunication*, SC33-1872
- *MQSeries Queue Manager Clusters*, SC34-5349
- *MQSeries Clients*, GC33-1632
- *MQSeries System Administration*, SC33-1873
- *MQSeries MQSC Command Reference*, SC33-1369
- *MQSeries Event Monitoring*, SC34-5760
- *MQSeries Programmable System Management*, SC33-1482
- *MQSeries Administration Interface Programming Guide and Reference*, SC34-5390
- *MQSeries Messages*, GC33-1876
- *MQSeries Application Programming Guide*, SC33-0807

- *MQSeries Application Programming Reference*, SC33-1673
- *MQSeries Programming Interfaces Reference Summary*, SX33-6095
- *MQSeries Using C++*, SC33-1877
- *MQSeries Using Java*, SC34-5456
- *MQSeries Application Messaging Interface*, SC34-5604

MQSeries platform-specific publications

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

MQSeries for AIX, V5.1

MQSeries for AIX Quick Beginnings, GC33-1867

MQSeries for AS/400, V5.1

MQSeries for AS/400 Quick Beginnings, GC34-5557

MQSeries for AS/400 System Administration, SC34-5558

MQSeries for AS/400 Application Programming Reference (ILE RPG), SC34-5559

MQSeries for AT&T GIS UNIX, V2.2

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

MQSeries for Compaq (DIGITAL) OpenVMS, V2.2.1.1

MQSeries for Digital OpenVMS System Management Guide, GC33-1791

MQSeries for Compaq Tru64 UNIX, V5.1

MQSeries for Compaq Tru64 UNIX Quick Beginnings, GC34-5684

MQSeries for HP-UX, V5.1

MQSeries for HP-UX Quick Beginnings, GC33-1869

MQSeries for OS/2 Warp, V5.1

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

Bibliography

MQSeries for OS/390, V5.2

MQSeries for OS/390 Concepts and Planning Guide, GC34-5650

MQSeries for OS/390 System Setup Guide, SC34-5651

MQSeries for OS/390 System Administration Guide, SC34-5652

MQSeries for OS/390 Problem Determination Guide, GC34-5892

MQSeries for OS/390 Messages and Codes, GC34-5891

MQSeries for OS/390 Licensed Program Specifications, GC34-5893

MQSeries for OS/390 Program Directory

MQSeries link for R/3, Version 1.2

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

MQSeries for SINIX and DC/OSx, V2.2

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

MQSeries for Sun Solaris, V5.1

MQSeries for Sun Solaris Quick Beginnings, GC33-1870

MQSeries for Sun Solaris, Intel Platform Edition, V5.1

MQSeries for Sun Solaris, Intel Platform Edition Quick Beginnings, GC34-5851

MQSeries for Tandem NonStop Kernel, V2.2.0.1

MQSeries for Tandem NonStop Kernel System Management Guide, GC33-1893

MQSeries for VSE/ESA, V2.1

MQSeries for VSE/ESA, Version 2 Release 1 Licensed Program Specifications, GC34-5365

MQSeries for VSE/ESA System Management Guide, GC34-5364

MQSeries for Windows, V2.0

MQSeries for Windows User's Guide, GC33-1822

MQSeries for Windows, V2.1

MQSeries for Windows User's Guide, GC33-1965

MQSeries for Windows NT, V5.1

MQSeries for Windows NT Quick Beginnings, GC34-5389

MQSeries for Windows NT Using the Component Object Model Interface, SC34-5387

MQSeries LotusScript Extension, SC34-5404

Softcopy books

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

HTML format

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

- MQSeries for AIX, V5.1
- MQSeries for AS/400, V5.1
- MQSeries for Compaq Tru64 UNIX, V5.1
- MQSeries for HP-UX, V5.1
- MQSeries for OS/2 Warp, V5.1
- MQSeries for OS/390, V5.2
- MQSeries for Sun Solaris, V5.1
- MQSeries for Windows NT, V5.1 (compiled HTML)
- MQSeries link for R/3, V1.2

The MQSeries books are also available in HTML format from the MQSeries product family Web site at:

<http://www.ibm.com/software/mqseries/>

Portable Document Format (PDF)

PDF files can be viewed and printed using the Adobe Acrobat Reader.

If you need to obtain the Adobe Acrobat Reader, or would like up-to-date information about the platforms on which the Acrobat Reader is supported, visit the Adobe Systems Inc. Web site at:

<http://www.adobe.com/>

PDF versions of relevant MQSeries books are supplied with these MQSeries products:

- MQSeries for AIX, V5.1
- MQSeries for AS/400, V5.1
- MQSeries for Compaq Tru64 UNIX, V5.1
- MQSeries for HP-UX, V5.1
- MQSeries for OS/2 Warp, V5.1
- MQSeries for OS/390, V5.2
- MQSeries for Sun Solaris, V5.1
- MQSeries for Windows NT, V5.1
- MQSeries link for R/3, V1.2

PDF versions of all current MQSeries books are also available from the MQSeries product family Web site at:

<http://www.ibm.com/software/mqseries/>

BookManager® format

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

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

PostScript format

The MQSeries library is provided in PostScript (.PS) format with many MQSeries Version 2 products. Books in PostScript format can be printed on a PostScript printer or viewed with a suitable viewer.

Windows Help format

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

MQSeries information available on the Internet

The MQSeries product family Web site is at:

<http://www.ibm.com/software/mqseries/>

By following links from this Web site you can:

- Obtain latest information about the MQSeries product family.
- Access the MQSeries books in HTML and PDF formats.
- Download MQSeries SupportPacs.

Related publications

This section describes the documentation available for some related products and issues mentioned in this book.

CICS

For information about those aspects of CICS Transaction Server for OS/390 that this book refers to, see the following books:

CICS Application Programming Reference, SC33-1688

CICS Application Programming Guide, SC33-1687

CICS Customization Guide, SC33-1683

CICS Messages and Codes, GC33-1694

CICS-Supplied Transactions, SC33-1686

CICS System Definition Guide, SC33-1682

CICS Internet and External Interfaces Guide, SC33-1944. This is the book name for CICS Transaction Server V1.2.

CICS External Interfaces Guide, SC33-1944. This is the book name for CICS Transaction Server V1.3.

CICS Transaction Server for OS/390 Version 1 Release 3: Web Support and 3270 Bridge, SG24-5480. This is a Redbook.

For information about those aspects of CICS for MVS/ESA Version 4.1 that this book refers to, see the following books:

CICS for MVS/ESA V4.1 Application Programming Reference, SC33-1170

CICS for MVS/ESA V4.1 Customization Guide, SC33-1165

CICS for MVS/ESA V4.1 CICS-Supplied Transactions, SC33-1168

CICS for MVS/ESA V4.1 System Definition Guide, SC33-1164

For information about CICS programming on other platforms, see the following books:

CICS on Open Systems Application Programming Guide, SC33-1568-00

CICS for OS/2 V2.0.1 Application Programming, SC33-0883

Transaction Server for OS/2 Warp, V4 Application Programming, SC33-1585

CICS for AS/400 Application Programming Guide, SC33-1386

CICS for Windows NT V2.0 Application Programming, SC33-1425

Transaction Server for Windows NT, V4 Application Programming Guide, SC33-1888

IMS

For information about those aspects of IMS that this book refers to, see the following books:

IMS/ESA Version 4 Application Programming: DL/I Calls, SC26-3062

Related publications

IMS/ESA Version 4 Application Programming: Design Guide, SC26-3066
IMS/ESA Version 5 Application Programming: Database Manager, SC26-8015
IMS/ESA Version 5 Application Programming: Design Guide, SC26-8016
IMS/ESA Version 5 Application Programming: Transaction Manager, SC26-8017
IMS/ESA Version 5 Open Transaction Manager Access Guide, SC26-8026
| *IMS/ESA V6 OTMA Guide and Reference*,
| SC26-8743

MVS/ESA

For information about those aspects of MVS/ESA that this book refers to, see the following book:
MVS/ESA Application Development Guide: Assembler Language Programs, GC28-1644

Design

For information on how to design panel-driven application interfaces, see the following book:
Systems Application Architecture, Common User Access: Basic Interface Design Guide, SC26-4583

C

For information about C programming, see the following books:
Guide to Tools for Programming in C, U6296-J-Z145-2-7600
SNI Programmer's Reference Manual, U6401-J-Z145-3-7600
OS/390 C/C++ Programming Guide, SC09-2362
| *ILE C for AS/400 Programmer's Guide*,
| SC09-2712

C++

For information about C++ programming, see the following books:
C Set++ for AIX: User's Guide, SC09-1968
VisualAge C++ for OS/2 User's Guide, S25H-6961
VisualAge C++ for OS/2 Programming Guide, S25H-6958
VisualAge for C++ for Windows User's Guide, S33H-5031
VisualAge for C++ for Windows Programming Guide, S33H 5032
VisualAge for C++ for AS/400 : C++ User's Guide, SC09-2416
OS/390 C/C++ Programming Guide, SC09-2362

COBOL

For information about COBOL programming that this book refers to, see the following books:
COBOL V3.2 SINIX pocket guide, U21709-J-Z145-2-7600
IBM COBOL Set for AIX Programming Guide, SC26-8423
IBM COBOL for MVS and VM, IBM VisualAge for COBOL for OS/2, IBM COBOL Set for AIX Language Reference, SC26-4769

LDAP

For information about LDAP, see the following redbook:
Understanding LDAP, SG24-4986

RPG

| For information about RPG programming, see the
| following books:
| *ILE RPG for AS/400 Programmer's Guide*,
| SC09-2507
| *ILE RPG for AS/400 Reference*, SC09-2508

Index

Special Characters

runmqtmc trigger monitor 207
runmqtrm trigger monitor
error detection 211
how to run 206

A

abend
AEY9 218
QLOP 218
accounting using message context 37
AccountingToken field 37
adapter
batch 215
CICS 216
IMS 219
trace points 218
ADS
in CICS 3270 bridge vectors 245
terminology with the CICS
bridge 262
used in CICS 3270 bridge vectors 241
AEY9 abend 218
alias queue
examples of when to use 44
overview 42
resolving queue name 101
alias queue manager definition 99
alternate PCB, IMS bridge 273
alternate user authority 104
AlternateUserId field 104
AMI (Application Messaging Interface) 17, 65
AMQ0ECHA sample program 397
amq0gbr0 sample program 360, 376
amq0get0 sample program 360, 378
AMQ0GET4 sample program 378
amq0put0 sample program 360, 373
amq0req0 sample program 360, 388
AMQ0REQ4 sample program 388
AMQ0SETA sample program 395
amqiech2 sample program 362, 397
amqiechx sample program 360, 397
amqiinq2 sample program 362, 394
amqiinqx sample program 360, 394
amqiset2 sample program 362, 395
amqisetx sample program 360, 395
amqltmc0 trigger monitor 207
amqmehc2 sample program 362, 397
amqmehcx sample program 360, 397
amqminq2 sample program 362, 394
amqminqx sample program 360, 394
amqmset2 sample program 362, 395
amqmsetx sample program 360, 395
amqrgрма sample program 380
AMQSAMP4 sample program 372
amqsbcg sample program 378
amqsbcg0 sample program 360, 378
amqsbcgx sample program 378

amqscic0 sample transaction 360, 407
amqscic21 sample transaction 360
amqsdliq sample program 360
amqsech sample program 397
amqsecha sample program 360, 397
AMQSECHA sample program 397
amqsechc sample program 397
AMQSERV4 sample program 206, 398, 399
amqsgbr sample program 376
amqsgbr0 sample program 360, 376
AMQSGBR4 sample program 376
amqsgbrc sample program 376
amqsgget0 sample program 360, 378
AMQSGGET4 sample program 378
amqsggetc sample program 360, 378
amqsggetw sample program 360, 378
amqsinqa sample program 394
amqsinqc sample program 394
amqsprma sample program 380
amqsptf0 sample program 375
amqsput0 sample program 360, 373
AMQSPUT4 sample program 373
amqsputc sample program 360, 373
amqsputw sample program 360, 373
amqsreq sample program 388
amqsreq0 sample program 360, 388
AMQSREQ4 sample program 388
amqsreqc sample program 388
amqsset sample program 395
amqsseta sample program 395
AMQSSETA sample program 395
amqssetc sample program 395
amqstrg sample program 398
amqstrg0 sample program 360, 398
AMQSTRG4 sample program 206, 398
amqstrgc sample program 398
amqstxgx sample program 360, 407
amqstxpx sample program 360, 407
amqstxsx sample program 360, 407
amqsvfc0 sample program 362, 398
AMQSVFC4 sample program 365, 398
amqsvfcx sample program 398
amqsvfx sample program 398
amqsvf0 sample transaction 360
amqsvf0.sqb sample 400
amqsvf0.sqc sample 400
amqsvf0.sq sample transaction 360, 418
amqsvf0.sqb sample 400
amqsvf0.sqc sample 400
amqsvag0.c sample 400
amqsvag0.cbl sample 400
amqsvas0.sqb sample 400
amqsvas0.sqc sample 400
amqsvrma sample program 380
amqzsc (XA switch load module) 190
amqzsc21 (XA switch load module) 190
amqzsc (XA switch load module) 190
API (Application Programming Interface)
calls 66
dealing with failure of a call 53
API-crossing exit for OS/390 221

application data 23
Application Data Structure
in CICS 3270 bridge vectors 245
terminology with the CICS
bridge 262
used in CICS 3270 bridge vectors 241
application design
for more than one platform 20
overview 13
Application Messaging Interface (AMI) 17, 65
application programming
application takeover 226
connection tag 227
dynamic queues 227
for the CICS 3270 bridge 237
for the CICS DPL bridge 233
index queues 227
maximum message size 227
migrating applications to use shared
queues 228
persistent messages 227
queue-sharing groups 226
serialized applications 226
application queue 191
applications
debugging with the CICI bridge 260
with the CICS bridge 231
applications, building
in C 299, 300
in C++ 301, 302
in COBOL 303
in Java 304
applications, writing 275
ApplIdentityData field 37
ApplOriginData field 37
assembler language
assemblers supported 471
examples
MQCLOSE 514
MQCONN 510
MQDISC 511
MQGET 517
MQGET with signaling 520
MQGET with wait option 519
MQINQ 522
MQOPEN for dynamic queue 511
MQOPEN for existing queue 513
MQPUT 514
MQPUT1 516
MQSET 522
macros 546
preparing your program to run 311
support for 80
using constants and structures 546
using the MQI 80
attributes
DefInputOpenOption 103
DefPriority 32
HardenGetBackout 35, 55
IndexType 134

- attributes (*continued*)
 - inquiring about 173
 - MaxMsgLength 111, 126
 - MaxPriority 32
 - MsgDeliverySequence 35, 126
 - ProcessName 197
 - queue manager 39
 - queues 43
 - selectors 173
 - setting 173
 - Shareability 103
 - TrigData 197
 - TriggerControl 202
 - TriggerData 192
 - TriggerDepth 203
 - TriggerInterval 204
 - TriggerMsgPriority 202
 - TriggerType 202
- audit trail using message context 36
- authority checking
 - alternate user authority on
 - MQOPEN 104
 - by MQCLOSE 96
 - by MQDISC 96
 - by MQOPEN 97
- automatically starting an application
 - an example 390
 - how triggering works 191
 - introduction 17

B

- backing out changes 35, 177
- backout, skipping 144
- BackoutCount field 35, 55
- base queue 44
- Basic
 - compilers supported 471
- Basic Mapping Support
 - with the CICS 3270 bridge 239
- batch for MQSeries for OS/390
 - adapter 215
 - building an application 312
 - calling the stub dynamically 315
 - restrictions 84
 - support for 214
- benefits of message queuing 9
- bibliography 565
- binding
 - FASTPATH 92
 - STANDARD 92
- BMS
 - with the CICS 3270 bridge 239
- BookManager 567
- bridge request queue
 - setting options with the CICS
 - bridge 256
- BRMQ
 - inbound structure with the CICS 3270
 - bridge 235
 - outbound structure with the CICS
 - 3270 bridge 236
- browse cursor 102, 149
- browsing (sample for MQSeries for OS/390) 437
- browsing messages 149
- browsing messages in logical order 151

- Buffer parameter 111
- BufferLength parameter 125
- building applications
 - in C 299, 300
 - in C++ 301, 302
 - in COBOL 303
 - in Java 304
- building Tuxedo applications 415
- building your application
 - batch with MQSeries for OS/390 312
 - CICS and MQSeries for OS/390 313
 - IMS 314
 - on AIX 285
 - on AS/400 291
 - on Digital OpenVMS 297
 - on HP-UX 305
 - on NCR UNIX SVR4 MP-RAS,
 - R3.0 295
 - on OS/2 Warp 323
 - on OS/390 311
 - on SINIX or DC/OSx 327
 - on Sun Solaris 331
 - on Tandem NSK 335
 - on VSE/ESA 339
 - on Windows 341
 - on Windows NT 343
- built-in formats 31

C

- C++
 - support for 76
- C++ language
 - compilers supported 471
- C++ language include files
 - <cmqc.h> 301, 302
 - <imqi.hpp> 301, 302
 - <imqtype.h> 301, 302
- C++ programs, compiling 302, 303
- C++ sample programs 359
- C language
 - compilers supported 471
 - examples
 - MQCLOSE 479
 - MQCONN 476
 - MQDISC 476
 - MQGET 482
 - MQGET with signaling 484
 - MQGET with wait option 483
 - MQINQ 486
 - MQOPEN for dynamic queue 477
 - MQOPEN for existing queue 478
 - MQPUT 479
 - MQPUT1 480
 - MQSET 487
 - include files 542
 - support for 77
- C language include files
 - <cmqc.h> 299, 300, 542
 - <cmqfc.h> 299, 300, 542
 - <cmqxc.h> 299, 300, 542
 - <cmqzc.h> 299, 300, 542
 - header files 542
 - include files 542
- C programs, compiling 299, 301
- call interface 66
- calling dynamically with MQSeries for OS/390 315
- CAM (credit application manager) 455
- CCSID (Coded Character Set Identifier)
 - same as queue manager 30
- CEDF (CICS Execution Diagnostic Facility) 82, 219
- CEMT I TASK
 - example with the CICS 3270
 - bridge 238
- CETR (CICS Trace Control transaction) 320
- CF (coupling facility) 6, 40
- channel
 - data-conversion exit 155
- channel queue 43
- CICS
 - adapter 216
 - API=crossing exit 221
 - assembler language applications 80
 - calling the stub dynamically with
 - MQSeries for OS/390 315
 - COBOL applications 288, 345, 346
 - CSQCAPX 221
 - debugging programs 320
 - Execution Diagnostic Facility 82
 - MQSeries for OS/390 support 214
 - on AS/400 293
 - preparing C programs 289, 334
 - sample transaction for MQSeries for
 - AIX 289, 308
 - sample transaction for MQSeries for
 - SINIX and DC/OSx 329
 - sample transaction for MQSeries for
 - Sun Solaris 334
 - storage protection facility 223
 - trace 320
 - Trace Control transaction 320
 - with MQSeries for AIX 288
 - with MQSeries for HP-UX 308
 - with MQSeries for OS/2 Warp 324
 - with MQSeries for OS/390 313
 - with MQSeries for SINIX and
 - DC/OSx 329
 - with MQSeries for Sun Solaris 333
 - with MQSeries for Windows NT 344
- CICS 3270 bridge
 - 3270 legacy applications 250
 - Application Data Structure
 - (ADS) 241, 245
 - application programming 237
 - Basic Mapping Support (BMS) 239
 - CEMT I TASK example 238
 - inbound BRMQ structure 235
 - inbound message structure 235
 - interpreting RECEIVE MAP
 - vectors 245
 - interpreting SEND MAP vectors 241
 - managing units of work 239
 - message structure 235
 - optimized emulation example 252
 - outbound BRMQ structure 236
 - outbound message structure 236
 - transactions 234
 - transactions in the distributed
 - environment 249
 - transactions with start data 249
 - unoptimized emulation example 251
 - using vectors 235

- CICS adapter
 - abends 218
 - QLOP abend 218
 - trace points 218
 - using CEDF 219
- CICS bridge
 - Application Data Structure (ADS) 231
 - Application Data Structure terminology 262
 - applications on OS/390 231
 - COMMAREA data 231
 - debugging applications 260
 - distributed programming 249
 - DPL programs 231
 - error handling 258
 - legacy applications 231
 - managing MsgId and CorrelId 256
 - setting bridge request queue options 256
 - setting MQCIH fields 253
 - setting MQMD fields 253
 - unit of work 256
 - using DPL programs 232
- CICS DPL bridge
 - application programming 233
 - COMMAREA data 232
 - managing units of work 234
 - message structure 232
 - transactions in the distributed environment 250
- CICS Execution Diagnostic Facility (CEDF) 219
- CICS for OS/2 user exit 15 186
- CICS sample transaction 407
- CKQC transaction 89, 320
- CKTI transaction 206, 211
- client (MQSeries)
 - LU 6.2 link library 307
 - MQSeries clients and servers 9
 - triggering support 191
 - what it is 6
- cluster
 - what it is 5
- cluster queue
 - MQOPEN option 102
 - overview 42
- clusters (message affinities)
 - MQSeries techniques 17
- COBOL
 - CICS applications 288, 345, 346
 - compilers supported 471
 - copy files 543
 - examples
 - MQCLOSE 496
 - MQCONN 492
 - MQDISC 492
 - MQGET 499
 - MQGET with signaling 502
 - MQGET with wait option 501
 - MQINQ 505
 - MQOPEN for dynamic queue 493
 - MQOPEN for existing queue 494
 - MQPUT 496
 - MQPUT1 498
 - MQSET 506
 - LITLINK directive 325, 328, 346
 - COBOL (*continued*)
 - on AIX 286
 - on AS/400 292
 - on HP-UX 307
 - on OS/2 325
 - on SINIX or DC/OSx 328
 - on Sun Solaris 332
 - on Windows NT 345
 - support for 79
 - using named constants 79
 - COBOL programs, compiling 303
 - Coded Character Set Identifier (CCSID)
 - same as queue manager 30
 - coded character sets 31
 - CodedCharSetId (CCSID) 30
 - message data 30
 - COMMAREA data
 - with the CICS DPL bridge 232
 - commit
 - single-phase 178
 - two-phase 178
 - committing changes 177
 - communication
 - connectionless 7
 - time-independent 8
 - compilers supported 471
 - compiling
 - for MQSeries for AIX 285
 - for MQSeries for AS/400 291
 - for MQSeries for AT&T GIS UNIX 295
 - for MQSeries for Compaq (DIGITAL) OpenVMS 297
 - for MQSeries for HP-UX 305
 - for MQSeries for OS/2 Warp 323
 - for MQSeries for OS/390 311
 - for MQSeries for SINIX and DC/OSx 327
 - for MQSeries for Sun Solaris 331
 - for MQSeries for Windows 341
 - for MQSeries for Windows NT 343
 - compiling C++ programs 302, 303
 - compiling C programs 299, 301
 - compiling COBOL programs 303
 - compiling Java programs 304
 - completion code 75
 - confirmation of arrival (COA) report 25
 - confirmation of delivery (COD) report 25
 - connecting to a queue manager 90, 92
 - connection handle
 - returned from MQCONN 91
 - returned from MQCONNX 92
 - using with MQGET 122
 - what it is 75
 - connection tag 227
 - connectionless communication 7
 - constants in COBOL 79
 - context
 - default 112
 - identity 37
 - message 36
 - MQOPEN options 103
 - MQPUT options 112
 - origin 37
 - context (Credit Check sample application) 461
 - Context field 109
 - control information 24
 - convert characters call 158
 - convert message data
 - MQGET 123, 147
 - copy files
 - how to use them 543
 - copying messages 149
 - correlation identifier 35
 - CorrelId
 - managing with the CICS bridge 256
 - CorrelId field 35, 133
 - coupling facility (CF) 6, 40
 - creating conversion-exit code 158
 - credit application manager (CAM) 455
 - credit check sample (MQSeries for OS/390) 451
 - crtmqcvx 158
 - CRTMQCVX 158
 - CSQ4BAA1 sample 437
 - CSQ4BCA1 sample 437
 - CSQ4BVA1 sample 437
 - CSQ4CAC1 sample 443
 - CSQ4CCB5 sample 459
 - CSQ4CCC1 sample 443
 - CSQ4CCG1 sample 439
 - CSQ4CVB1 sample 455
 - CSQ4CVB2 sample 455
 - CSQ4CVB3 sample 458
 - CSQ4CVB4 sample 458
 - CSQ4CVB5 sample 459
 - CSQ4CVC1 sample 443
 - CSQ4CVD1 sample 448
 - CSQ4CVD2 sample 449
 - CSQ4CVD3 sample 449
 - CSQ4CVD4 sample 450
 - CSQ4CVD5 sample 451
 - CSQ4ICB3 sample 463
 - CSQ4TCD1 sample 448
 - CSQ4TCD2 sample 449
 - CSQ4TCD4 sample 450
 - CSQ4TCD5 sample 451
 - CSQ4TVD1 sample 448
 - CSQ4TVD2 sample 449
 - CSQ4TVD4 sample 450
 - CSQ4TVD5 sample 451
 - CSQCAPX sample 221
 - CSQQTRMN transaction 206, 211
 - cursor, browse 102, 149
 - CVTMQMDTA 158
- D**
- data
 - application 23
 - message 23, 111
- data conversion
 - amqsvfc0 sample program 398
 - AMQSVFC4 sample program 398
 - amqsvfcx sample program 398
 - application 31
 - convert characters call 158
 - convert MQSeries Data Type command 158
 - create MQSeries conversion-exit command 158
 - IMS bridge 274
 - interface 155

- data conversion (*continued*)
 - message 147
 - MQGET 123, 147
 - MQXCNV call 68
 - OS/390 considerations 156
 - UNIX environment 167
- data-conversion exit 155, 157
 - amqsvfc0 sample program 398
 - AMQSVFC4 sample program 398
 - amqsvfcx sample program 398
 - convert characters call 158
 - convert MQSeries Data Type
 - command 158
 - create MQSeries conversion-exit
 - command 158
 - IMS bridge 274
 - invoking 156
 - MQXCNV call 68
 - skeleton 157
 - UNIX environment 167
 - writing
 - AS/400 161
 - Digital OpenVMS 166
 - OS/2 162
 - OS/390 164
 - Tandem NSK systems 165
 - UNIX systems 166
 - Windows NT 171
- data conversion interface (DCI) 155
- data definition files 69
 - copy files 541
 - header files 541
 - include files 541
 - macros 541
- data in a message 23, 111
- data types
 - elementary 68
 - structures 68
- datagram 24
- DataLength parameter 125
- date and time of messages 38
- DCE sample exits 359
- dead-letter (undelivered-message) queue
 - handler 58
 - overview 48
 - sample to deal with messages on
 - it 419
 - use within MQSeries for OS/390
 - sample 460
 - using 57
- dead-letter queue
 - shared 228
- dead-letter queue handler
 - brief description 58
 - sample 419
- DeadLetterQName field 197
- debugging applications
 - with the CICS bridge 260
- debugging programs 319
- default context 112
- defining alias for queue manager 99
- DefInputOpenOption attribute 103
- DefPriority attribute 32
- design considerations
 - performance hints and tips 18
- disconnecting from a queue manager 95

- distributed programming
 - CICS bridge 249
- distribution lists 115
 - identifying 116
 - opening 116
 - putting messages to 118
 - using the MQPMR structure 119
- DOS, no triggering support 191
- DOS clients
 - compilers supported 471
- DPL programs 231
- dynamic linking of MQI calls for
 - MQSeries for OS/390 315
- dynamic queue
 - closing temporary queue 105
 - creating 104
 - overview 45
 - permanent queue properties 46
 - temporary queue properties 45
 - when to use 46
- dynamic queues, shared queues 227
- dynamic XA resource management
 - structure 185
- DynamicQName field 104

E

- EBCDIC newline character
 - conversion 148
- ECB (event control block) 143
- emulation
 - example with the CICS 3270
 - bridge 251, 252
- Encina sample transaction 418
- Encoding field 30
- environments for MQSeries for
 - OS/390 213
- error handling
 - with the CICS bridge 258
- errors
 - dead-letter (undelivered-message)
 - queue 57
 - dealing with failure of a call 53
 - incorrect message data 55
 - report message 55
 - system interruptions 53
 - undelivered-message queue 57
- event control block 143
- event-driven processing 8
- event queue 43
- examples
 - assembler language
 - MQCLOSE 514
 - MQCONN 510
 - MQDISC 511
 - MQGET 517
 - MQGET with signaling 520
 - MQGET with wait option 519
 - MQINQ 522
 - MQOPEN for dynamic queue 511
 - MQOPEN for existing queue 513
 - MQPUT 514
 - MQPUT1 516
 - MQSET 522

C

- MQCLOSE 479
- MQCONN 476
- MQDISC 476

examples (*continued*)

C (*continued*)

- MQGET 482
- MQGET with signaling 484
- MQGET with wait option 483
- MQINQ 486
- MQOPEN for dynamic queue 477
- MQOPEN for existing queue 478
- MQPUT 479
- MQPUT1 480
- MQSET 487
- COBOL
 - MQCLOSE 496
 - MQCONN 492
 - MQDISC 492
 - MQGET 499
 - MQGET with signaling 502
 - MQGET with wait option 501
 - MQINQ 505
 - MQOPEN for dynamic queue 493
 - MQOPEN for existing queue 494
 - MQPUT 496
 - MQPUT1 498
 - MQSET 506
- PL/I
 - MQCLOSE 529
 - MQCONN 526
 - MQDISC 526
 - MQGET 532
 - MQGET with signaling 535
 - MQGET with wait option 533
 - MQINQ 537
 - MQOPEN for dynamic queue 527
 - MQOPEN for existing queue 528
 - MQPUT 529
 - MQPUT1 531
 - MQSET 538
- exception report 25
- exclusive access to a queue 102
- Execution Diagnostic Facility 82
- execution key of CICS programs 223
- exit programs 221
 - data conversion 157
- expiry report 25
- external syncpoint
 - coordination 184
 - interfaces 185
 - restrictions 186
 - X/Open XA interface 185

F

- FASTPATH binding 92
- feedback codes, IMS bridge 271
- Feedback field 27
- fields
 - AlternateUserId 104
 - AppIdentityData 37
 - AppOriginData 37
 - BackoutCount 35, 55
 - Context 109
 - CorrelId 35, 133
 - DeadLetterQName 197
 - DynamicQName 104
 - Encoding 30
 - Feedback 27
 - Format 30

fields (*continued*)

- GroupId
 - match options 133
 - MQMO 133
- InitiationQName 196
- MsgId 133
- Persistence 34
- Priority 32
- PutApplName 37
- PutApplType 37
- PutDate 37
- PutMsgRecFields 110
- PutMsgRecOffset 110
- PutMsgRecPtr 110
- PutTime 37
- RecsPresent 109
- ReplyToQ 36
- ReplyToQMGr 36
- Report 25
- ResolvedQMGrName 109
- ResolvedQName 109
- ResponseRecOffset 110
- ResponseRecPtr 110
- StrucId 108
- UserIdentifier 37
- Version 109
- WaitInterval 124, 141

format

- control information 30
- message data 30

Format field 30

formats

- built-in 31
- user-defined 31

G

get (sample for MQSeries for OS/390) 435

get-message options structure 122

getting

- a particular message 133
- message from triggered queue 205
- message when the length is unknown 150
- messages 121
- options 121

glossary 553

GMT (Greenwich Mean Time) 38

group

- identifier 35

group attach 10

group batch attach 10

GroupStatus field

- MQGMO structure 125

H

handle

- scope of connection 91
- scope of connection handle 98
- scope of object handle 98
- using 75
- using object handle 97

HardenGetBackout attribute 35, 55

heap size

- for OS/2 MQI client 324
- MQSeries for Windows NT 344

HTML (Hypertext Markup Language) 566

Hypertext Markup Language (HTML) 566

I

identity context 37

IGQ (intra-group queuing)

- what it is 6

IMS

- adapter 219
- building an MQSeries for OS/390 application 314
- calling the stub dynamically with MQSeries for OS/390 315
- closing objects 97
- enquiry application (IMS) 268
- mapping MQSeries messages to transactions 270
- support for 214
- using MQI calls 265
- using syncpoints 265
- writing a server application 266
- writing an enquiry application 268
- writing MQSeries applications 265

IMS bridge

- alternate PCB 273
- data conversion 274
- feedback codes 271
- IMS commands 269
- LLZZ data segment 274
- mapping MQSeries messages to transactions 270
- message segmentation 274
- NAK 269
- reply messages 273
- sense codes 271
- undelivered messages 269
- writing applications 269

IMS commands, IMS bridge 270

include files

- PL/I for MQSeries for OS/390 547

increasing MaxMsgLength 135

index queues

- shared queues 227

initiation queue 47

- example to create one 196
- what it is 193

initiation queue, shared 228

InitiationQName field 196

inquiring about attributes

- MQSeries for AS/400 sample program 394
- MQSeries for OS/2 Warp sample program 394
- MQSeries for OS/390 sample 443
- MQSeries for UNIX sample program 394
- MQSeries for Windows NT sample program 394
- using MQINQ 173

interfaces to external syncpoint managers 185

internal syncpoint coordination 183

intra-group queuing (IGQ)

- what it is 6

intra group queuing agent 47

invoking data-conversion exit 156

J

Java programs, compiling 304

JCL (Job Control Language)

- batch 312
- CICS and MQSeries for OS/390 313
- IMS 314

L

languages 76

languages supported 471

large messages

- reference messages 135
- segmented messages 135

LDAP (lightweight directory access protocol) 349

legacy applications

- with the CICS 3270 bridge 250
- with the CICS bridge 231

libraries, linking

- in C 300, 301
- in C++ 302, 303
- in COBOL 304

libraries to use

- with MQSeries for AIX 285
- with MQSeries for AT&T GIS UNIX 295
- with MQSeries for Compaq (DIGITAL) OpenVMS 297
- with MQSeries for HP-UX 306
- with MQSeries for OS/2 Warp 323
- with MQSeries for SINIX and DC/OSx 330
- with MQSeries for Sun Solaris 332
- with MQSeries for Windows 341
- with MQSeries for Windows NT 343

library files 69

libsna.a 307

libsna stubs.a 307

lightweight directory access protocol (LDAP) 349

linking

- for MQSeries for AIX 285
- for MQSeries for AS/400 291
- for MQSeries for AT&T GIS UNIX 295
- for MQSeries for Compaq (DIGITAL) OpenVMS 297
- for MQSeries for HP-UX 305
- for MQSeries for OS/2 Warp 323
- for MQSeries for OS/390 311
- for MQSeries for SINIX and DC/OSx 327
- for MQSeries for Sun Solaris 331
- for MQSeries for Windows 341
- for MQSeries for Windows NT 343

linking in the MQI client environment

- when using LU 6.2 307

linking libraries

- in C 300, 301
- in C++ 302, 303

- linking libraries (*continued*)
 - in COBOL 304
- LLZZ data segment, IMS bridge 274
- local queue 41
- looking at a message 149

M

- macros, assembler language 546
- mail manager sample application (MQSeries for OS/390) 444
- MatchOptions field
 - MQGMO structure 125
- maximum message length
 - increasing 135
- maximum message size, shared queues 227
- MaxMsgLength attribute 111, 126
- MaxPriority attribute 32
- MCA (message channel agent), definition of 4
- message
 - backed out 35
 - browsing 149
 - browsing and removing 151
 - browsing in logical order 151
 - browsing when message length unknown 150
 - channel agent definition 4
 - confirm arrival 25
 - confirm delivery 25
 - context
 - MQOPEN options 103
 - MQPUT options 112
 - types 36
 - copying 149
 - creating 23
 - data 23, 111
 - data conversion
 - considerations 31
 - MQGET 147
 - data format 30
 - datagram 24
 - definition 4
 - descriptor
 - MQMD structure 24
 - when using MQGET 122
 - when using MQPUT 108
 - design 15
 - exception 25
 - expiry 25
 - getting 121
 - getting a particular 133
 - greater than 4 MB 135
 - groups 32
 - identifier 35
 - logical ordering 126
 - looking at 149
 - maximum size 111
 - negative action notification 26
 - notification of arrival 142
 - order of retrieval from a queue 126
 - originator information 37
 - persistence 34
 - persistence and triggers 209
 - physical ordering 126
 - positive action notification 26
 - priority 32, 126

- message (*continued*)
 - priority and triggers 209
 - problem delivering 56
 - putting 107
 - putting one 113
 - reference 139
 - removing after browsing 151
 - reply 25
 - reply, IMS bridge 273
 - report 25, 55
 - request 25
 - retry sending 56
 - return to sender 57
 - sample to deal with those on
 - dead-letter queue 419
 - segmentation 136
 - segmented 33
 - selecting from a queue 34
 - signaling 142
 - size 111
 - structure 23
 - trigger 192, 209
 - trigger after queue manager
 - restart 209
 - trigger format 209
 - type for status information 25
 - type when no reply required 24
 - types 24
 - undeliverable, IMS bridge 270
 - undelivered 56
 - undelivered, sample to handle 419
 - use of types 24
 - waiting for 141
- message affinities (clusters)
 - MQSeries techniques 17
- message channel agent (MCA), definition of 4
- message context (Credit Check sample application) 461
- message data conversion, MQGET 123, 147
- message handler sample (MQSeries for OS/390) 464
- message queue, definition of 4
- Message Queue Interface 17
 - calls 66
 - data definition files 69
 - dealing with failure of a call 53
 - elementary data types 68
 - library files 69
 - structures 68
 - stub programs 69
 - using System/390 assembler 80
- message queuing 3
 - benefits of 9
 - features 6
- message segmentation, IMS bridge 274
- message structure
 - for the CICS DPL bridge 232
 - inbound with the CICS 3270
 - bridge 235
 - outbound with the CICS 3270
 - bridge 236
- messages
 - mapping to IMS transaction
 - types 270
 - retrieving in correct order 226

- migrating applications to use shared queues 228
- model queue 45, 104
- MQ*_DEFAULT values
 - with MQSeries for AIX 78
- MQ_MSG_HEADER_LENGTH 112
- MQCA_* values 173
- MQCIH
 - setting fields with the CICS
 - bridge 253
- MQCLOSE
 - authority checking 96
 - call parameters 106
 - closing a queue 105
- MQCLOSE, using the call
 - Assembler example 514
 - C language example 479
 - COBOL example 496
 - PL/I example 529
- MQCMIT 180
- MQCONN
 - call parameters 90
 - scope of 91
- MQCONN, using the call
 - Assembler example 510
 - C language example 476
 - COBOL example 492
 - PL/I example 526
- MQCONNX 92
- MQDH 111
- MQDISC
 - authority checking 96
 - when to use 95
- MQDISC, using the call
 - Assembler example 511
 - C language example 476
 - COBOL example 492
 - PL/I example 526
- MQDLH 57, 111
- MQGET
 - backing out changes 177
 - buffer size 125
 - call parameters 121
 - committing changes 177
 - data conversion 147
 - increase speed of 134
 - message data conversion 123
 - message options 122
 - order of message retrieval 126
 - to get a specific message 133
 - triggered queues 205
 - unknown message length 150
 - using MQGMO 122
 - using MQMD 122
 - when it fails 154
 - when to use 121
- MQGET, using the call
 - Assembler example 517
 - C language example 482
 - COBOL 499
 - PL/I example 532
- MQGET, using the call with signaling
 - Assembler example 520
 - C language example 484
 - COBOL example 502
 - PLO/I example 535

MQGET, using the call with the wait option
 Assembler example 519
 C language example 483
 COBOL example 501
 PL/I example 533

MQGMO 122

MQGMO_*
 ACCEPT_TRUNCATED_MSG 125

MQGMO_BROWSE_* 335
 FIRST 149
 MSG_UNDER_CURSOR 150
 NEXT 149

MQGMO_CONVERT 147

MQGMO_MARK_SKIP_BACKOUT 55
 explanation 144

MQGMO_MSG_UNDER_CURSOR 151

MQGMO_WAIT 141

MQI (Message Queue Interface)
 calls 66
 client library files 69
 data definition files 69
 dealing with failure of a call 53
 elementary data types 68
 IMS applications 265
 library files 69
 overview 17
 structures 68
 stub programs 69
 using System/390 assembler 80

MQI client
 LU 6.2 link library 307

MQIA_* values 173

MQIIH 269

MQINQ
 call parameters 174
 use of selectors 173
 when it fails 175

MQINQ, using the call
 C language example 486
 COBOL example 505
 PL/I example 537

MQINQ, using the MQINQ and MQSET calls
 Assembler example 522

MQMD
 overview 24
 setting fields with the CICS bridge 253
 when using MQGET 122
 when using MQPUT 108

MQMT_* values 24

MQOD 99

MQOO_* values 102

MQOPEN
 browse cursor 149
 call parameters 98
 MQOO_* values 102
 object handle 97
 using MQOD 99
 using options parameter 102

MQOPEN, using the call to create a dynamic queue
 Assembler example 511
 C language example 477
 COBOL example 493
 PL/I example 527

MQOPEN, using the call to open an existing queue
 Assembler example 513
 C language example 478
 COBOL example 494
 PL/I example 528

MQPMO 108

MQPUT
 backing out changes 177
 call parameters 107
 committing changes 177
 context information 112
 if it fails 120
 quiescing queue manager 109
 syncpointing 109
 using MQPMO 108

MQPUT, using the call
 Assembler example 514
 C language example 479
 COBOL example 496
 PL/I example 529

MQPUT1
 call parameters 113
 if it fails 120
 performance 107

MQPUT1, using the call
 Assembler example 516
 C language example 480
 COBOL example 498
 PL/I example 531

MQRC_*
 SECOND_MARK_NOT_ALLOWED 145

MQRMIXASwitch 185

MQRMIXASwitchDynamic 185

MQSeries applications
 planning 13
 testing 20

MQSeries client
 connection to queue manager 91
 using triggering 192
 what it is 6

MQSeries data conversion interface 155

MQSeries for AIX
 amqisetx 395
 amqmsetx 395
 amqsseta 395
 amqzsc 190
 amqzsc21 190
 amqzsca 190
 build TUXEDO server environment 407
 building your application 285
 CICS support 288
 compilers supported 471
 key features 10
 sample programs 359
 scope of MQCONN 91
 set sample 395
 syncpoints 182
 triggering using samples 390
 TUXEDO samples 407
 ubbstxcx.cfg example 411
 XA switch load module 190

MQSeries for AS/400
 AMQZSTUB 291
 building your application 291
 compilers supported 471

MQSeries for AS/400 (continued)
 compiling 291
 CRTCMOD 291
 disconnecting from queue manager 95
 key features 10
 linking 291
 sample program using triggering 392
 SQL programming considerations 294
 syncpoint considerations with CICS for AS/400 182
 syncpoints 182, 187

MQSeries for AT&T GIS UNIX
 amqisetx 395
 amqmsetx 395
 amqsseta 395
 building your application 295
 C compiler 295
 compilers supported 471
 flags when compiling 295
 key features 10
 link libraries 295
 sample programs 359
 set sample 395
 syncpoints 182
 triggering using samples 390
 TUXEDO samples 407
 ubbstxcx.cfg example 411

MQSeries for Compaq (DIGITAL) OpenVMS
 building your application 297
 C compiler 297
 compilers supported 471
 flags when compiling 297
 link libraries 297
 sample programs 359

MQSeries for DIGITAL UNIX (Compaq Tru64 UNIX)
 compilers supported 471
 sample programs 359

MQSeries for HP-UX
 amqisetx 395
 amqmsetx 395
 amqsseta 395
 amqzsc 190
 build TUXEDO server environment 409
 building your application 305
 CICS support 308
 compilers supported 471
 sample programs 359
 scope of MQCONN 91
 set sample 395
 syncpoints 182
 triggering using samples 390
 TUXEDO samples 407
 ubbstxcx.cfg example 411
 XA switch load module 190

MQSeries for OS/2 Warp
 building your application 323
 CICS support 324
 compilers supported 471
 key features 10
 sample programs 359
 scope of MQCONN 91

- MQSeries for OS/2 Warp (*continued*)
 - syncpoints 182
 - Transaction Server support 324, 326
 - triggering using samples 390
 - TZ environment variable 38
- MQSeries for OS/390
 - building your application 311
 - CMQA 546
 - CMQDLHA 546
 - CMQDXPA 546
 - CMQEPP 547
 - CMQGMOA 546
 - CMQIIHA 546
 - CMQMDA 546
 - CMQODA 546
 - CMQP 547
 - CMQPMOA 546
 - CMQTMA 546
 - CMQTMCA 546
 - CMQXA 546
 - CMQXPA 546
 - CMQXQHA 546
 - compilers and assemblers supported 471
 - CSQBSTUB 312
 - CSQCSTUB 313
 - CSQQSTUB 314
 - key features 9
 - requesting no backout of MQGET 144
 - using signaling 142
- MQSeries for SINIX and DC/OSx
 - build TUXEDO server environment 410, 411
 - building your application 327
 - CICS support 329
 - compilers supported 471
 - flags when compiling 327
 - link libraries 330
 - sample programs 359
- MQSeries for Sun Solaris
 - build TUXEDO server environment 408, 411
 - building your application 331
 - C compiler 331
 - CICS support 333
 - compilers supported 471
 - link libraries 332
 - sample programs 359
- MQSeries for Tandem NonStop Kernel
 - building your application 335
 - compilers supported 471
 - notification of message arrival 142
 - sample programs 359
 - syncpoint considerations 188
 - using signaling 142
- MQSeries for VSE/ESA
 - building your application 339
 - compilers supported 471
 - sample programs 359
- MQSeries for Windows
 - building your application 341
 - sample programs 359
 - using signaling 142
- MQSeries for Windows, V2.0
 - compilers supported 471

- MQSeries for Windows, V2.1
 - compilers supported 471
- MQSeries for Windows NT
 - build TUXEDO server environment 412
 - building your application 343
 - CICS support 344
 - compilers supported 471
 - key features 10
 - sample programs 359
 - scope of MQCONN 91
 - syncpoints 182
 - Transaction Server support 344
 - triggering using samples 390
 - TUXEDO sample makefile 414
 - TUXEDO samples 407
 - ubbstxcn.cfg example 413
- MQSeries object
 - closing 105
 - creating 14
 - introduction 14
 - namelist 48
 - naming 49
 - opening 97
 - process definition attributes 48
 - create 196
 - queue 41
 - queue manager 39
 - rules for naming 49
 - storage class 49
 - what it is 39
- MQSeries on UNIX systems
 - key features 10
 - sample programs 359
 - syncpoints 182
 - triggering using samples 390
 - TUXEDO 407
 - ubbstxcx.cfg example 411
- MQSeries publications 565
- MQSeries Workflow 225
- MQSET
 - attribute list 175
 - call parameters 175
 - use of selectors 173
- MQSET, using the call
 - C language example 487
 - COBOL example 506
 - PL/I example 538
- MQSET, using the MQINQ and MQSET calls
 - Assembler example 522
- MQTM 209
- MQTM (trigger message) 207
- MQTMC (trigger message, character) 207
- MQTMC2 (trigger message, character) 207
- MQXCNV data-conversion call 68
- MQXQH 111
- MsgDeliverySequence attribute 35, 126
- MsgId
 - managing with the CICS bridge 256
- MsgId field 133
- MsgToken field
 - MQGMO structure 125

N

- name resolution 50, 99
- namelist
 - attributes 48
 - opening 97
 - rules for naming 49
 - sample application 458
- naming of MQSeries objects 49
- negative action notification (NAN) report 26
- notification of message arrival 142

O

- object
 - closing 105
 - creating 14
 - descriptor 99
 - handle 75
 - introduction 14
 - namelist 48
 - naming 49
 - opening 97
 - process definition 196
 - attributes 48
 - queue 41
 - queue manager 39
 - rules for naming 49
 - storage class 49
 - using handle 97
 - what it is 39
- object-oriented programming (OOP) 279
- OOP (object-oriented programming) 279
- opening an MQSeries object 97
- opening distribution lists
 - identifying distribution lists 116
 - identifying Object Records 116
 - the MQOD structure 116
 - the MQOR structure 116
- Options field
 - MQGMO structure 123
 - MQPMO structure 109
- Options parameter (MQOPEN call) 102
- order of message retrieval 126
- origin context 37
- OS/390
 - batch restrictions 84
 - OpenEdition 220
 - support for 214
 - WLM (workload manager) 225
 - workload manager (WLM) 225
- OTMA sense codes 271

P

- parameters
 - Buffer 111
 - BufferLength 125
 - DataLength 125
 - Options 102
- PDF (Portable Document Format) 566
- performance
 - design hints and tips 18
 - MQGET and buffer size 125
 - MQGET for a particular message 134
 - MQPUT1 107

- performance (*continued*)
 - persistent messages 34
- permanent dynamic queue,
 - properties 46
- Persistence field 34
- persistent messages, shared queues 227
- PL/I
 - CMQEPP 547
 - CMQP 547
 - compilers supported 471
 - examples
 - MQCLOSE 529
 - MQCONN 526
 - MQDISC 526
 - MQGET 532
 - MQGET with signaling 535
 - MQGET with wait option 533
 - MQINQ 537
 - MQOPEN for dynamic queue 527
 - MQOPEN for existing queue 528
 - MQPUT 529
 - MQPUT1 531
 - MQSET 538
 - include files 547
 - on AIX 287
 - on OS/2 326
 - on Windows NT 347
 - support for 83
- planning an MQSeries application 13
- platform support
 - list of 18
- Portable Document Format (PDF) 566
- positive action notification (PAN)
 - report 26
- PostScript format 567
- print message (sample for MQSeries for OS/390) 439
- Priority field 32
- priority in messages 32
- problem delivering a message,
 - overview 35
- problem determination
 - abend codes issued by the CICS adapter 218
 - trace points in CICS adapter 218
 - using CEDF with the CICS adapter 219
- problem determination, use of report message 55
- process definition object
 - attributes 48
 - example to create one 196
 - opening 97
 - rules for naming 49
 - triggering prerequisite 196
 - what it is 192
- ProcessName 207
- ProcessName attribute 197
- programming languages 76
- publications
 - MQSeries 565
- put (sample for MQSeries for OS/390) 432
- put-message options 108
- PutApplName field 37
- PutApplType field 37
- PutDate field 37

- PutMsgRecFields field 110
- PutMsgRecOffset field 110
- PutMsgRecPtr field 110
- PutTime field 37
- putting
 - messages 107
 - one message 113
- putting messages to a distribution list
 - the MQPMR structure 119

Q

- QLOP abend, CICS adapter 218
- QLOP abend on MQSeries for OS/390 429
- QMQM library 542
- QSG (queue-sharing group) 10
 - what it is 6, 40
- queue
 - alias 42, 44
 - application 192
 - attributes 43
 - authority check on MQOPEN 97
 - base 44
 - channel 43
 - closing 97, 105
 - cluster 42
 - creating 41
 - dead-letter 48, 57
 - dead-letter on MQSeries for OS/390 460
 - definition 4
 - dynamic
 - permanent 46
 - temporary 45
 - dynamic, creation of 104
 - event 43
 - exclusive access 102
 - handle 97
 - initiation 47, 193
 - introduction to 41
 - local definition 41
 - model 45, 104
 - name resolution 50
 - name resolution when remote 105
 - object handle 97
 - opening 97
 - order of messages 34
 - remote
 - definition 41
 - putting messages 112
 - using 44
 - using local definition 99
 - using MQOPEN 105
 - reply-to 36
 - resolving name 99
 - rules for naming 49
 - selecting messages 34
 - shared 42
 - shared access 102
 - system admin command 48
 - system command 43
 - system command input 48
 - system default 43, 48
 - transmission 42, 47
 - triggered 204
 - undelivered message 57

- queue (*continued*)
 - undelivered-message 48
- queue attributes for MQSeries for OS/390
 - sample application 443
- queue manager
 - alias definition 99
 - attributes 39
 - authority checking 96
 - connecting using MQCONN 90
 - connecting using MQCONNX 92
 - definition 5
 - disconnecting 95
 - location of default 90
 - number per system 5
 - reply-to 36
 - restart and trigger messages 209
 - scope of MQCONN 91
 - workload management 40
- queue-sharing group (QSG) 10
 - what it is 6, 40
- queue-sharing groups
 - application programming 226
- queuing
 - definition 3
 - features 6
- quiescing connection
 - MQGET 123
- quiescing queue manager
 - how applications should react 54
 - MQCONN 91
 - MQOPEN 104
 - MQPUT 109

R

- reason codes 75
- RECEIVE MAP vectors
 - interpreting with the CICS 3270 bridge 245
- recoverable resource manager services (RRS)
 - batch adapter 215
 - what it is 181
- recovery 9, 215
- RecsPresent field 109
- reenterable assembler-language programs 82
- reference messages 139
- remote queue
 - definition 41
 - using 44
 - using local definition of 99
 - using MQOPEN 105
- reply message 25
- reply messages, IMS bridge 273
- reply-to queue 36
- reply-to queue manager 36
- ReplyToQ field 36
- ReplyToQMgr field 36
- report
 - confirmation of arrival (COA) 25
 - confirmation of delivery (COD) 25
 - exception 25
 - expiry 25
 - negative action notification (NAN) 26
 - positive action notification (PAN) 26

- Report field 25
- report message
 - creating 55
 - options 26
 - type of 25
- reports
 - application-generated 28
 - MQSeries-generated 27
 - retrieval of 28
 - segmented messages 27
- request message 25
- resolution of queue names 50, 99
- ResolvedQMGrName field 109
- ResolvedQName field
 - MQGMO structure 124
 - MQPMO structure 109
- resource manager, XA compliant name 185
- ResponseRecOffset field 110
- ResponseRecPtr field 110
- restrictions in OS/390 batch 84
- retry sending message 56
- return codes 75
- ReturnedLength field
 - MQGMO structure 125
- RPG language
 - compilers supported 471
 - on AS/400 293
 - support for 82
- RPG sample programs 359
- RRS (recoverable resource manager services)
 - batch adapter 215
 - what it is 181
- running a program automatically
 - an example 390
 - how triggering works 191

S

- sample applications
 - API-crossing exit for OS/390 221
- sample applications for MQSeries for OS/390
 - browse 437
 - credit check 451
 - features of MQI demonstrated 421
 - get 435
 - logging on to CICS 428
 - mail manager 444
 - message handler 464
 - preparing in batch 425
 - preparing in CICS Transaction Server for OS/390 428
 - preparing in IMS 431
 - preparing in TSO 427
 - print message 439
 - put 432
 - queue attributes 443
- sample programs
 - building C versions 369
 - building COBOL versions 370
 - building TAL versions 371
 - C++ 359
 - DCE exits 359
 - preparing and running
 - AS/400 367

- sample programs (*continued*)
 - preparing and running (*continued*)
 - Digital OpenVMS systems 367
 - OS/2 and Windows NT 368
 - Tandem NSK 369
 - UNIX systems 367
 - Windows 371
 - RPG 359
- sample programs for MQSeries for AS/400
 - AMQ0ECHA 397
 - AMQ0GET4 378
 - AMQ0REQ4 388
 - AMQ0SETA 395
 - AMQSAMP4 372
 - AMQSECHA 397
 - AMQSERV4 398, 399
 - AMQSGBR4 376
 - AMQSGET4 378
 - AMQSPUT4 373
 - AMQSREQ4 388
 - AMQSSETA 395
 - AMQSTRG4 398
 - AMQSVFC4 398
 - put 373
 - trigger monitor 398
 - trigger server 399
 - using remote queues 400
 - using triggering 392
- sample programs for MQSeries for OS/2 Warp, Windows NT, and UNIX systems
 - amq0gbr0 376
 - amq0get0 378
 - amq0put0 373
 - amq0req0 388
 - amqiech2 397
 - amqiechx 397
 - amqiinq2 394
 - amqiinqx 394
 - amqiset2 395
 - amqisetx 395
 - amqmset2 395
 - amqmsetx 395
 - amqrgrm 380
 - amqrgrma 380
 - amqsbcg 378
 - amqsbcg0 378
 - amqsbcgx 378
 - amqscic0 407
 - amqsdllq 419
 - amqsech 397
 - amqsecha 397
 - amqsechc 397
 - amqsgbr 376
 - amqsgbr0 376
 - amqsgbrx 376
 - amqsget0 378
 - amqsgetc 378
 - amqsgetw 378
 - amqsinq 394
 - amqsinqa 394
 - amqsinqc 394
 - amqsprpm 380

- sample programs for MQSeries for OS/2 Warp, Windows NT, and UNIX systems (*continued*)
 - amqsprma 380
 - amqsptl0 375
 - amqsput0 373
 - amqsputc 373
 - amqsputw 373
 - amqsreq 388
 - amqsreq0 388
 - amqsreqc 388
 - amqsset 395
 - amqsseta 395
 - amqssetc 395
 - amqstrg 398
 - amqstrg0 398
 - amqstrgc 398
 - amqstxgx 417
 - amqstxpx 417
 - amqstxsx.c 407
 - amqsvfc0 398
 - amqsvfcx 398
 - amqsxab0.sqb 400
 - amqsxab0.sqc 400
 - amqsxaf0.sqb 400
 - amqsxaf0.sqc 400
 - amqsxag0.c 400
 - amqsxag0.cbl 400
 - amqsxas0.sqb 400
 - amqsxas0.sqc 400
 - amqsxrm 380
 - amqsxrma 380
 - browse 376
 - browser 378
 - CICS transaction 407
 - data conversion 398
 - dead-letter queue handler 419
 - distribution list 375
 - echo 397
 - get 378
 - inquire 394
 - put 373
 - reference messages 380
 - request 388
 - set sample 395
 - trigger monitor 398
 - TUXEDO 407
 - TUXEDO get 417
 - TUXEDO put 417
 - using remote queues 400
 - using triggering 390
 - XA transaction manager 400
- scope, handles 91, 98
- security 8
- Segmentation field
 - MQGMO structure 125
- segmented messages 33
 - reports 27
- segmented messages, IMS bridge 274
- SegmentStatus field
 - MQGMO structure 125
- selection of messages from queues 34
- selector for attributes 173
- SEND MAP vectors
 - interpreting with the CICS 3270
 - bridge 241
- send message, retry on failure 56

- sense codes, IMS 271
- serialized applications 226
- server application (IMS) 266
- server environment
 - TUXEDO 407
- setting attributes 173
- setting attributes on MQSeries for OS/390 443
- Shareability attribute 103
- shared access to a queue 102
- shared queue 10
 - overview 42
 - what it is 6, 40
- shared queues
 - application programming 226
 - dead-letter queue 228
 - initiation queue 228
 - SYSTEM.* queues 228
- Signal handling on UNIX 84
- Signal1 field 124, 143
- Signal2 field
 - MQGMO structure 124
- signaling 16, 142
- single-phase commit 178
- size of messages 111
- skeleton data-conversion exit 157
- skipping backout 144
- softcopy books 566
- SQL on AS/400 294
- stack size
 - for OS/2 MQI client 324
 - MQSeries for Windows NT 344
- starting applications automatically
 - an example 390
 - how triggering works 191
 - introduction 17
- static XA resource management
 - structure 185
- store-and-forward 7
- StrucId field
 - MQGMO structure 123
 - MQPMO structure 108
- structures 68
 - in COBOL copy files 543
- stub program for MQSeries for OS/390
 - batch 312
 - CICS 313
 - CSQBSTUB 312
 - calling dynamically 315
 - CSQCSTUB 313
 - calling dynamically 315
 - CSQQSTUB 314
 - IMS 314
- stub programs 69
- SupportPac 567
- syncpoint
 - calls by platform 67
 - considerations 178
 - external coordination 184
 - external manager interfaces 185
 - IMS applications 265
 - in CICS for AS/400 applications 182
 - in Tandem NSK applications 188
 - in the Credit Check sample application 461
 - internal coordination 183
 - MQBACK 181

- syncpoint (*continued*)
 - MQCMIT 180
 - overview 8
 - single-phase commit 178
 - two-phase commit 178
 - with MQSeries for AIX 182
 - with MQSeries for AS/400 182, 187
 - with MQSeries for HP-UX 182
 - with MQSeries for OS/2 Warp 182
 - with MQSeries for OS/390 214
 - with MQSeries for Windows NT 182
 - with MQSeries on UNIX systems 182
 - X/Open XA interface 185
- sysplex 6, 40
- SYSTEM.* queues, shared 228
- system command queue 43
- system command queues 48
- system default queue 43, 48
- system interruptions 53

T

- TAL
 - compilers supported 471
 - support for 83
- techniques with MQSeries 16
- temporary dynamic queue
 - closing 105
 - properties 45
- terminology used in this book 553
- testing MQSeries applications 20
- threads, maximum no. 91
- time and date of messages 38
- time-independent communication 8
- TMI (trigger monitor interface) 207
- trace entries for CICS adapter 320
- trace points in CICS adapter 218
- Transaction Server
 - with MQSeries for OS/2 Warp 324, 326
 - with MQSeries for Windows NT 344
- transactions
 - with CICS 3270 bridge 234
- transactions in the distributed environment
 - CICS 3270 bridge 249
 - CICS DPL bridge 250
- transactions with start data
 - CICS 3270 bridge 249
- translation of data 148
- transmission queue 42, 47
- TrigData attribute 197
- trigger
 - event 192
 - conditions for 198
 - controlling 202
 - feedback code 211
 - following queue manager restart 209
 - message
 - definition 192
 - MQTM format 209
 - object attribute changes 209
 - persistence and priority 209
 - properties 209
 - without application messages 199
 - monitor
 - what it is 193
 - writing your own 207

- trigger (*continued*)
 - monitor, provided
 - amqltmc0 206
 - AMQSERV4 206
 - AMQSTRG0 206
 - AMQSTRG4 206
 - AMQSTRG4 sample program 398
 - CKTI 206
 - CSQQTRMN 206
 - runmqmtc 206
 - runmqtrm 206
 - process definition 192
 - server
 - AMQSERV4 sample program 399
 - type of 203
 - trigger monitor
 - provided, by platform 206
 - what it is 193
 - trigger monitor interface (TMI) 207
 - TriggerControl attribute 202
 - TriggerData attribute 192
 - TriggerDepth attribute 203
 - triggered applications 335
 - triggering
 - application design 204
 - application queue 191
 - example of type DEPTH 203
 - example of type EVERY 203
 - example of type FIRST 203
 - getting messages 205
 - how it works 194
 - how it works with the samples 390
 - IMS bridge 276
 - introduction 17, 191
 - no DOS support 191
 - points to note 194
 - prerequisites 196
 - process definition attributes 48
 - sample program
 - for MQSeries for AS/400 392
 - sample trigger monitor for MQSeries for OS/2 Warp 398
 - sample trigger monitor for MQSeries for Windows NT 398
 - sample trigger monitor for MQSeries on UNIX systems 398
 - sequence of events 194
 - setting conditions 202
 - what it is 191
 - when it does not work 210
 - with the request sample on MQSeries for OS/2 Warp 390
 - with the request sample on MQSeries for Windows NT 390
 - with the request sample on MQSeries on UNIX systems 390
 - with units of work 204
 - without application messages 199
 - triggering for MQSeries for OS/390
 - sample application 455
 - TriggerInterval attribute 204
 - TriggerMsgPriority attribute 202
 - TriggerType attribute 202
 - trusted applications 92
 - Tuxedo applications, building 415
 - TUXEDO sample makefile for MQSeries for Windows NT 414

- TUXEDO sample programs
 - amqstxgx 407
 - amqstxpx 407
 - amqstxsx 407
 - building server environment 407
- TUXEDO ubbstxcn.cfg example for MQSeries for Windows NT 413
- TUXEDO ubbstxcx.cfg example for MQSeries on UNIX systems 411
- two-phase commit 178
- TZ environment variable
 - MQSeries for OS/2 Warp only 38

U

- ubbstxcn.cfg example for MQSeries for Windows NT 413
- ubbstxcx.cfg example for MQSeries on UNIX systems 411
- undelivered-message queue, using 57
- undelivered messages, IMS bridge 270
- unit of work
 - message persistence 34
 - syncpoint 177
 - triggering 204
 - with the CICS bridge 256
- units of work
 - managing with CICS DPL bridge 234
 - managing with the CICS 3270 bridge 239
- UNIX signal handling 84
- use of message types 24
- user-defined formats 31
- user exits 221
- UserIdentifier field 37

V

- valid syntax
 - creating conversion-exit code 159
 - input data set 159
- vectors
 - using with CICS 3270 bridge 235
- Version field 109
 - MQGMO structure 123
 - MQPMO structure 109
- Visual Basic
 - on Windows 341
 - on Windows NT 347
- Visual Basic language
 - module files 542, 543
- Visual Basic module files
 - CMQB.BAS 543
 - CMQB3.BAS 542
 - CMQB4.BAS 542, 543
 - CMQBB.BAS 543
 - CMQCFB.BAS 543
 - CMQXB.BAS 543
 - module files 542, 543

W

- waiting for messages 16, 141
- WaitInterval field 124, 141
- Windows 3.1 clients
 - compilers supported 471
- Windows 95 and Windows 98 clients
 - compilers supported 471

- Windows Help 567
- WLM (workload manager) 225
- Workflow 225
- workload management
 - queue manager 40
- workload manager (WLM) 225
- writing applications 275
- writing exit programs
 - data conversion
 - Digital OpenVMS 166
 - MQSeries for AS/400 161
 - MQSeries for OS/2 Warp 162
 - MQSeries for OS/390 164
 - Tandem NSK systems 165
 - UNIX systems 166
 - Windows NT 171

X

- X/Open XA interface support 185
- XA resource manager
 - name 185
- XA transaction manager samples 400

Sending your comments to IBM

If you especially like or dislike anything about this book, please use one of the methods listed below to send your comments to IBM.

Feel free to comment on what you regard as specific errors or omissions, and on the accuracy, organization, subject matter, or completeness of this book.

Please limit your comments to the information in this book and the way in which the information is presented.

To make comments about the functions of IBM products or systems, talk to your IBM representative or to your IBM authorized remarketer.

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

You can send your comments to IBM in any of the following ways:

- By mail, to this address:

User Technologies Department (MP095)
IBM United Kingdom Laboratories
Hursley Park
WINCHESTER,
Hampshire
SO21 2JN
United Kingdom

- By fax:
 - From outside the U.K., after your international access code use 44-1962-870229
 - From within the U.K., use 01962-870229
- Electronically, use the appropriate network ID:
 - IBM Mail Exchange: GBIBM2Q9 at IBMMAIL
 - IBMLink™: HURSLEY(IDRCF)
 - Internet: idrcf@hursley.ibm.com

Whichever method you use, ensure that you include:

- The publication title and order number
- The topic to which your comment applies
- Your name and address/telephone number/fax number/network ID.



Printed in the United States of America
on recycled paper containing 10%
recovered post-consumer fiber.

SC33-0807-12

