

Course Guide

IBM MQ V9 Application Development (Windows Labs)

Course code WM513 / ZM513 ERC 1.0



June 2017 edition

Notices

This information was developed for products and services offered in the US.

IBM may not offer the products, services, or features discussed in this document 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 document. The furnishing of this document does not grant you any license to these patents. You can send license inquiries, in writing, to:

*IBM Director of Licensing
IBM Corporation
North Castle Drive, MD-NC119
Armonk, NY 10504-1785
United States of America*

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 jurisdictions 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 publication. IBM may make improvements and/or changes in the product(s) and/or the program(s) described in this publication at any time without notice.

Any references in this information to non-IBM websites are provided for convenience only and do not in any manner serve as an endorsement of those websites. The materials at those websites are not part of the materials for this IBM product and use of those websites is at your own risk.

IBM may use or distribute any of the information you provide in any way it believes appropriate without incurring any obligation to you.

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.

This information contains examples of data and reports used in daily business operations. To illustrate them as completely as possible, the examples include the names of individuals, companies, brands, and products. All of these names are fictitious and any similarity to actual people or business enterprises is entirely coincidental.

Trademarks

IBM, the IBM logo, and ibm.com are trademarks or registered trademarks of International Business Machines Corp., registered in many jurisdictions worldwide. Other product and service names might be trademarks of IBM or other companies. A current list of IBM trademarks is available on the web at "Copyright and trademark information" at www.ibm.com/legal/copytrade.shtml.

© Copyright International Business Machines Corporation 2017.

This document may not be reproduced in whole or in part without the prior written permission of IBM.

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

Contents

Trademarks	xi
Course description	xii
Agenda	xiv
Unit 1. IBM MQ overview	1-1
How to check online for course material updates	1-2
Unit objectives (1 of 2)	1-3
Unit objectives (2 of 2)	1-4
Business drivers for enterprises today	1-5
Traditional connectivity and message-oriented middleware	1-6
IBM MQ components and baseline	1-7
Queue manager	1-8
Queue manager terminology checkpoint	1-9
Messages	1-10
Queues	1-11
Queue terminology checkpoint	1-13
Some local queues are designated for a special purpose(1 of 2)	1-14
Some local queues are designated for a special purpose(2 of 2)	1-15
Basic Message Queue Interface (MQI) calls	1-16
Channels	1-18
Sender-receiver channel pair without remote queue	1-19
Sender-receiver channel pair with remote queue	1-20
IBM MQ distributed object relationships	1-21
Multi-hopping	1-22
Cumulative checkpoint with distributed platforms	1-24
Typical queue manager point-to-point channel definitions	1-25
IBM MQ clusters	1-26
Queue manager cluster definition	1-28
What are shared queues?	1-30
Triggering	1-31
Triggering: Process scenario	1-32
Messaging styles	1-33
IBM MQ administrative capabilities	1-34
QLOCAL partial attributes and defaults	1-35
QREMOTE complete display less date and time created fields	1-37
Message descriptor (MQMD) header (1 of 2)	1-38
Message descriptor (MQMD) header (2 of 2)	1-39
Transactions: Terminology	1-40
IBM MQ design and development impact behind the scenes	1-41
Terminology checkpoint: Security areas	1-43
Security considerations for IBM MQ	1-44
Where to look for errors	1-45
Key topic review	1-46
Course summary (1 of 2)	1-47
Course summary (2 of 2)	1-48
Review questions (1 of 2)	1-49
Review questions (2 of 2)	1-50
Review answers (1 of 2)	1-51

Review answers (2 of 2)	1-52
Exercise: Working with IBM MQ to find your message	1-53
Exercise objectives	1-54
Unit 2. Basic design and development concepts	2-1
Unit objectives (1 of 2)	2-2
Unit objectives (2 of 2)	2-3
Common messaging patterns	2-4
Application design and performance	2-5
MQI programming options	2-7
Message Queue Interface (MQI)	2-8
MQI data definition files	2-9
A look at selected lines of the cmqc.h header file	2-10
Elementary data types: Partial list	2-11
Structure data types that are used on MQI calls: Partial list	2-12
Structure data types that are used with message data: Partial list	2-13
MQMD message descriptor structure (partial)	2-14
amqsput0.c elementary and structure data definitions	2-15
Named constants	2-16
MQI function calls (1 of 3)	2-17
MQI function calls (2 of 3)	2-18
MQI function calls (3 of 3)	2-19
Handles	2-20
Message Queue Interface review	2-21
Recurring considerations in the MQI	2-22
Common function call sequence	2-23
MQCONN	2-24
MQCONNX	2-25
Version-dependent attributes in structures: Example MQCNO	2-26
MQCNO options	2-28
Connection security (MQCSP) parameter structure	2-29
Implementing connection authentication in MQCONNX	2-30
MQCONN, MQCONNX reason code examples for MQCC_FAILED	2-31
MQCONNX-related environment variables	2-32
MQSAMP_USER_ID and connection authentication settings	2-33
A first look at MQOPEN	2-34
A first look at MQPUT	2-35
A first look at MQGET	2-36
MQINQ and MQSET	2-37
MQINQ and MQSET declarations	2-38
MQINQ invocation	2-39
MQSET invocation	2-40
MQCLOSE and options	2-41
MQCLOSE reason code examples for MQCC_FAILED	2-42
MQDISC and reason code examples for MQCC_FAILED	2-43
MQI and object attribute override considerations	2-44
Manipulating character and binary strings with C language	2-45
Examples of preparing C programs	2-46
Unit summary (1 of 2)	2-47
Unit summary (2 of 2)	2-48
Review questions (1 of 2)	2-49
Review questions (2 of 2)	2-50
Review answers (1 of 3)	2-51
Review answers (2 of 3)	2-52
Review answers (3 of 3)	2-53
Exercise: Getting started with IBM MQ development	2-54

Exercise objectives	2-55
Unit 3. MQOPEN, queue name resolution, and MQPUT	3-1
Unit objectives	3-2
The MQOPEN call	3-3
MQOPEN	3-4
MQOD: Information about the object to open (1 of 2)	3-5
MQOD: Information about the object to open (2 of 2)	3-7
MQOPEN options that are used with MQPUT	3-8
MQOPEN reason codes: Partial (1 of 2)	3-9
MQOPEN reason codes: Partial (2 of 2)	3-10
Queue name resolution	3-11
Sender-receiver channel pair with remote queue	3-12
Sender-receiver channel pair without remote queue	3-13
Multi-hopping	3-14
MQPUT-related structures and options	3-15
MQMD (1 of 2)	3-17
MQMD (2 of 2)	3-19
Message types: ReplyToQ and ReplyToQMgr	3-21
Report messages	3-22
Expiry	3-23
MQMD expiry and the CAPEXPRY queue or topic attribute	3-24
Feedback	3-25
Encoding	3-26
CodedCharSetId	3-27
Message formats (1 of 2)	3-28
Message formats (2 of 2)	3-29
Priority	3-30
Persistence	3-31
Reply-to-Q and Reply-To-QMgr processing	3-32
Message context	3-33
Passing context	3-34
Context handling	3-35
Alternative user authority	3-36
Put message options (MQPMO) structure (1 of 2)	3-37
Put message options (MQPMO) structure (2 of 2)	3-39
MQPMO options	3-40
MQPMO action	3-41
The MQPUT1 call	3-42
MQPUT reason codes (partial)	3-43
Creating dynamic queues	3-44
Common use for a dynamic queue	3-45
Unit summary	3-46
Review questions (1 of 2)	3-47
Review questions (2 of 2)	3-48
Review answers (1 of 4)	3-49
Review answers (2 of 4)	3-50
Review answers (3 of 4)	3-51
Review answers (4 of 4)	3-52
Working with MQOPEN and queue name resolution, MQPUT, and MQMD fields	3-53
Exercise objectives	3-54
Unit 4. Getting messages and retrieval considerations	4-1
Unit objectives	4-2
How are messages retrieved from a queue?	4-3
MQOPEN options used with MQGET	4-4

MQGET declarations and parameters	4-5
Get message options (MQGMO) structure (1 of 2)	4-6
Get message options (MQGMO) structure (2 of 2)	4-7
MQGMO options (1 of 2)	4-8
MQGMO options (2 of 2)	4-10
Buffer length (1 of 2)	4-11
Buffer length (2 of 2)	4-12
MQGET reason codes CompCode MQCC_WARNING (partial)	4-13
MQGET reason codes CompCode MQCC_FAILED (partial)	4-14
Sending replies to the reply-to queue	4-15
Message ID and correlation ID	4-16
Request and reply queue consideration	4-17
MsgId, CorrelId, and application parallelism	4-18
MQPUT and MQPUT1: Message ID and correlation ID	4-19
Retrieval by Message ID and Correlation ID	4-21
Retrieving every message	4-23
MsgId, CorrelId, reports, and replies	4-24
Browsing messages	4-26
Browsing the same message	4-28
Message tokens	4-29
Browse and mark	4-30
Cooperative applications	4-31
WAIT	4-32
WAIT with WaitInterval	4-33
Typical server application	4-34
MQGET with SET SIGNAL option (z/OS only)	4-35
Unit summary	4-36
Review questions	4-37
Review answers	4-38
Exercise: Correlating requests to replies	4-39
Exercise objectives	4-40
Correlating requests to replies	4-41
Unit 5. Data conversion	5-1
Unit objectives	5-2
Data conversion: A case for “receiver makes good”	5-3
Two components of data conversion	5-4
Coded character set identifier and terminology	5-5
Encoding	5-6
What gets converted	5-7
IBM MQ data: How MQMD fields are converted (1 of 2)	5-8
IBM MQ data: How MQMD fields are converted (2 of 2)	5-9
Message data conversion	5-10
Creating a data conversion exit	5-11
Considerations for applications that do original MQPUT	5-12
Exception case when you convert on the sender side	5-13
Unit summary	5-14
Review questions (1 of 2)	5-15
Review questions (2 of 2)	5-16
Review answers (1 of 2)	5-17
Review answers (2 of 2)	5-18
Unit 6. Bind and Message groups	6-1
Unit objectives	6-2
IBM MQ objectives for highly available architectures	6-3
Application design revisited	6-4

Bind attribute	6-5
Physical and logical messages, segments, and groups	6-6
Message groups	6-7
Grouping logical messages	6-8
Retrieving logical messages (1 of 2)	6-10
Retrieving logical messages (2 of 2)	6-12
Spanning units of work	6-13
Message segmentation	6-15
Segmentation by the queue manager	6-16
Reassembly by the queue manager or application	6-17
Segmentation by the application	6-18
Other considerations for application reassembly	6-19
Segmentation and message group: Sample MQPUT	6-20
Segmentation and message group: Sample MQGET	6-21
Segmented messages and reports	6-22
Unit summary	6-23
Review questions	6-24
Review answers (1 of 2)	6-25
Review answers (2 of 2)	6-26
Unit 7. Committing and backing out units of work	7-1
Unit objectives	7-2
Transactions: Terminology	7-3
Local units of work	7-5
Global units of work: Syncpoint coordination	7-6
Syncpoint	7-7
MQPUT within syncpoint control	7-8
MQGET within syncpoint control	7-9
MQBEGIN	7-10
MQBEGIN error codes	7-11
MQCMT	7-12
MQCMT error codes	7-13
MQBACK	7-14
Poisoned messages	7-15
Mark skip backout (z/OS only)	7-16
Remote updates	7-17
Coordination choices	7-18
Triggering	7-19
Trigger types	7-20
Trigger type depth	7-21
MQTMC2 (trigger message)	7-22
Triggering and syncpoint	7-23
Unit summary	7-24
Review questions	7-25
Review answers	7-26
Commit and back out review	7-27
Exercise objectives	7-28
Unit 8. Asynchronous messaging	8-1
Unit objectives	8-2
Using MQGET for traditional message retrieval	8-3
Asynchronous message consumer	8-4
Lifecycle of a message consumer	8-6
Message consumer types	8-7
MQCB: Manage callback	8-8
MQCB function	8-9

MQCB operations	8-10
MQCBD: Callback data descriptor	8-11
MQCTL control callbacks	8-13
MQCTL operations	8-15
MQCB_FUNCTION: Callback function	8-16
MQCBC: Callback context	8-18
MQCBC call types	8-20
Unit summary	8-21
Review questions	8-22
Review answers (1 of 2)	8-23
Review answers (2 of 2)	8-24
Asynchronous messaging review	8-25
Exercise objectives	8-26
Unit 9. IBM MQ clients	9-1
Unit objectives	9-2
IBM MQ clients	9-3
Programming languages supported for clients	9-4
Examples of preparing C client programs	9-5
Considerations when building client applications	9-6
IBM MQ base and extended transactional client	9-7
IBM MQ client connectivity options (1 of 2)	9-8
IBM MQ client connectivity options (2 of 2)	9-9
URL support for CCDT files	9-10
Precedence order for finding the CCDT	9-11
MQI channel objects	9-12
Testing client to server connectivity with MQCONN	9-14
Revisiting MQCONN	9-15
Version-dependent attributes MQCNO (1 of 2)	9-16
Version-dependent attributes MQCNO (2 of 2)	9-17
MQCD structure (partial)	9-18
MQCD_CLIENT_CONN_DEFAULT (partial)	9-19
Checklist to use MQCONN for the client connection	9-20
Connect with CCDT and put a message	9-21
Use MQSERVER environment variable	9-22
IBM MQ client connection capabilities summary	9-23
Redistributable clients	9-24
IBM MQ client security considerations	9-26
Unit summary	9-27
Review questions (1 of 2)	9-28
Review questions (2 of 2)	9-29
Review answers (1 of 2)	9-30
Review answers (2 of 2)	9-31
Working with an IBM MQ client	9-32
Exercise objectives	9-33
Unit 10. Introduction to publish/subscribe	10-1
Unit objectives	10-2
Point-to-point and publish/subscribe	10-3
Distributed publish/subscribe patterns	10-4
Publish/subscribe basic components	10-5
Current publish/subscribe functions: A little history	10-6
Publish/subscribe terminology baseline	10-7
Publish/subscribe functions	10-8
Topic tree, topic strings, topic nodes, and topic objects	10-9
Topic objects	10-10

Topic object attributes: DISPLAY TOPIC view	10-11
Topic object attributes: DISPLAY TPSTATUS view	10-12
Topic alias	10-13
Wildcard schemes	10-14
Topic status with IBM MQ Explorer	10-15
Subscriptions: The three aspects (1 of 2)	10-16
Subscriptions: The three aspects (2 of 2)	10-17
Subscription commands: DIS SUB(VEGET*) ALL	10-18
Subscription commands: DISPLAY PUBSUB ALL	10-19
Subscription commands: DISPLAY SUB(*) TOPICSTR DEST	10-20
Subscription-related command: DISPLAY QLOCAL	10-21
Publications	10-22
Retained publications	10-23
Publish/subscribe testing in IBM MQ Explorer	10-24
Distributed publish/subscribe	10-25
Publish/subscribe lifecycle descriptions	10-26
Publish/subscribe lifecycles: Managed non-durable subscriber	10-27
MQSUB for subscriber	10-28
Subscription description structure (MQSD)	10-29
MQSD options	10-30
Working with topic strings	10-31
MQOPEN for publisher	10-32
MQPUT for publisher	10-33
Unit summary	10-34
Review questions (1 of 2)	10-35
Review questions (2 of 2)	10-36
Review answers (1 of 3)	10-37
Review answers (2 of 3)	10-38
Review answers (3 of 3)	10-39
Working with publish/subscribe basics	10-40
Exercise objectives	10-41
Unit 11. Advanced Message Queuing Protocol (AMQP) and IBM MQ Light	11-1
Unit objectives	11-2
Advanced Message Queuing Protocol (AMQP)	11-3
Partial list of AMQP products and adopters	11-5
IBM MQ Light	11-6
Some use cases for IBM MQ Light	11-7
IBM MQ Light concepts	11-8
IBM MQ Light components	11-10
IBM MQ Light API for node.js: mqlight.createClient()	11-12
IBM MQ Light API for node.js: mqlight.Client.send()	11-14
IBM MQ Light API send sample as implemented in node.js	11-16
IBM MQ Light API node.js: mqlight.Client.subscribe()	11-17
IBM MQ Light API receive sample as implemented in node.js	11-19
IBM MQ and IBM MQ Light scenario terminology baseline	11-21
Special terminology clarification	11-23
IBM MQ as a messaging provider for AMQP applications	11-24
Exchanging messages with IBM MQ V7 and later	11-25
Send messages from IBM MQ Light to IBM MQ	11-26
Send messages from an IBM MQ to IBM MQ Light	11-28
From IBM MQ Light to IBM MQ queue-based application	11-30
From an IBM MQ queue-based application to IBM MQ Light	11-31
AMQP message format and terminology	11-32
Mapping IBM MQ to AMQP message (1 of 2)	11-33
Mapping IBM MQ to AMQP message (2 of 2)	11-35

Mapping AMQP to IBM MQ message (1 of 6)	11-37
Mapping AMQP to IBM MQ message (2 of 6)	11-38
Mapping AMQP to IBM MQ message (3 of 6)	11-39
Mapping AMQP to IBM MQ message (4 of 6)	11-40
Mapping AMQP to IBM MQ message (5 of 6)	11-41
Mapping AMQP to IBM MQ message (6 of 6)	11-42
Enabling the IBM MQ AMQP environment	11-43
Confirm the AMQP functionality in the queue manager	11-44
Start and check the AMQP service	11-45
DIS SVSTATUS(SYSTEM.AMQP.SERVICE)	11-46
Configure the AMQP channel	11-47
Display the AMQP channel and channel status	11-49
Test with the IBM MQ Light sample applications	11-50
Subscribe and publish across IBM MQ and IBM MQ Light (1 of 2)	11-52
Subscribe and publish across IBM MQ and IBM MQ Light(2 of 2)	11-53
Both IBM MQ and IBM MQ Light applications get messages	11-54
Display topic status for the queue manager	11-55
AMQP channel log files	11-56
Security	11-57
Unit summary	11-58
Review questions (1 of 2)	11-59
Review questions (2 of 2)	11-60
Review answers (1 of 3)	11-61
Review answers (2 of 3)	11-62
Review answers (3 of 3)	11-63
Connecting IBM MQ Light applications to IBM MQ applications	11-64
Exercise objectives (1 of 2)	11-65
Exercise objectives (2 of 2)	11-66
Unit 12. Course summary	12-1
Unit objectives	12-2
Course objectives	12-3
Course objectives	12-4
Earn an IBM Badge	12-5
To learn more on the subject	12-6
Enhance your learning with IBM resources	12-7
Unit summary	12-8
Course completion	12-9
Appendix A. List of abbreviations	A-1

Trademarks

The reader should recognize that the following terms, which appear in the content of this training document, are official trademarks of IBM or other companies:

IBM, the IBM logo, and ibm.com are trademarks or registered trademarks of International Business Machines Corp., registered in many jurisdictions worldwide.

The following are trademarks of International Business Machines Corporation, registered in many jurisdictions worldwide:

AIX®	CICS®	developerWorks®
IMS™	Initiate®	MVS™
Notes®	Passport Advantage®	Redbooks®
Tivoli®	WebSphere®	z/OS®

Intel is a trademark or registered trademark of Intel Corporation or its subsidiaries in the United States and other countries.

Linux is a registered trademark of Linus Torvalds in the United States, other countries, or both.

Microsoft and Windows are trademarks of Microsoft Corporation in the United States, other countries, or both.

Java™ and all Java-based trademarks and logos are trademarks or registered trademarks of Oracle and/or its affiliates.

UNIX is a registered trademark of The Open Group in the United States and other countries.

VMware is a registered trademark or trademark of VMware, Inc. or its subsidiaries in the United States and/or other jurisdictions.

Other product and service names might be trademarks of IBM or other companies.

Course description

IBM MQ V9 Application Development (Windows Labs)

Duration: 3 days

Purpose

This course helps you develop the skills that are necessary to implement various application requirements on IBM MQ versions up to and including IBM MQ V9.0.2. It focuses on procedural application development for IBM MQ.

The course begins by describing IBM MQ and the effect of design and development choices in the IBM MQ environment. It then covers IBM MQ application programming topics such as methods of putting and getting messages, identifying code that creates queue manager affinities, working with transactions, and uses of the publish/subscribe messaging style.

Finally, the course describes the IBM MQ Light interface, introduces Advanced Message Queuing Protocol (AMQP), and explains how to set up an AMQP channel and how to interface with IBM MQ Light.

Hands-on exercises throughout the course reinforce the lecture material and give you experience with IBM MQ clients.

Audience

This course is designed for application developers and architects who are responsible for the development and design of IBM MQ applications.

Prerequisites

- Successful completion of *Technical Introduction to IBM MQ* (WM103G), or comparable experience with IBM MQ
- Experience in business application design
- Experience in C language development

Objectives

- Describe key IBM MQ components and processes
- Explain the effect of design and development choices in the IBM MQ environment
- Describe common queue attributes and how to control these attributes in an application
- Differentiate between point-to-point and publish/subscribe messaging styles
- Describe the calls, structures, and elementary data types that compose the message queue interface

- Describe how IBM MQ determines the queue where messages are placed
- Explain how to code a program to get messages by either browsing or removing the message from the queue
- Describe how to handle data conversion across different platforms
- Explain how to put messages that have sequencing or queue manager affinities
- Explain how to commit or back out messages in a unit of work
- Describe how to code programs that run in an IBM MQ Client
- Explain the use of asynchronous messaging calls
- Describe the basics of writing publish/subscribe applications
- Describe the Advanced Message Queuing Protocol (AMQP)
- Differentiate among the various IBM MQ Light AMQP implementations
- Explain how to use IBM MQ applications to interface with IBM MQ Light

Agenda



Note

The following unit and exercise durations are estimates, and might not reflect every class experience.

Day 1

- (00:15) Course introduction
- (01:00) Unit 1. IBM MQ overview
- (01:00) Exercise 1. Working with IBM MQ to find your message
- (01:00) Unit 2. Basic design and development concepts
- (01:00) Exercise 2. Getting started with IBM MQ development
- (01:00) Unit 3. MQOPEN, queue name resolution, and MQPUT
- (01:00) Exercise 3. Working with MQOPEN and queue name resolution, MQPUT, and MQMD fields

Day 2

- (01:00) Unit 4. Getting messages and retrieval considerations
- (01:00) Exercise 4. Correlating requests to replies
- (00:30) Unit 5. Data conversion
- (00:30) Unit 6. Bind and Message groups
- (00:30) Unit 7. Committing and backing out units of work
- (00:15) Exercise 5. Commit and back out review
- (00:30) Unit 8. Asynchronous messaging
- (00:30) Exercise 6. Asynchronous messaging review

Day 3

- (00:30) Unit 9. IBM MQ clients
- (01:00) Exercise 7. Working with an IBM MQ client
- (01:00) Unit 10. Introduction to publish/subscribe
- (00:45) Exercise 8. Working with publish/subscribe basics
- (01:00) Unit 11. Advanced Message Queuing Protocol (AMQP) and IBM MQ Light
- (01:00) Exercise 9. Connecting IBM MQ Light applications to IBM MQ applications
- (00:15) Unit 12. Course summary

Unit 1. IBM MQ overview

Estimated time

01:00

Overview

This unit provides an understanding of IBM MQ as a base to the development lectures, with an emphasis on writing well behaved, scalable applications. An IBM MQ developer can code programs that might result in performance problems, or can introduce affinities that impose limits on the ability to scale the infrastructure. This unit lays the foundation for the topics in subsequent units, and introduces potential issues to avoid.

How you will check your progress

Accountability:

- Review questions
- Lab Exercises

References

IBM Knowledge Center for IBM MQ V9 documentation



How to check online for course material updates



Note: If your classroom does not have internet access, ask your instructor for more information.

Instructions

1. Enter this URL in your browser:
ibm.biz/CloudEduCourses
2. Find the product category for your course, and click the link to view all products and courses.
3. Find your course in the course list and then click the link.
4. The wiki page displays information for the course. If a course corrections document exists, this page is where it is found.
5. If you want to download an attachment, such as a course corrections document, click the **Attachments** tab at the bottom of the page.


The screenshot shows a navigation bar with four tabs: "Comments (0)", "Versions (1)", "Attachments (1)" (which is highlighted in blue), and "About".
6. To save the file to your computer, click the document link and follow the prompts.

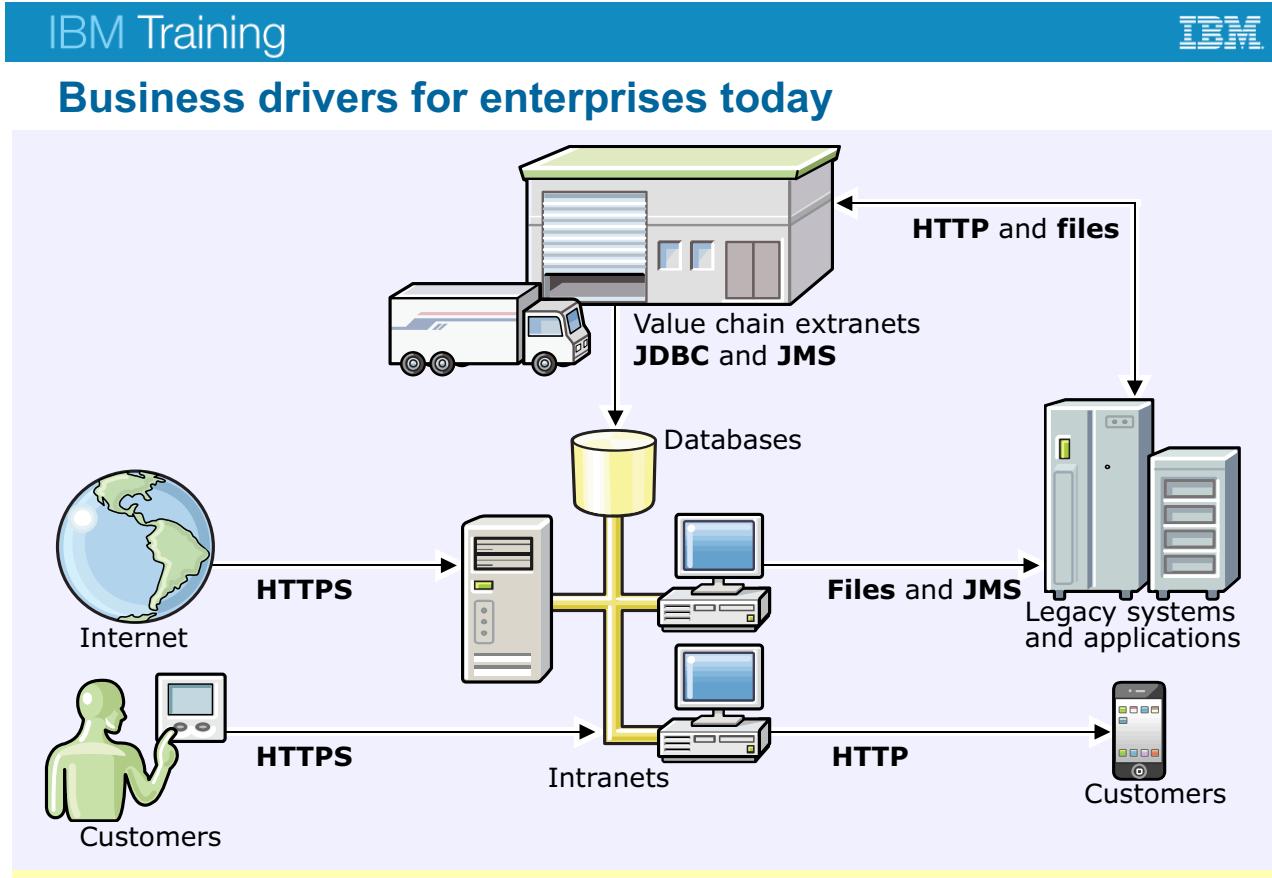
Figure 1-1. How to check online for course material updates

Unit objectives (1 of 2)

- Explain the advantages of message-oriented middleware
- List the basic IBM MQ components
- Describe the correct terminology to use when working with IBM MQ resources
- Distinguish the various types of queues and how they are used
- List basic IBM MQ application programming interface functions
- Explain queue name resolution
- Explain IBM MQ channels
- Describe how application design affects IBM MQ clusters
- Describe queue sharing groups
- Describe the use of triggering in IBM MQ
- Explain the differences between IBM MQ clients and IBM MQ servers
- Distinguish between point-to-point and publish/subscribe messaging styles

Unit objectives (2 of 2)

- Describe attributes that are present in a queue definition
- Explain the message descriptor fields, how they relate to queue attributes, and how they influence application behavior
- Distinguish between local and global units of work
- Describe how design and development decisions impact various IBM MQ resources
- Describe IBM MQ security and how it might impact application development
- Explain where to look for information on IBM MQ errors



Enterprises need **infrastructure flexibility** to remain competitive.

IBM MQ overview

© Copyright IBM Corporation 2017

Figure 1-4. Business drivers for enterprises today

Business requirements are changing in response to “the Internet of Things”. What is meant by “the Internet of Things”? It means the connections that exist across users and applications, which today are increased exponentially.

The amount of data that is exchanged is also increased, and organizations must be ready to accept high amounts of information from business partners.

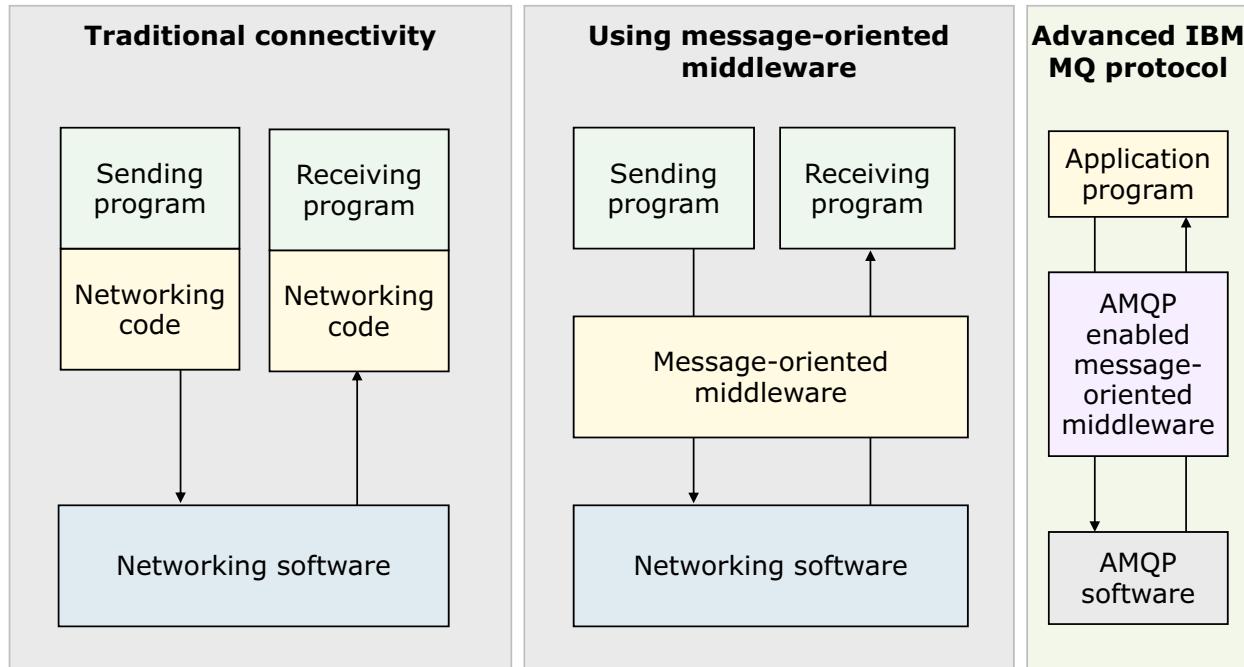
Old “batch” processes, if not disappearing, are diminishing resulting in fewer change windows for the enterprise.

Applications must be resilient and capable of handling failure. Users expect the same quick response now, as when they used to have a direct connection to the data.

Users also need to get information now rather than wait for a batch process. Now also means accessed by a mobile device on demand.

Extending existing processes would get prohibitively expensive; applications need to become flexible so they can be used as needed. What does flexible mean? Data must be available on demand, in different ways such as in mobile applications. Customers and Business Partners have higher expectations, and the inability to meet the expectation equates to loss of business.

Traditional connectivity and message-oriented middleware



IBM MQ overview

© Copyright IBM Corporation 2017

Figure 1-5. Traditional connectivity and message-oriented middleware

When message-oriented middleware is used, all the work of connecting, polling, handling failure, and implementing change is removed from the application and handed off to the messaging middleware, providing flexibility to react to business conditions.

The rightmost box in the diagram illustrates Advanced Message Queuing Protocol, or AMQP. AMQP is a newer messaging protocol that is gaining popularity with part of the messaging community. One of IBM's AMQP implementations is IBM MQ Light. Selected IBM MQ platforms can interface with AMQP applications such as IBM MQ Light. The last unit in this course covers AMQP, and how selected IBM MQ platforms interface with AMQP applications such as IBM MQ Light.

IBM MQ components and baseline

- The first part of this unit focuses on the basic IBM MQ components
 - Messages
 - Queues
 - Queue manager
 - Channels
 - Message queuing (application) interface
- IBM MQ can be installed with IBM MQ server software, with IBM MQ client software, or with both
 - IBM MQ server and IBM MQ client are two distinct installed products
 - Unless a topic is explicitly identified as applying to an IBM MQ client, the information that is presented pertains to installations of IBM MQ server software

Figure 1-6. IBM MQ components and baseline

As you start developing IBM MQ applications, you need to test your applications. Depending on your work environment, you might need to understand some autonomy or set up part of the infrastructure that you need to be able to test your code. You also must understand the technology behind the code you are writing. This unit sets the base for your development work by introducing key IBM MQ concepts that you must understand to be able to write and test your code.

The queue manager, or messaging server, owns IBM MQ resources and controls processes such as defining objects, trigger starting a channel or process, creating event messages, expiring messages, and message distribution by using clustering.

Your business data travels as messages that the queue managers exchange. Queues are where messages are kept. Channels move messages from one queue manager to another queue manager, sometimes jumping across other queue managers.

The Message Queue Interface, or MQI, is the application interface that is used to put and get messages in IBM MQ. The MQI is not to be confused with IBM MQ client channels, which are also referred to as MQI channels. In this unit, you refer to IBM MQ servers. IBM MQ clients are explained in a later unit.



Queue manager

- System program that owns the resources it services
- Provides services that are needed for messaging
- A server can host more than one queue manager
- Queue managers that share a server need different TCP/IP port numbers



IBM MQ overview

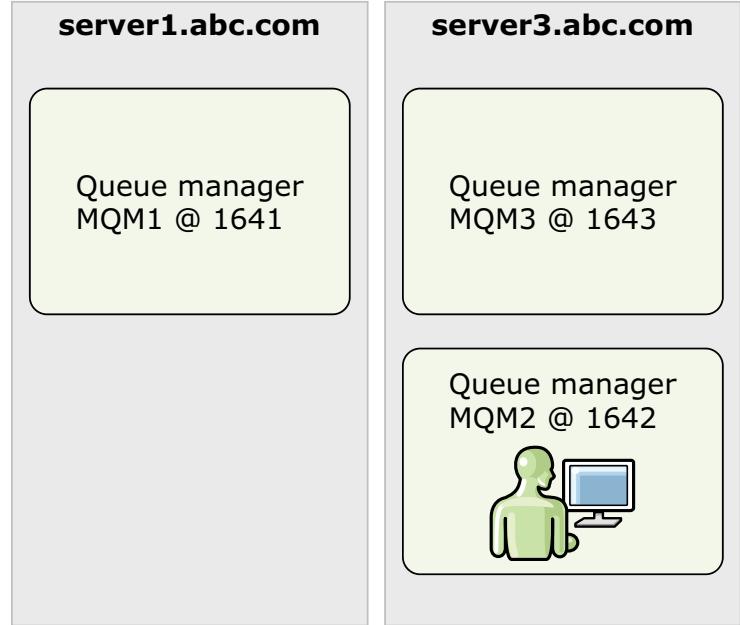
© Copyright IBM Corporation 2017

Figure 1-7. Queue manager

When you install IBM MQ server software, you need to create one or more queue managers. A queue manager owns resources that are defined to it, which you use to exchange messages. When you have more than one queue manager in the same host, the queue managers must have a different port number. The default IBM MQ port is 1414.

Queue manager terminology checkpoint

- **Local queue manager** is the currently referenced queue manager
- All other queue managers are referred to as **remote queue managers**
- Example:
 - If working with MQM2, then MQM2 is the local queue manager and MQM1 and MQM3 are remote queue managers
 - Objects and resources that are defined in MQM2 are said to be “locally owned”



IBM MQ overview

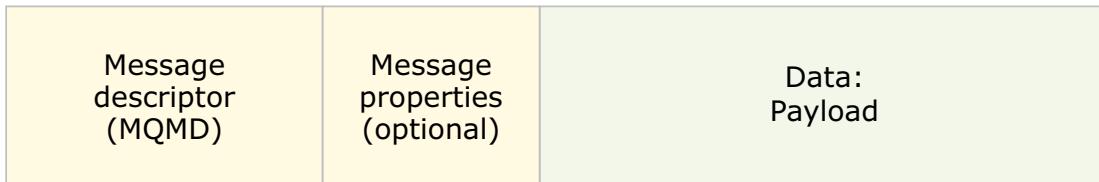
© Copyright IBM Corporation 2017

Figure 1-8. Queue manager terminology checkpoint

When you work with IBM MQ, you refer to the queue manager you are working with at the time as the “local” queue manager, and all other queue managers as “remote” queue managers.

Messages

- Messages contain the business data or payload
- Messages are a distinct string of bytes exchanged between two programs
- All messages contain an IBM MQ message descriptor (MQMD) with control information
- Optionally, applications can add message properties to a message



IBM MQ overview

© Copyright IBM Corporation 2017

Figure 1-9. Messages

Messages contain the message descriptor, or MQMD structure, and the message data or payload. Messages might contain other headers.

The IBM MQ message descriptor, or MQMD, contains several fields. These fields have default values. In a later unit, you learn about predefined structures such as the MQMD. You also learn to use predefined initialization structures.

Message properties allow an application to select messages to process or to retrieve information about a message without accessing MQMD or MQRFH2 headers. A message property is data that is associated with a message, consisting of a textual name and a value of a particular type. Message properties can be used to include business data or state information without having to store it in the application data.

You take a first look at an MQMD structure later in this unit, and in more detail in a later unit.

Queues

- A queue is a defined destination for messages
- Four types of queues can be created: QLOCAL, QREMOTE, QALIAS, QMODEL
- Some local queues are designated for special purposes in a queue manager
- Remote queues (QREMOTE) are pointers to local queues in a remote queue manager
- An alias queue or QALIAS is a pointer to a local queue or a locally owned remote queue
- A model queue or QMODEL is a template to create a dynamic local queue



Only local queues that are defined as a QLOCAL queue type can hold messages

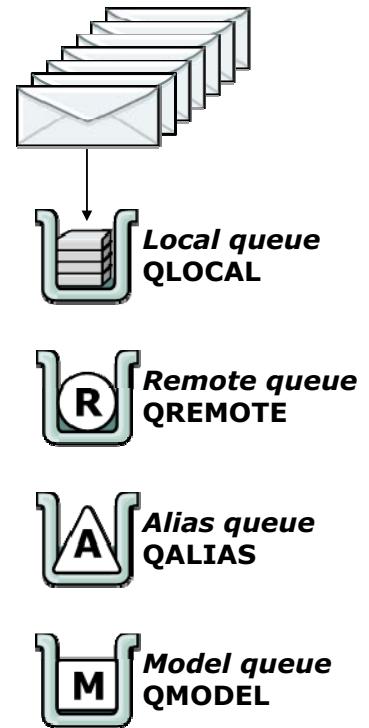


Figure 1-10. Queues

Queues are where applications send messages to and get messages from.

When you start developing IBM MQ code, you might spend some determining “where is my message.” Queues have different types. What is key to remember is:

- That *only local queues of type QLOCAL at definition time hold messages*. Keeping this concept in mind is helpful when looking for a message.
- The *type of queue* counts when the queue manager is doing queue name resolution, that is, when it determines which is the target queue to send a message. You learn more about queue resolution in a later unit.

Queues other than QLOCAL queue types are pointers or templates to other queues:

- Remote queues point to a transmit queue, and a remote queue manager and target queue. QREMOTE is the queue type only. You can also think of these queues as “local QREMOTES”.
- Alias queues, or QALIAS, point to another local queue or a topic. You revisit these queues in a later unit.

Model queues, or QMODEL, can be thought of as a template to create queues. The QLOCAL type queues, which are created when an application uses a model queue, hold messages. The model queue itself does not hold messages. Since these local queues are created in your code, they are

referred to as a **dynamic queue**. You work to create these **dynamic queues** by using a model queue in a later exercise.

You also learn about shared queues. Shared queues are a z/OS-only capability where several queue managers in a queue-sharing group can use the same definition of a queue, with messages stored in the same place. Messages for a shared queue are held in a z/OS coupling facility structure. If a shared queue manager fails, another queue manager in the queue-sharing group can access the messages in the shared queue.

Shared queue capability should *not* be mistaken for IBM MQ clusters, where several queue managers can define the same queue. In a cluster, if a queue manager fails:

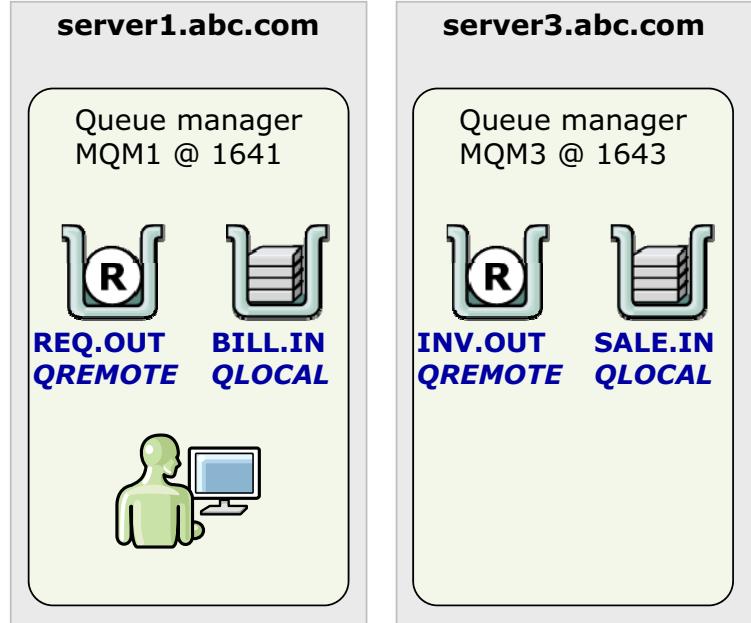
- The messages in the failed queue manager cannot be accessed without a failover solution.
- In a cluster, new requests can go to the same named clustered queue in a different, active queue manager member of the cluster.

Shared queues are a z/OS topic. You learn more about clusters later in this unit.

Queue terminology checkpoint

Relative to current queue manager MQM1

- REQ.OUT is a local remote queue, locally owned remote queue, or local QREMOTE
- BILL.IN is a locally owned local queue, or a local QLOCAL
- INV.OUT is a remote QREMOTE, or remotely owned remote queue
- SALE.IN is a remote local queue, or remotely owned local queue
- Locally owned remote queue REQ.OUT points to remotely owned local queue SALE.IN



MQM3 queues INV.OUT and SALE.IN can also be referred to as "remote queues"

Figure 1-11. Queue terminology checkpoint

With the introduction of queues, it is time to make more distinctions about the terminology used.

Sometimes practitioners use the term “current queue manager” instead of “local queue manager”; they both mean the queue manager currently being worked on.

As noted in the queue manager terminology checkpoint, the terms *local* and *remote* are relative to the queue manager where work is done.

At times, you might need to have conversations about the queues and queue managers. A good way to keep the local and remote concepts clear is to:

- Draw a box for each queue manager
- Draw the respective queues inside each box when referring to queue manager definitions

A simple drawing facilitates and clarifies conversations about the infrastructure. Even experienced IBM MQ administrators use the approach of drawing the queue managers and queues.

Some local queues are designated for a special purpose (1 of 2)

- Transmit or transmission queue 
 - Local queue with its usage attribute set to XMITQ in the queue definition
 - If an application puts a message to a remote queue, it goes to a transmit queue
 - Channels use transmit queues to send messages to the remote queue manager
 - Applications should not write to a transmission queue, only the queue manager

- Initiation queues 
 - Identified as an initiation queue in a definition of another local queue
 - Associated with triggering

Figure 1-12. Some local queues are designated for a special purpose(1 of 2)

The basic queue types were discussed in the last slide. Several other queues are named after their purpose, but these queues are also local queues, or QLOCALs.

Transmission queues and initiation queues are QLOCALs.

Transmission queues can be a deciding factor for queue name resolution, as you see in a later unit.

Some local queues are designated for a special purpose (2 of 2)

- Dead-letter queue 

- Local queue that is identified to the queue manager as its dead-letter queue to hold undeliverable messages
- Usually the SYSTEM.DEAD.LETTER queue is designated as dead-letter queue; however, a different local queue can be defined such as BILLING.DLQ
- The dead-letter queue has an MQDLH header that contains the reason that a message was placed in the dead-letter queue and other pertinent information

- Queues starting with “SYSTEM” 

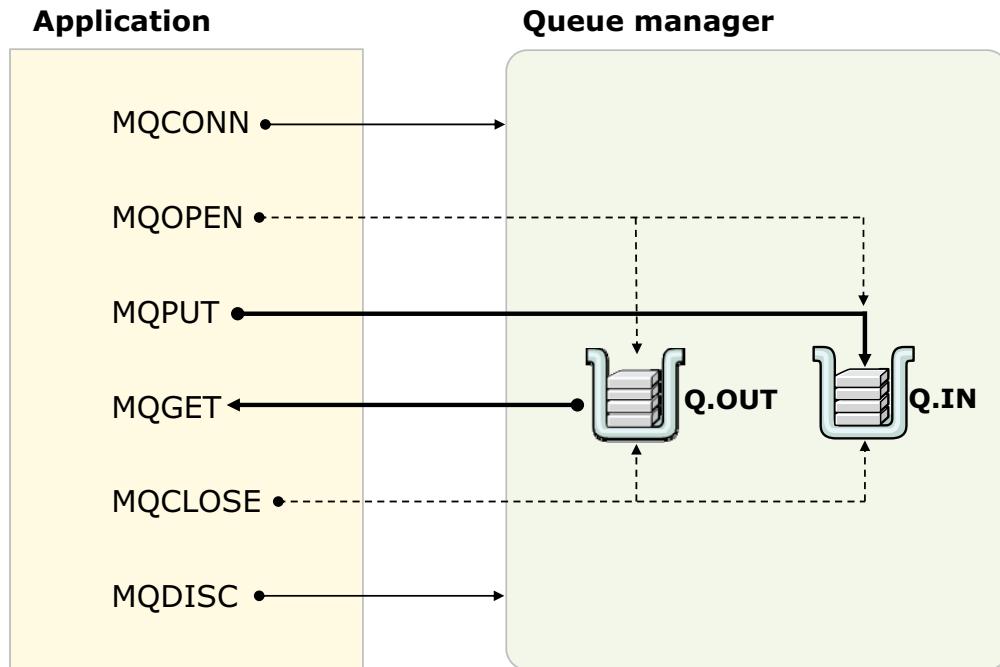
- Most of the SYSTEM.* queues are local queues that are used by the queue manager
- SYSTEM.* queues can be browsed but unless otherwise documented should not be used to put messages

Figure 1-13. Some local queues are designated for a special purpose(2 of 2)

You have the dead-letter queue, which is also a QLOCAL. The name of the dead-letter queue might be any syntactically correct name for a queue, such as MQ01.DEAD.QUEUE. What makes it the dead-letter queue is being identified to the queue manager as the dead-letter queue. Some organizations use the SYSTEM.DEAD.LETTER.QUEUE, which must also be identified to the queue manager. For distributed queue managers, initially a queue manager does not know which queue to use as a dead-letter queue until it is configured in the DEADQ parameter of the queue manager object.

A series of queues are prefixed with SYSTEM.*. The SYSTEM.* queues are also special-purpose QLOCALS.

Basic Message Queue Interface (MQI) calls



IBM MQ overview

© Copyright IBM Corporation 2017

Figure 1-14. Basic Message Queue Interface (MQI) calls

You learned about queues, messages, and channels. The main part of this course focuses on how your data gets in and out of these messages by using the IBM MQ function calls.

Overall, applications connect to a queue manager and open a queue. If the MQCONNECT and MQOPEN calls are successful, then messages can be put to, or retrieved from, a queue.

The MQOPEN provides access to the queue. Each queue needs its MQOPEN call, and its own MQCLOSE call.

If many messages are expected to be placed in the queue, a loop of puts can accomplish this purpose. It would be more efficient than connecting and opening the queue for every put.

Applications can also open a topic and “publish” to this topic. You work with the concept of opening a topic in a later exercise.

Applications can work with messages in many different ways. For example, do you want an obsolete message kept around? Do you want important messages to be preserved and tracked? Do you need to automate consumption of your messages?

**Note**

The figure in the slide is a high-level representation. The MQOPEN broken arrow to each queue denotes that each queue has an MQOPEN call, and each queue has an MQCLOSE call.

The slide shows the sequence in which the calls need to be coded. Each queue requires its own MQOPEN, MQCLOSE, and related structure declarations, as you see in a later unit.

Subsequent units of this course focus on different function calls that are used to work with IBM MQ messages and objects. Attention is also given to the different structures, variables, and options available to manipulate the actions in the function calls.

Channels

- Channels are the combination of MCA's and network
- Usually the sending MCA, the network connection, and the receiving MCA
- Channels come in categories:
 - **Message channels:** Distributed or clustered
 - **Client (or MQI) channels**
 - **AMQP channels**

AMQ8414: Display Channel details.
 CHANNEL (MQM1.MQM3)
 CHLTYPE (SDR)
 CONNAME (server3.abc.com(1643))
 TRPTYPE (TCP)
 XMITQ (MQM3)



"MQI" has two uses in IBM MQ



IBM MQ overview

© Copyright IBM Corporation 2017

Figure 1-15. Channels

Channels are the processes that move messages across queue managers or servers with an IBM MQ client installation. They have different categories of channels:

- Message channels
- Client, or MQI channels
- Advanced Message Queuing Protocol, or AMQP channels (introduced with the IBM MQ V8.0.0.4 product distribution package)

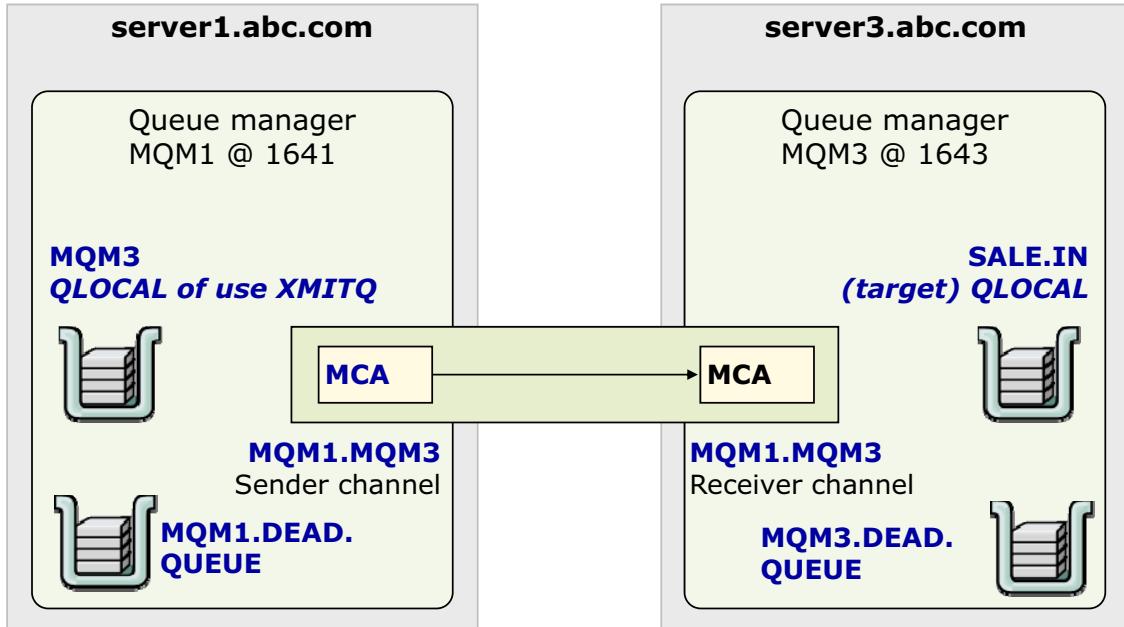
Different types of channel pairs exist within distributed channels. Sender-receiver channels are the most prevalent type of distributed message channel pairs.

Clustered channels are discussed later in this unit. The focus now is on distributed message channels. MQI and AMQP channels are discussed in later units.

As you progress through this course, the MQI acronym might become confusing, as MQI is used to refer to two different IBM MQ components:

- MQI is normally used to refer to the IBM MQ application programming interface; that is, the function calls, structures and options used for application development.
- MQI is also used to refer to a client, versus a distributed channel because the IBM MQ client uses the messaging and queuing interface in its communication with the IBM MQ server.

Sender-receiver channel pair without remote queue



IBM MQ overview

© Copyright IBM Corporation 2017

Figure 1-16. Sender-receiver channel pair without remote queue

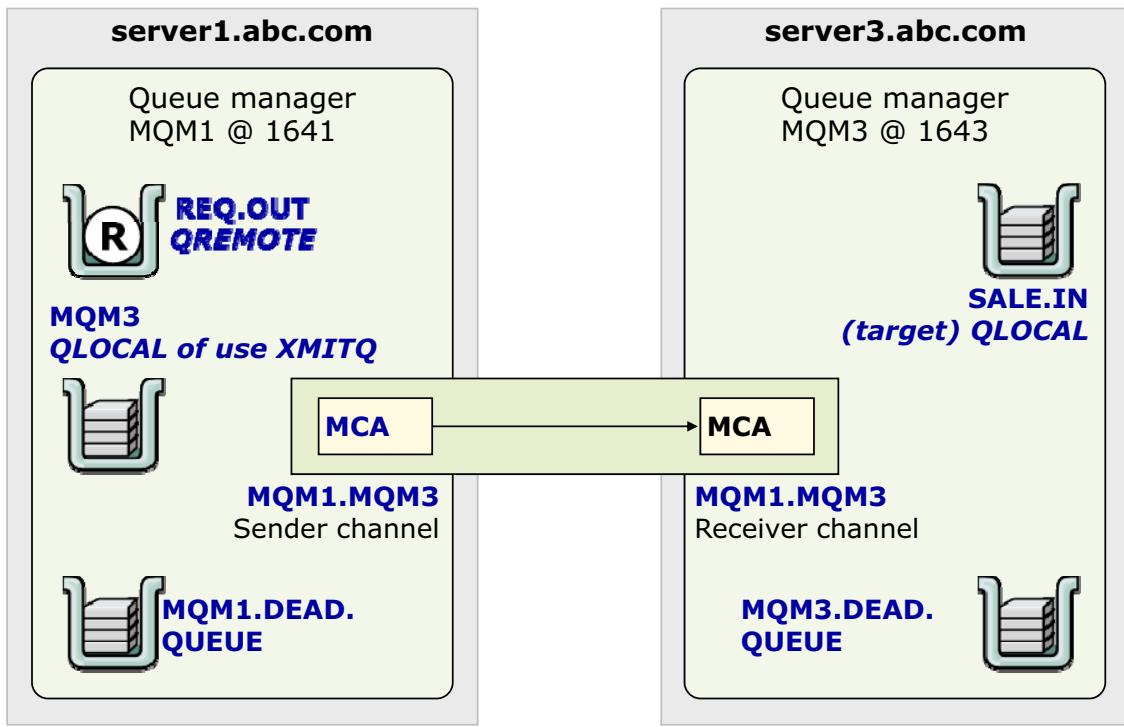
This visual of a sender-receiver channel pair displays some basic concepts. Both the sender channel and the receiver channel are named MQM1.MQM3. If the sender and receiver names are not identical, the channel does not start. The channels might be called anything, such as TRICK.OR.TREAT. If both ends of the channel have the same name, assuming all other connectivity details are correct, they work.

Be careful with the connection name (CONNNAME) attributes. Omitting the port number results in a connection to port 1414 attempted. 1414 is the IBM MQ default port.

This channel takes messages from transmit queue MQM3 to send it to the remote queue manager of the same name. It is a good practice to name the channels to be the same as the “from” and “to” queue managers, in the correct from-to order.

If a second channel pair needs to be defined from and to the same queue managers, the name can always be qualified to differentiate. The from-to naming convention aids with self-documentation of the channel.

Sender-receiver channel pair with remote queue



IBM MQ overview

© Copyright IBM Corporation 2017

Figure 1-17. Sender-receiver channel pair with remote queue

You defined the queues and channels, and sent a message. Now you need to locate the message.

The application for this diagram uses the REQ.OUT QREMOTE in the MQM1 local queue manager to place a message to the SALE.IN remote local queue in the MQM3 remote queue manager.

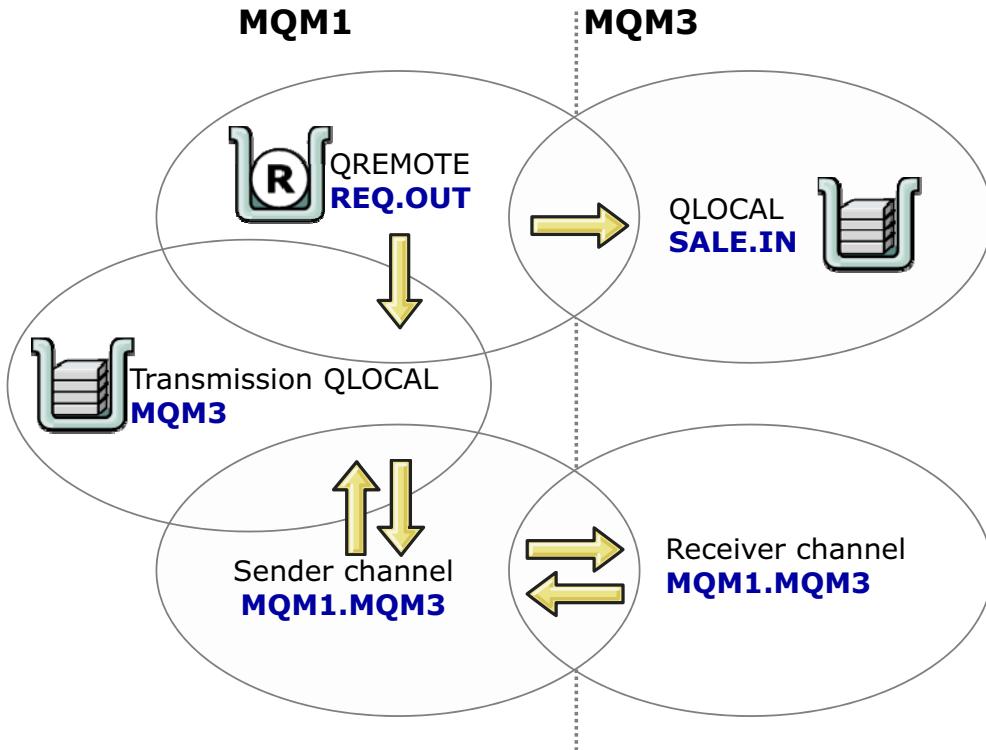
Where can the message be?

QREMOTES, or local remote queues, do not hold messages, so the path of the message on its way to the SALE.IN queue is as follows:

- When an application uses REQ.OUT for the MQPUT, the message goes to the transmission queue associated with the REQ.OUT remote queue, that is, the MQM3 local transmit queue.
- If the channel is triggered or already running, the channel picks up the message and forwards it to the MQM3 queue manager. If the channel is not running, the message is in the MQM3 transmission queue of the MQM1 queue manager.
- If the SALE.IN remote local queue is defined and available, the message is placed in its target queue.

The message might not arrive at the target queue for several reasons. The message might end up in the local or remote dead-letter queue. Look for queue manager log messages, which might indicate what happened.

IBM MQ distributed object relationships



IBM MQ overview

© Copyright IBM Corporation 2017

Figure 1-18. IBM MQ distributed object relationships

The same transmit queue can be used for several remote queues (QREMOTE type queues), and for one message channel. Looking at object relationships, when a message is put in MQM1 local remote queue REQ.OUT, this message is placed in the MQM3 transmit queue.

If channel MQM1.MQM3 is running, the MCA gets the message and sends it to the remote target local queue at MQM3, SALE.IN.

Designating MQM1 as the local queue manager, you can see how the QREMOTE is related to the transmission queue on the local queue manager. The QREMOTE is also related to the target queue in the remote queue manager. The sender channel is related to the transmission queue in the local queue manager, and the receiver channel in the remote queue manager.

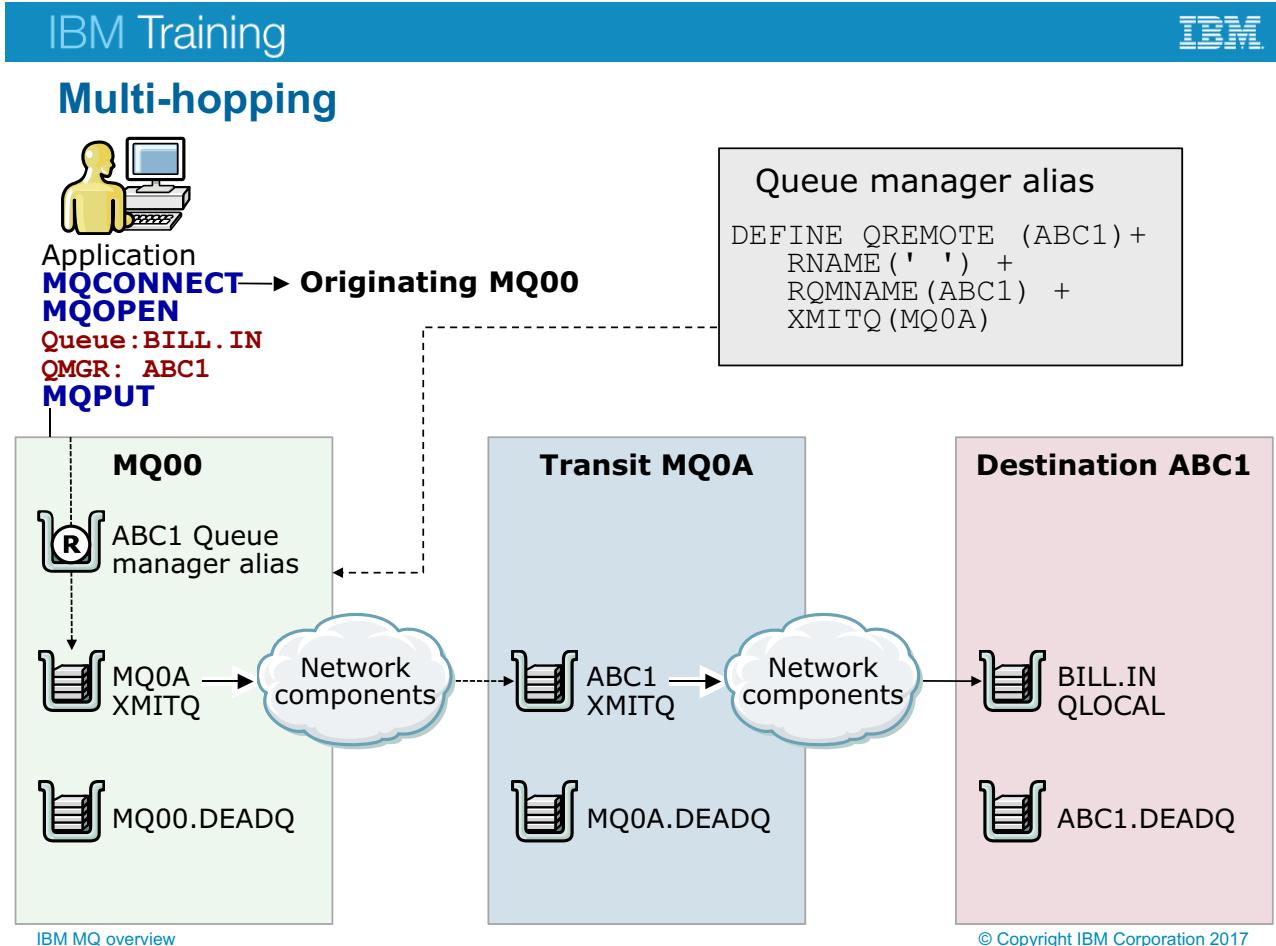


Figure 1-19. Multi-hopping

This visual depicts the case of multi-hopping, where one or more queue managers are traversed on the way to the intended target queue manager.

This slide uses a new term, **queue manager alias**. To define a queue manager alias, you use a QREMOTE object to help the queue manager resolve the routing. Queue manager aliases are more complex to understand, but they make more sense when you complete the queue name resolution topic in a later unit.

The message that the application sent might take the following path:

- At MQ00, all messages for ABC1 are intercepted by the ABC1 queue manager alias; value ABC1 is placed in the transmission header.
- The queue manager alias definition at MQ00 is:

```
DEFINE QREMOTE (ABC1) RNAME(' ') RQMNAME(ABC1) XMITQ(MQ0A)
```

- MQ00 does have a transmission queue defined named MQ0A.

Queue manager MQ0A receives the messages from MQ00 with ABC1 on the transmission queue header, but does not alter the transmission queue header because the name of the destination is already set to queue manager, ABC1. Queue manager MQ0A has a transmission queue that is called ABC1.

Definition of a multi-hopping configuration is an advanced topic. It is shown because it is critical to understand the established IBM MQ adage, “receiver makes good”. A message might hop across several queue managers before reaching its target. If the sending application does data conversion, it might not be the correct conversion if the target queue manager differs from pass-through queue managers. You see more on this topic on the data conversion unit.

Cumulative checkpoint with distributed platforms

At server1.abc.com

- Create, configure, and start queue manager MQM1
- Set dead-letter queue to SYSTEM.DEAD.LETTER.QUEUE
- Define and start listener
 - Set to automatically start with queue manager at port 1641
- Define QLOCAL MQM3 to use as a transmission queue
- Define QREMOTE REQ.OUT to:
 - Use QLOCAL MQM3 as transmit queue
 - Point to remote QLOCAL SALE.IN in queue manager MQM3 as target
- Define sender channel MQM1.MQM3 to:
 - Point to server server3.abc.com port 1643
 - Use MQM3 as its transmission queue
 - After RECEIVER is defined, start channel

At server3.abc.com

- Create, configure, and start queue manager MQM3
- Set dead-letter queue to SYSTEM.DEAD.LETTER.QUEUE
- Define and start listener. Set to automatically start with queue manager at port 1643
- Define QLOCAL SALE.IN
- Define receiver channel MQM1.MQM3

IBM MQ overview

© Copyright IBM Corporation 2017

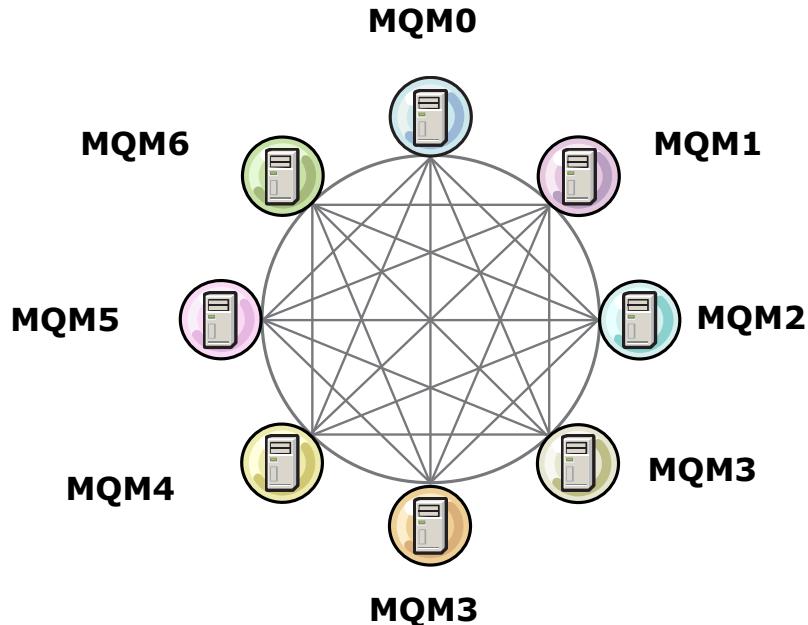
Figure 1-20. Cumulative checkpoint with distributed platforms

This slide summarizes the definitions that are made for the channel pair from MQM1 to MQM3. After you configure and start a queue manager on each side, the definitions that are needed to send a message that uses a remote queue over a sender-receiver channel pair are:

- On the sender side:
 - A local queue of type XMITQ to be used by the QREMOTE and sender channel definitions.
 - Optional: A queue remote. In a coming unit, you learn that a QREMOTE definition is not always needed.
 - A sender channel to the remote queue manager.
- On the receiver, or remote queue manager:
 - A local queue, which is associated with the sender side QREMOTE.
 - A receiver channel of the same name as the sender channel in the sender side.

It is assumed that when you configure the queue manager on each side, you assign a port, which is used in the sender channel unless it is 1414. It is also assumed that you identify a queue to be used as the dead-letter queue. You learn how the dead-letter queue is set in a lab exercise.

Typical queue manager point-to-point channel definitions



IBM MQ overview

© Copyright IBM Corporation 2017

Figure 1-21. Typical queue manager point-to-point channel definitions

If you need to establish point-to-point connectivity across all your queue managers, you end up with numerous channels that point to and from each queue manager. You might have more than one channel pair across the same two queue managers, which perhaps handle messages of different sizes, or different classes of service.

Normally the application developer does not get involved in defining the entire infrastructure, but it is important for the developer to understand the differences in point-to-point definitions. These definitions can be divided into distributed message channels and clustered channels.

When you looked at the sender-receiver pair, you worked with a sender-receiver distributed message channel pair.

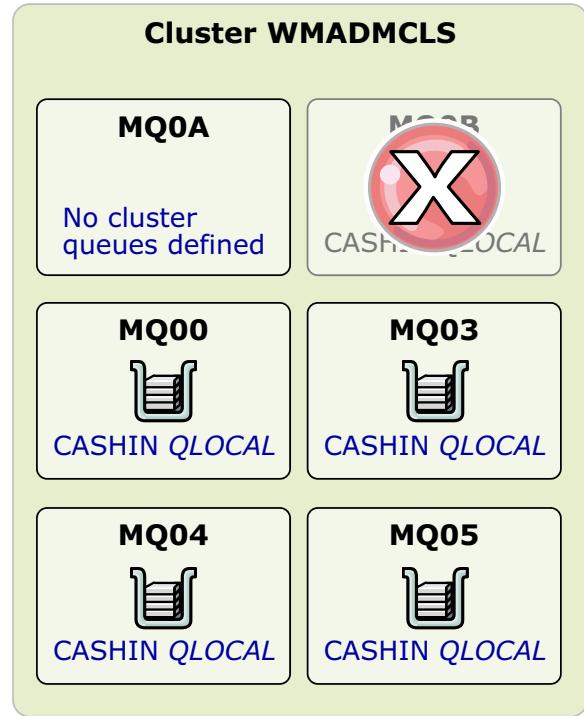
Clustered channels are less labor-intensive for the IBM MQ administrator. Clustered channels pose other considerations to the application developer. These considerations, which involve avoiding the introduction of queue manager affinities, are discussed in a later unit.

You now look at clusters and clustered message channels.

IBM MQ clusters

- Simplified administration
 - Reduce number of remote queues, transmit queues, and channel definitions
- Workload balancing
 - Same queue can be hosted in several queue managers
 - Route around failures
- Scalable
- Contribute to high availability
- MQPUT to local queue manager if queue exists locally
 - If the queue is not found locally, an algorithm is used for target queue selection
- MQGET done to local queue manager
- Publish/subscribe can use cluster

IBM MQ overview



© Copyright IBM Corporation 2017

Figure 1-22. IBM MQ clusters

An IBM MQ cluster is a grouping of queue managers such that the queue managers share knowledge of each other's queues and can exchange messages across queue managers without defining specific channels between all queue managers. After a queue manager "joins" a cluster, it learns about the queues in other clusters, and how to reach those queues. The information that is needed to reach the clustered queues is kept in a queue manager cluster repository.

A repository can be full or partial. A cluster should have two repositories that are designated as full repositories. These two full repositories should know about each other. The queue managers that hold these full repositories are called the full repository queue managers. Other queue managers members of the cluster hold partial repositories. These partial repositories collect information on a "need to know" basis, and also send any news about objects that are defined to one of the full repository queue managers.

Two repositories are the preferred number of full repositories, but not a requirement. Each cluster should have at least two full repositories, and can have more than two. However, more than two repositories might cause management of the cluster to be more complex.

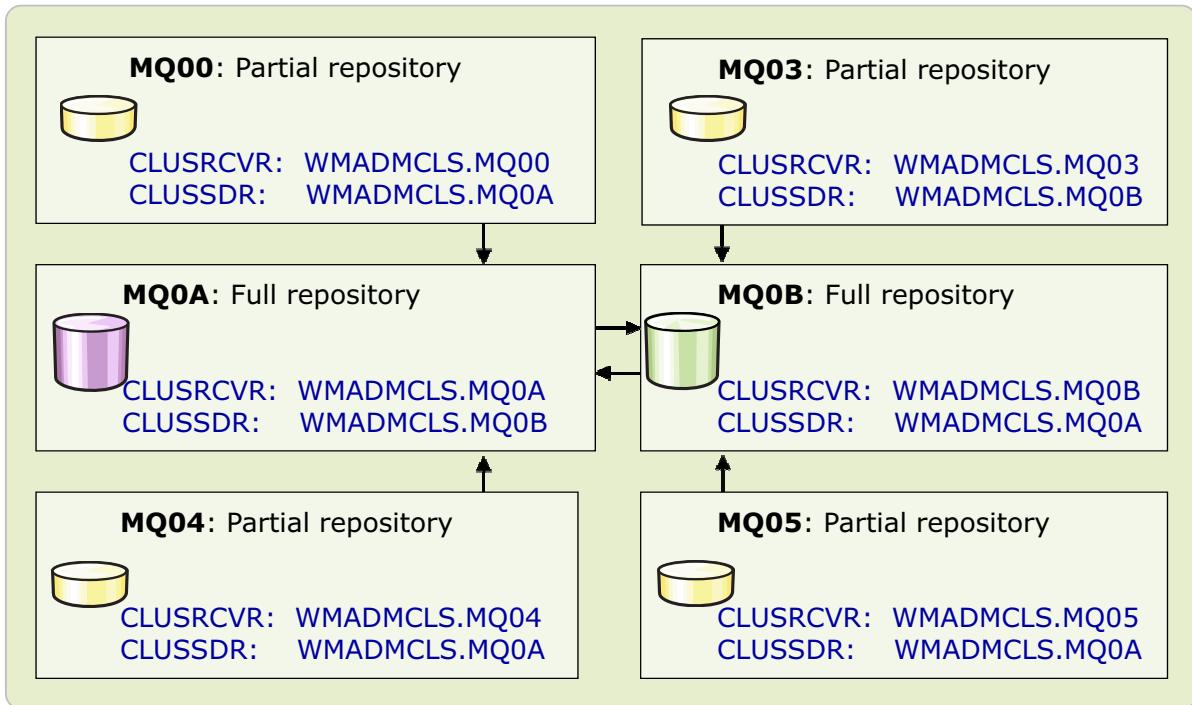
If an application needs to put a message to the CASHIN queue from MQ0A, this queue would be available in queue managers MQ00, MQ03, MQ04, and MQ05. IBM MQ recognizes the member queue manager that is not available, and excludes that queue manager from the potential target queue managers.

As you look at how the clusters are defined, the reduction in the number of IBM MQ object definitions that are required becomes apparent.

The slide qualifies the statement “contribute to high availability” because a cluster is not a high availability solution. Messages in cluster queues in a failed cluster queue manager are inaccessible unless the queue manager is part of a queue-sharing group or configured for high availability. However, clustering contributes to high availability, scalability, and workload balancing for new incoming requests.

MQGET calls are always done to the local queue manager.

Queue manager cluster definition



Note: Arrows represent the full repository that CLUSSDR channels point to

Figure 1-23. Queue manager cluster definition

When a cluster is defined, it is not necessary to manually define channels across all queue manager members of the cluster. For each queue manager member of the cluster, the administrator defines:

- One CLUSRCVR type channel, which unlike the RCVR type channel, provides information on how to get to it, that is, has a CONNAME attribute
- One CLUSSDR channel, which points to *one* of the two full cluster repositories

As you can see, significant work is saved when defining a cluster, as the queue manager takes care of the transmission queues and other details. At a low level, you still have as many channels; however, these channels are dynamically created by using the two cluster channel definitions: CLUSRCVR and CLUSSDR. Now each queue manager can host the same queue, such as the CASHIN queue shown on the previous slide. If one queue manager fails, other queue managers can process the messages. However, notice that a message in a failed queue manager is “marooned” until the failed queue manager restarts, or a failover solution takes over.



Important

Successful implementation of IBM MQ clusters depends on avoiding queue manager affinities when designing and coding an application. Use of certain bind options and message groups, taught in a later unit, introduces queue manager affinities.

What are shared queues?

- Exclusive to z/OS
- Queues whose messages are kept in a coupling facility
- Queue managers in the same z/OS sysplex access the same shared queues by using connectivity via the coupling facility
- When several z/OS queue managers share the same queues, they are called a **queue-sharing group**
- Provide high availability in an IBM MQ infrastructure

Shared queues in a queue-sharing group

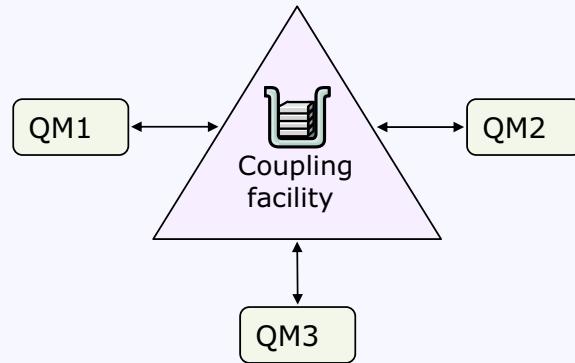


Figure 1-24. What are shared queues?

Shared queues are available on the z/OS platform. Shared queues are *not* to be confused with the SHARE queue attribute, which determines whether more than one application instance can retrieve messages from the queue.

Shared queues are used in queue-sharing groups. A queue-sharing group uses a z/OS component, which is called a coupling facility, to store messages. The queue managers in the queue-sharing group all have access to these messages. Unlike clusters, if a queue manager fails, the message is still available to the other queue managers' members of the group, as the message is kept in the coupling facility accessible to all queue managers. If the coupling facility does not fail, or its space becomes exhausted, the messages are available. Coupling facilities can also be configured with failover technology.

Several techniques can be used to mitigate space consumption in a coupling facility. Queue-sharing groups and coupling facilities are covered in detail in course WM312, *IBM MQ V8 Advanced System Administration for z/OS* or WM313, *IBM MQ V9 Advanced System Administration for z/OS*.

Triggering

- Capability of starting a process by:
 - Configuring attributes of the target queue
 - Defining a PROCESS object in the queue manager
- If the defined trigger conditions are met, triggering is engaged
- The trigger monitor for a **message channel** is called the channel initiator
 - Does not require a PROCESS definition
 - Channel name is included in TRIGDATA attribute
- IBM MQ for z/OS provides the following trigger monitors:
 - CKTI for use with CICS
 - CSQQTRMN for use with IMS
 - Channel initiator started task for channels

IBM MQ overview

```

QUEUE(BILL_IN)
TYPE(QLOCAL)
CRDATE(2014-02-26)
CURDEPTH(1)
DEFBIND(OPEN)
DEFPSIST(NO)
INITQ(SYSTEM.DEFAULT.)
INITIATION.QUEUE)
TRIGGER
PROCESS(BILLING.PROCESS)
QDEPTHHI(80)
TRIGDATA( )
TRIGDPTH(1)
TRIGTYPE(FIRST)
USAGE(NORMAL)

```

© Copyright IBM Corporation 2017

Figure 1-25. Triggering

IBM MQ provides the capability to start a process upon the arrival of messages in a queue. This process is called triggered. What is triggered is the local queue.

Two types of triggering are available: triggering done to start an application, and triggering done to start a channel upon receipt of messages in the channel's associated transmit queue.

The processes have two main differences:

- Triggering an application requires that a process is associated with the triggering definition. Triggering for a channel does not require a process definition.
- The process that reacts to the trigger event and starts the application or process is called a trigger monitor. The process that starts a channel is called a channel initiator.

IBM MQ for z/OS provides CICS and IMS trigger monitors. Distributed IBM MQ has trigger monitors for applications.

You look at how to trigger a process in more detail in the next slide.

IBM Training



Triggering: Process scenario

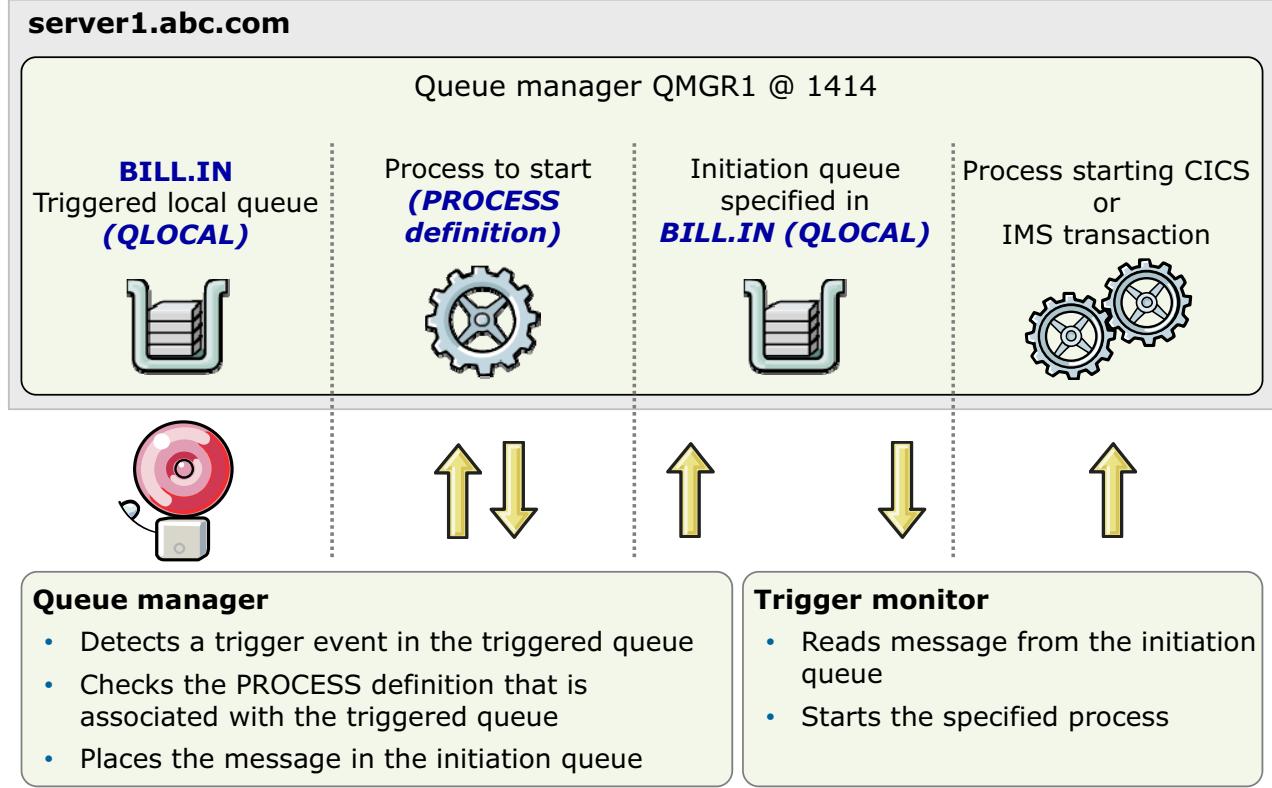


Figure 1-26. Triggering: Process scenario

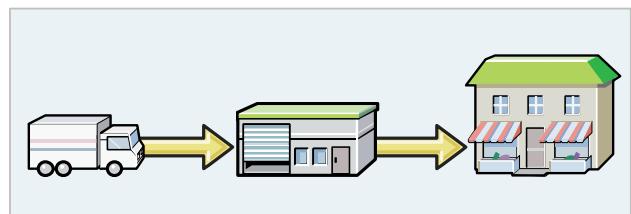
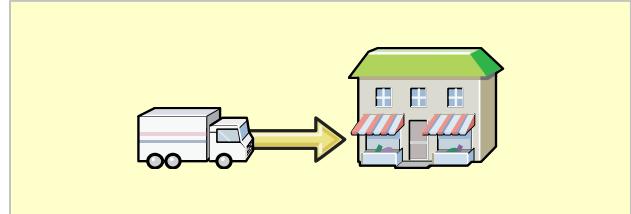
Triggering basics for a process include the following information:

- The queue manager detects a “trigger event” in the triggered queue. This event matches the frequency and number of messages that are required to cause the trigger specifications in the queue.
- After detecting the trigger, the queue manager checks the process definition that is associated with the triggered queue. The process definition is a defined IBM MQ object.
- After obtaining the information from the process definition, the queue manager places a specially formatted message with the information that is required to start the process in the initiation queue that is associated with the triggered queue.
- The trigger monitor that checks the initiation queue reads the message from the initiation queue and starts the process.

This process is similar for trigger starting channels, except that the process definition is not required, and the trigger monitor is the channel initiator.

Messaging styles

- Point-to-point application
 - Sends messages to a predefined destination
 - Application does not need to know where the destination is because IBM MQ locates the target by using object definitions
- Publish/subscribe application
 - Publishes messages to an interim destination according to a topic
 - Interested recipients subscribe to the topic
 - No explicit connection between publishing and subscribing applications



Unless publish/subscribe is specifically mentioned, subsequent topics apply to the point-to-point messaging style

Figure 1-27. Messaging styles

The two messaging styles are point-to-point and publish/subscribe.

With point-to-point messaging, you know where the message is going. The publish/subscribe messaging style is further decoupled. In publish/subscribe, the producers of messages send or publish messages to an interim place where the consumers or subscribers obtain or subscribe to a topic. This topic or topic string is associated to the messages.

You look at publish/subscribe in a later unit. The topics in the rest of this unit apply to point-to-point messaging.

IBM MQ administrative capabilities

- Equivalent options to perform common actions on IBM MQ objects such as define, alter, display, delete, and show status
- IBM MQ scripts (MQSC) that use `rwmqsc` in distributed platforms, `CSQUTIL` on z/OS:
 - `DEFINE QLOCAL`
 - `DEFINE CHANNEL`
 - `DIS Q`
 - `DIS CHSTATUS`
- IBM MQ Explorer
- Programmable command formats

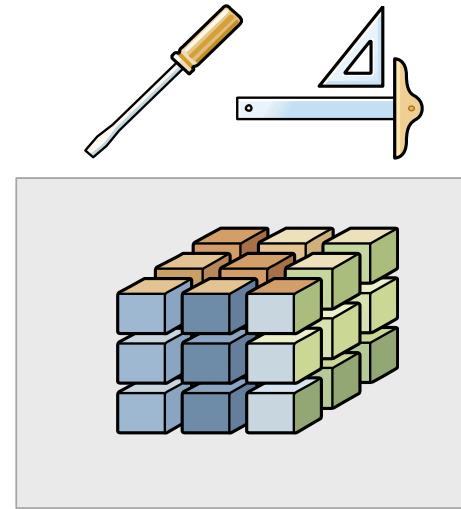


Figure 1-28. IBM MQ administrative capabilities

IBM MQ offers several options for administration. Which option you use depends on what is most appropriate for the task at hand and your personal preference. For instance:

- When creating many objects, maybe associated with a new application, you might want to enter these definitions in a file and define them using the *run IBM MQ script command*, or `rwmqsc` utility. In the lab exercises for this course, you make extensive use of the `rwmqsc` utility.
- You can also use IBM MQ Explorer for most administrative functions, such as creating queue manager, queues, and channels. You also use IBM MQ Explorer in your labs.

Programmable command formats, or PCFs, are more suited for automation, and their use includes application code so it is less likely to be used unless a specific application is coded.

QLOCAL partial attributes and defaults

- When defining a QLOCAL, the only required parameter is the queue name
 - When an attribute is not specified in the definition, it is given a default value
- Some of the attributes of a local queue definition get populated by the queue manager:
 - QUEUE shows queue name
 - Create date (CRDATE)
 - Count of existing messages in the queue, or current depth (CURDEPTH)
- The persistence attribute (DEFPSIST) defaults to non-persistent
 - The persistence attribute determines whether a message survives restarts of the queue. Standards on how and when to use persistence should be established.
- A transmission queue is a QLOCAL with its USAGE attribute set to XMITQ

```
DIS Q(BILL_IN)
QUEUE(BILL_IN)
TYPE(QLOCAL)
CRDATE(2014-02-26)
CURDEPTH(1)
DEFBIND(OPEN)
DEFPSIST(NO)
INITQ( )
NOTRIGGER
PROCESS( )
QDEPTHHI(80)
TRIGDATA( )
TRIGDPTH(1)
TRIGTYPE(FIRST)
MAXMSGL(4194304)
USAGE(NORMAL)
```

IBM MQ overview

© Copyright IBM Corporation 2017

Figure 1-29. QLOCAL partial attributes and defaults

Defining a queue local is simple. To define a queue local without any special requirements, you use a DEFINE QLOCAL followed by the queue name and the queue is created; but many attribute values are defaulted. As you do more work with IBM MQ, you learn the rest of the attributes and their importance.

As you start the development topics, you also learn how your code can override many of the attributes in the queue definition, and test the overrides in an exercise. It is critical to understand this capability, as a developer contributes to the well-being of the queue manager by understanding the requirements and applying them adequately.

Make note of field DEFPSIST. DEFPSIST determines the persistence of the message from the queue object definition. You compare this field in the queue definition attribute to the corresponding field in the message descriptor structure in a later slide.

When is it critical to preserve a message? What is done with time sensitive messages that become obsolete? You learn more on these topics later.

**Note**

The queue manager sets some of the queue attributes, for instance, CURDEPTH, which displays how many messages are in the queue; or CRDATE, which displays the date that the queue was created. An application can override many of the attributes of a queue definition. If it is imperative to use a specific attribute, it is a good practice to have the application set it, rather than default to the attribute in the queue definition.

Persistence is an attribute that was introduced earlier in this course. A message's persistence determines whether the message survives restarts of the queue manager. If the message is critical to the application, for instance, is not a simple query, it should be made persistent. Conversely, you want to avoid persistence for messages that might become obsolete and persist in occupying space.

In large organizations, it is possible that someone heard that "messages do not survive restarts of the queue manager", and draft requirements that all messages must be persistent, without adequately analyzing the need for persistence.

QREMOTE complete display less date and time created fields

- A message count (CURDEPTH) attribute is not present in a QREMOTE
- The remote queue points to the name of the transmission queue (XMITQ) where messages that are put to this queue are placed
- The remote queue also points to the destination queue, or name of the local queue in the remote queue manager (RNAME) where messages are transmitted
- The remote queue manager name attribute (RQMNAME) of the remote queue definition contains the name of the queue manager where messages are transmitted

```

QUEUE (REQ.OUT)
TYPE (QREMOTE)
CLUSNL( )
CLUSTER( )
CLWLPRTY(0)
CLWLRank(0)
CUSTOM( )
DEFBIND(OPEN)
DEFPRTY(0)
DEFPSIST(NO)
DEFPRESP(SYNC)
DESCR( )
PUT(ENABLED)
RQMNAME (QM3)
RNAME (SALE.IN)
SCOPE(QMGR)
XMITQ (QM3)

```

Figure 1-30. QREMOTE complete display less date and time created fields

As you see later in the *queue name resolution* topic, it is not imperative to have a QREMOTE definition to send a message to another queue manager. Use of a queue remote does make sending the messages less code dependent.

Notice that no CURDEPTH attribute is displayed in the QREMOTE attributes. The CURDEPTH is missing because the QREMOTE object itself never holds messages. As you see in later slides, messages that are placed in a QREMOTE might be found in:

- The transmit queue that is associated with the QREMOTE. The transmit queue is specified in the QREMOTE XMITQ attribute.
- The dead-letter queue of the local or remote queue manager. Here the “local” dead-letter queue is the dead-letter queue that is located in the same queue manager as the QREMOTE definition.

The target queue is the QLOCAL in the remote queue manager that the QREMOTE RNAME attribute points to.

Message descriptor (MQMD) header (1 of 2)

```
*****Message descriptor****

StrucId : 'MD' Version : 2
Report : 0 MsgType : 8
Expiry : -1 Feedback : 0
Encoding : 546 CodedCharSetId : 437
Format : 'MQSTR'
Priority : 0 Persistence : 0
MsgId : X'414D5120514D475231202020202020E0A5435620002302'
CorrelId : X'000000000000000000000000000000000000000000000000000000000000000'
BackoutCount : 0
ReplyToQ : '
ReplyToQMgr : 'QMGR1'
'
** Identity Context
UserIdentifier : 'nomad789'
AccountingToken :
X'1601051500000F2728C0B8A01516C8396B659E8030000000000000000000000000000000B'
```

Figure 1-31. Message descriptor (MQMD) header (1 of 2)

These next two slides provide a first look at a message that is preceded by the message descriptor, or MQMD structure.

It is a first look at the MQMD structure. In this course, you learn to work with this structure to control many aspects of your messages.

Notice how the start of the structure shows a structure ID (StrucId) and Version fields. In a later unit, you learn the importance of the Version field.

To the right of the Priority field, note the Persistence field. Depending on your code, you might set this field to override the DEFPSIST field of the queue definition attribute.



Message descriptor (MQMD) header (2 of 2)

IBM MQ overview

© Copyright IBM Corporation 2017

Figure 1-32. Message descriptor (MQMD) header (2 of 2)

The diagram shows the second part of the MQMD structure.

Note how the application that put the message is identified as `amqspput.exe` in field PutAppName. `amqspput.exe` is the distributed program to put messages, also available in source form.

A later unit teaches how to interpret the PutApplType, and how to use or interpret the remaining fields.

The message data is shown at the end of the MQMD structure.

The information in these two MQMD slides was obtained by using sample program amqsbcg.

`amqsbcbg` browses a message, leaving it in the queue, formats the MQMD descriptor, and dumps out the message data. In later units, you learn the difference about code that is used to browse (*non-destructive*), and code that is used to remove messages from a queue (*destructive*).

Transactions: Terminology

- A **resource manager** is a system that owns and controls its components such as
 - A database manager owns its tables
 - A queue manager owns its queues
- A **transaction** or **unit of work** is a set of changes that must be completed in their entirety (**committed**) or restored to a previous consistent state (**backed out**)
- A **transaction manager** is a subsystem that coordinates units of work
- A IBM MQ queue manager can:
 - Act as a transaction manager over its own resources
 - Manage updates to DB2 tables
 - Run under a compatible external transaction manager

Transaction



Funds requested



Account debited



Funds dispensed

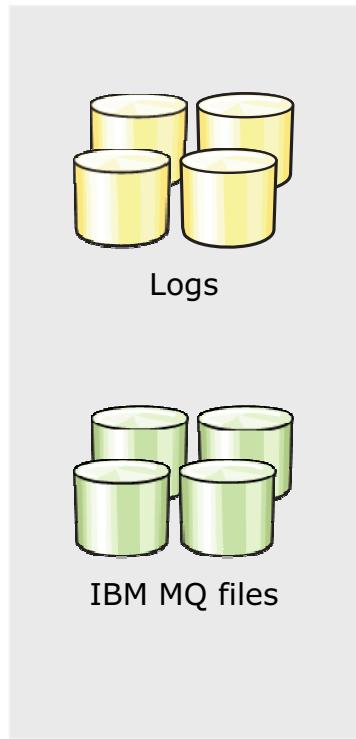
An ATM withdrawal is a common scenario of a process that needs to be coordinated with a transaction manager

Figure 1-33. Transactions: Terminology

This slide baselines transaction-related terminology by using a common example of a transaction: an automated teller machine (ATM), or cash point withdrawal. When you request funds from an ATM, the background program removes the funds from your account and dispenses the funds in the machine. However, if something malfunctions after the funds are removed, but before you are able to access your funds, then what happens? Transactions play a role in such situations. Regarding your withdrawal, if a failure to complete the withdrawal is detected, having your code work as a transaction causes the removal of funds from your account to be reversed if the dispensing component fails.

IBM MQ can act as a transaction manager, and can also be part of a transaction under a compatible external transaction manager. You learn more about transactions and synchronization points in a later unit.

IBM MQ design and development impact behind the scenes



- Unnecessary use of persistence fills the IBM MQ logs 
- Long-running transactions affect the IBM MQ logs 
- Large messages affect the IBM MQ queue files 
- Designs requiring that a set of messages is received in sequence might affect the ability to implement IBM MQ clusters 
- Application designs where messages are left on the queue for a long time and no application removes the messages' impact space in the IBM MQ files

IBM MQ overview

© Copyright IBM Corporation 2017

Figure 1-34. IBM MQ design and development impact behind the scenes

As an IBM MQ application developer, you must be aware and vigilant about choices that might adversely affect the queue manager and messaging infrastructure. You might work in different types of organization, from small to large. Regardless of the size, a common occurrence is that when a queue manager fails, particularly a production queue manager, management contacts an administrator. When the administrator determines the cause of failure, if this cause was a “misbehaved application”, the application developer is contacted next. By this time, it is with a larger escalation.

No assurances are available about the level of IBM MQ knowledge that the resource that is passing the application requirements to you might possess. Sometimes an individual hears that non-persistent messages do not survive restarts of the queue manager, and always requests that messages be made persistent. However, is a score or a stock quote valid after a certain time? Why keep messages that use up space in the IBM MQ files that contain obsolete information?

In the early days of IBM MQ, an unwritten rule stated that you must not use a queue as a database. This rule holds true today. If an application requires database functions, the use of queues to store the information should be questioned. The outage that is a result of the misuse of the technology might surface at the most critical time in the business cycle.

This slide summarizes some practices that should be avoided. Subsequent units in this course also mention areas where care should be exercised, such as coding an application that contains queue manager affinities by using BIND options in the code.

Application environments often start small, and might not plan for unprecedented growth. Coding standards either did not exist, or might be overlooked during development. When the infrastructure architect plans to scale the environment by using clusters, the architect needs to make sure that no queue manager affinities were introduced in the existing applications. Many efforts to scale the environment are delayed due to applications that do not scale due to affinities.

While your job as a developer is to create the code, you also have responsibilities to communicate and question requirements when you are asked to code applications that:

- Exhaust the logging environment by making all messages persistent, or by having long-running units of work
- Exhaust the queue storage by using IBM MQ as a database and not removing messages promptly
- Introduce queue manager affinities by depending on the arrival of a group of messages in a set order or by coding bind options that introduce queue manager affinities

The use of message groups, or the need for binding an application to a specific queue manager, might be unavoidable.

It is your responsibility as a developer to consider the consequences, or at least raise a concern, when a development choice that constrains the environment is requested.

Terminology checkpoint: Security areas

- **Identification:** Determine the identity of a person who is using a system or resource
- **Authentication:** Proving that the person or resource that is being identified is who it claims to be
- **Access control/authorization:** Limiting access to resources to only those people who need it
- **Confidentiality:** Protection of sensitive information
- **Data integrity:** Ability to detect tampering with critical data
- **Auditing:** Ability to determine whether any unauthorized access was done or attempted

* *The terms “access control” and “authorization” are used interchangeably*

Figure 1-35. Terminology checkpoint: Security areas

This slide contains a baseline of the terminology that is used for general security purposes, also applicable to IBM MQ security.

Security considerations for IBM MQ

- IBM MQ provides mechanisms to address the different security areas with platform-specific facilities

- IBM MQ resources to be secured fall into three areas:
 - **Administer IBM MQ:** Create queue managers, queues, channels, administer security, work with IBM MQ libraries, and access log files
 - **Use of the MQI:** Connect to a channel, open a queue, and publish or subscribe to a topic
 - **Channel security:** Connect to a queue manager, open a transmit queue or target queues, and administer channel initiators and listeners

- Architectural considerations
 - Application-level security
 - Link-level security

Figure 1-36. Security considerations for IBM MQ

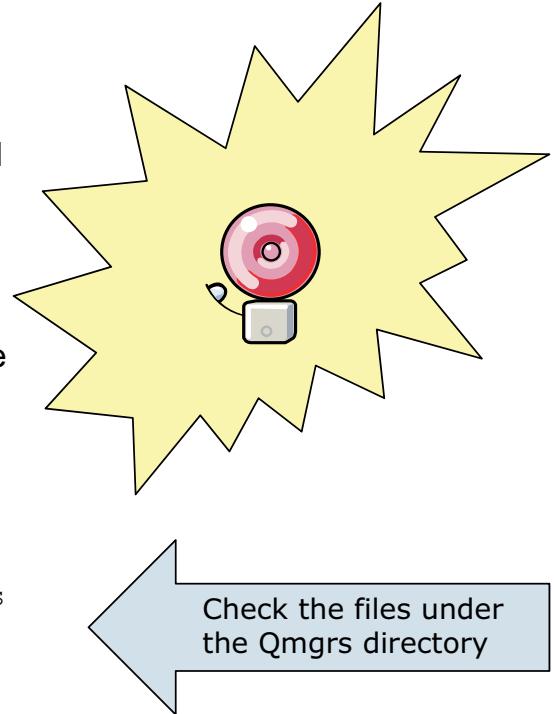
IBM MQ security provides different security capabilities to secure different aspects of IBM MQ. When referring to IBM MQ security, you might hear the following terms:

- Application-level security. This level of security includes the application, from the start of the IBM MQ put or get, all through, if necessary, to the remote queue manager. It includes capabilities such as:
 - Authorization of the user's access to the queue
 - The need to identify whether data at rest must be encrypted
 - Connection authentication

- Link-level security:
 - Applies to channel-to-channel security
 - Might include different technologies, such as object permissions, channel authentication rules, or SSL

Where to look for errors

- Error codes
 - Can be returned to an application
 - Can be the result of issuing a command
 - Can arise from a situation detected by the queue manager, such as low on log space, or channel starting
- Four-character error codes
 - In distributed platforms, you can use the `mqrc` utility
 - Look up errors in the IBM Knowledge Center
- Log files for queue manager-specific errors are at:
`<install_loc>/qmgrs/QMName/errors`
- Log files for IBM MQ product-related errors are at: `<install_loc>/errors`



IBM MQ overview

© Copyright IBM Corporation 2017

Figure 1-37. Where to look for errors

If you find a four-character numeric return code by using a distributed queue manager, you can use the `mqrc` command followed by the return code to obtain a summary line of the error.

1+1=2 Example

You type the command followed by the code in a command-prompt window:

`mqrc 2053`

Response:

`2053 0x00000805 MQRC_Q_FULL`

You can also find error information by using the other resources that are listed in this slide. When using the logs, you need to use the **queue manager logs**, under the `qmtrs` director. Other error logs, which are not under the `qmtrs` directory, contain **IBM MQ product errors**; they do not contain queue manager-related errors.

Key topic review

- Two base product options
 - Server
 - Client
- Queues
 - Hold messages
 - Point to a queue that holds messages
 - Serves as a template for a dynamic (local) queue
- Relative terminology distinctions
 - Local
 - Remote
- Two main categories of channels
 - **Message channels:** Distributed or clustered
 - **Client (or MQI) channels**
- Two messaging styles
 - Point-to-point
 - Publish/subscribe



IBM MQ overview

© Copyright IBM Corporation 2017

Figure 1-38. Key topic review

Much information is covered in this unit. In this slide you review the material that was covered:

- IBM MQ for distributed platforms is available in two installable product options: the server and the client. Other packaging options are available for IBM MQ for z/OS.
- You learned about different types of queues. Remember, only queues of type QLOCAL hold messages.
- Channels have two main categories: message channels that can be distributed or clustered, and MQI channels for IBM MQ clients. Omitted from the slide is a new third type of channel that was recently introduced to interface with AMQP applications such as IBM MQ Light. The new type of channel is an AMQP channel.
- Messaging has two styles: point-to-point, where the destination for a message is known, and publish/subscribe, which works with topics or topic strings. In publish/subscribe, the messages are “brokered” by the publish/subscribe engine.

Unit summary (1 of 2)

- Explain the advantages of message-oriented middleware
- List the basic IBM MQ components
- Describe the correct terminology to use when working with IBM MQ resources
- Distinguish the various types of queues and how they are used
- List basic IBM MQ application programming interface functions
- Explain queue name resolution
- Explain IBM MQ channels
- Describe how application design affects IBM MQ clusters
- Describe queue sharing groups
- Describe the use of triggering in IBM MQ
- Explain the differences between IBM MQ clients and IBM MQ servers
- Distinguish between point-to-point and publish/subscribe messaging styles

[IBM MQ overview](#)

© Copyright IBM Corporation 2017

Figure 1-39. Course summary (1 of 2)

Unit summary (2 of 2)

- Describe attributes that are present in a queue definition
- Explain the message descriptor fields, how they relate to queue attributes, and how they influence application behavior
- Distinguish between local and global units of work
- Describe how design and development decisions impact various IBM MQ resources
- Describe IBM MQ security and how it might impact application development
- Explain where to look for information on IBM MQ errors

Review questions (1 of 2)

1. Which of the following queues can hold messages?
 - a. A remote queue
 - b. The dead-letter queue
 - c. A local queue
 - d. A transmission queue

2. Work is being done in queue manager QM1. This queue manager owns remote queue BILL.OUT, which points to queue manager's QM2 local queue BILL.OUT.DATA. Which statements accurately describe the environment?
 - a. BILL.OUT.DATA is a remotely owned local queue
 - b. BILL.OUT.DATA is a remotely owned remote queue
 - c. BILL.OUT queue holds messages
 - d. QM2 is the remote queue manager



IBM MQ overview

© Copyright IBM Corporation 2017

Figure 1-41. Review questions (1 of 2)

Write your answers here:

1.

2.

Review questions (2 of 2)

3. Select the best answer or answers. What port will a channel with the CONNAME attribute coded as ('server123.enterp.com') point to?
- a. The default IBM MQ port
 - b. The TCP/IP queue manager designated port
 - c. 1414
 - d. The remote queue manager's port



Figure 1-42. Review questions (2 of 2)

Write your answers here:

3.

Review answers (1 of 2)

1. Which of the following queues can hold messages?
 - a. A remote queue
 - b. The dead-letter queue
 - c. A local queue
 - d. A transmission queue
2. Work is being done in queue manager QM1. This queue manager owns remote queue BILL.OUT, which points to queue manager's QM2 local queue BILL.OUT.DATA. Which statements accurately describe the environment?
 - a. BILL.OUT.DATA is a remotely owned local queue
 - b. BILL.OUT.DATA is a remotely owned remote queue
 - c. BILL.OUT queue holds messages
 - d. QM2 is the remote queue manager

The answer is: b, c, and d.

The answer is: a and d.



Review answers (2 of 2)

3. Select the best answer or answers. What port will a channel with the CONNAME attribute coded as ('server123.enterp.com') point to?
- a. The default IBM MQ port
 - b. The TCP/IP queue manager designated port
 - c. 1414
 - d. The remote queue manager's port

The answer is: a and c.



Figure 1-44. Review answers (2 of 2)

Exercise: Working with IBM MQ to find your message

IBM MQ overview

© Copyright IBM Corporation 2017

Figure 1-45. Exercise: Working with IBM MQ to find your message

Exercise objectives

- Determine the status of queue managers in a server
- Start a queue manager
- Use the runmqsc utility and command scripts to create IBM MQ objects and check results
- Put messages to local and alias queues and determine whether the messages arrived at the intended destination
- Determine the trajectory and possible stops of a message put to a remote queue
- Start a sender channel and check the channel status
- Check the queue manager error logs
- Determine where your message is
- Examine the dead letter queue and identify the reason that a message was placed in the queue



Unit 2. Basic design and development concepts

Estimated time

01:00

Overview

This unit introduces the components of the message queue interface, or MQI. You learn about header files, structures, and other items needed for your code. You alter a program to add processing of a second queue. You learn how the attributes you use in your code supersede object definition attributes. You learn about the MQCONN, MQOPEN, MQPUT, MQCLOSE, and MQDISC calls. Finally, you learn how to determine the connection authentication settings of a queue manager, and how to incorporate connection authentication code in your program.

How you will check your progress

- Checkpoint questions
- Lab exercises

References

IBM Knowledge Center for IBM MQ V9 documentation

Unit objectives (1 of 2)

- Describe common messaging patterns
- Explain key architecture and performance considerations for message and application design
- List the available programming options
- Describe the calls, structures, and elementary data types that compose the message queue interface
- Describe message types and message formats
- Explain how to use the MQCONN or MQCONNX calls, and the various options of the MQCONNX call, to connect to a queue manager
- Describe how the MQOPEN, MQPUT, and MQGET calls use the output of the MQCONN or MQCONNX calls

Basic design and development concepts

© Copyright IBM Corporation 2017

Figure 2-1. Unit objectives (1 of 2)

Unit objectives (2 of 2)

- Explain how to use the MQCNO and MQCSP structures with the MQCONN function call to implement connection authentication
- Describe the use of the MQINQ and MQSET calls and the differences between them
- Distinguish the superseding characteristics between object and MQI attributes
- Describe how to compile a C program in the Linux and Windows environments

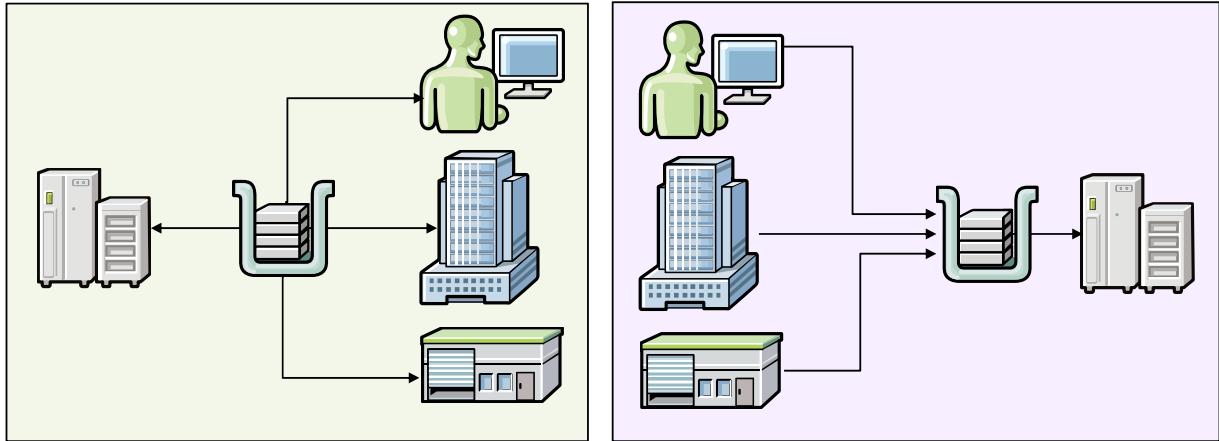
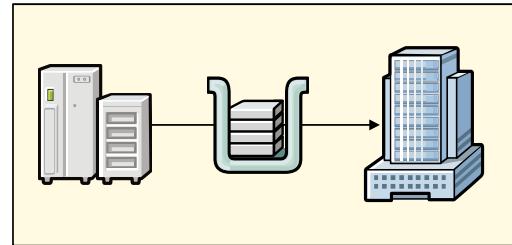
Basic design and development concepts

© Copyright IBM Corporation 2017

Figure 2-2. Unit objectives (2 of 2)

Common messaging patterns

- One-to-one
- One-to-many
- Many-to-one
- Publish/subscribe
- Fire-and-forget
- Request-reply
- Callback



Basic design and development concepts

© Copyright IBM Corporation 2017

Figure 2-3. Common messaging patterns

When you develop IBM MQ applications, you need to address different scenarios with your code:

- A simple application where you either put or get messages to or from one specific location
- A scenario where one application might be sending messages to many applications
- A case where several applications send messages to one application
- Publish/subscribe applications, which use a further decoupled messaging style
- Asynchronous fire-and-forget situations
- A common scenario, a request followed by a reply
- A callback situation, where an application starts a process based on the arrival of messages; not to be confused with IBM MQ triggering

Throughout this course, you look at the IBM MQ function calls and options to handle different messaging patterns.

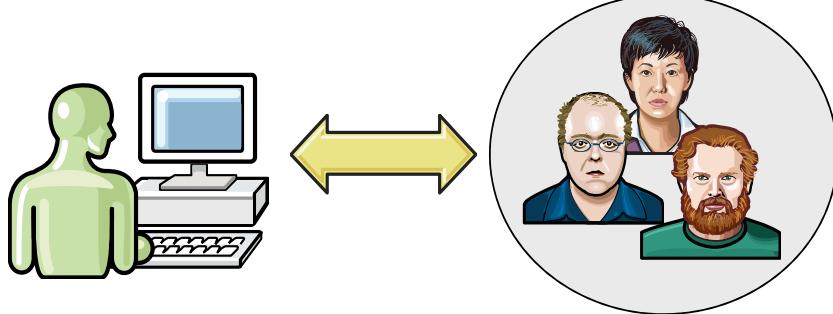
Application design and performance

Design impacts

- Unnecessary use of persistence fills the IBM MQ logs
- Long-running transactions impact the IBM MQ logs
- Large messages impact the IBM MQ queue files
- Sequence requirements
- Leaving messages on queue

Performance impacts

- Persistence
- Message length and performance
- Queues with mixed-length messages
- MQPUT1 call use
- Synchronization point frequency
- Searching for specific messages



Basic design and development concepts

© Copyright IBM Corporation 2017

Figure 2-4. Application design and performance

A significant amount of information is presented in this course. If this information is used inefficiently, or for tasks that are not intended for IBM MQ, applications can exhibit poor performance, or even lead to an outage of the queue manager. Before you start coding an application, you need to look at designs or practices that should be carefully evaluated.

IBM MQ uses disk space to store messages in queues, and to log or persist messages that must survive restarts of the queue manager. Logs are also used to track transactions, that is, a sequence of tasks that must either all complete, or all be backed out. You learn about transactions in a later unit. As you progress through the course, be aware of practices such as:

- Information that is valid for a short time, such as a stock quotation query, does not need to be made persistent. Using persistence unnecessarily is a misuse of the queue manager's disk space.
- An established IBM MQ convention is to “*not use IBM MQ as a database*”. IBM MQ is intended to exchange messages. Messages should be consumed as soon as possible. Conversely, as much as possible messages should not be left in the queues indefinitely, or be selectively retrieved. Messages use up space in the IBM MQ disks, and selective retrieval is not conducive to the best performance.

- How large do your messages need to be? If you look at IBM MQ performance reports, you learn that the larger the message, the less optimal the performance. The information to be included needs to be carefully weighed against the performance requirements.

Design choices also affect the ability to scale the application.

As you progress through this course, you learn details to help you code well behaved applications. Depending on your organizational environment, you might be getting coding requirements from an architect. If requirements call for specifications not conducive to a “well behaved application”, *always question* the intent of the design and the future direction of the application. It is better to intercept potential problems early than after an application goes into production, or the need to scale must be delayed due to queue manager affinities. Code that leads to queue manager affinities is covered in a later unit.

When a queue manager outage occurs, the IBM MQ team and the executive team are called. The next call is to the developer who coded the application that caused the outage.

MQI programming options

Procedural languages

C
Visual Basic (Windows)
COBOL
Assembler language
(IBM MQ for z/OS)
RPG (IBM MQ for IBM i)
PL/I (IBM MQ for z/OS)

Object-oriented languages

.NET
ActiveX
C++
Java
JMS



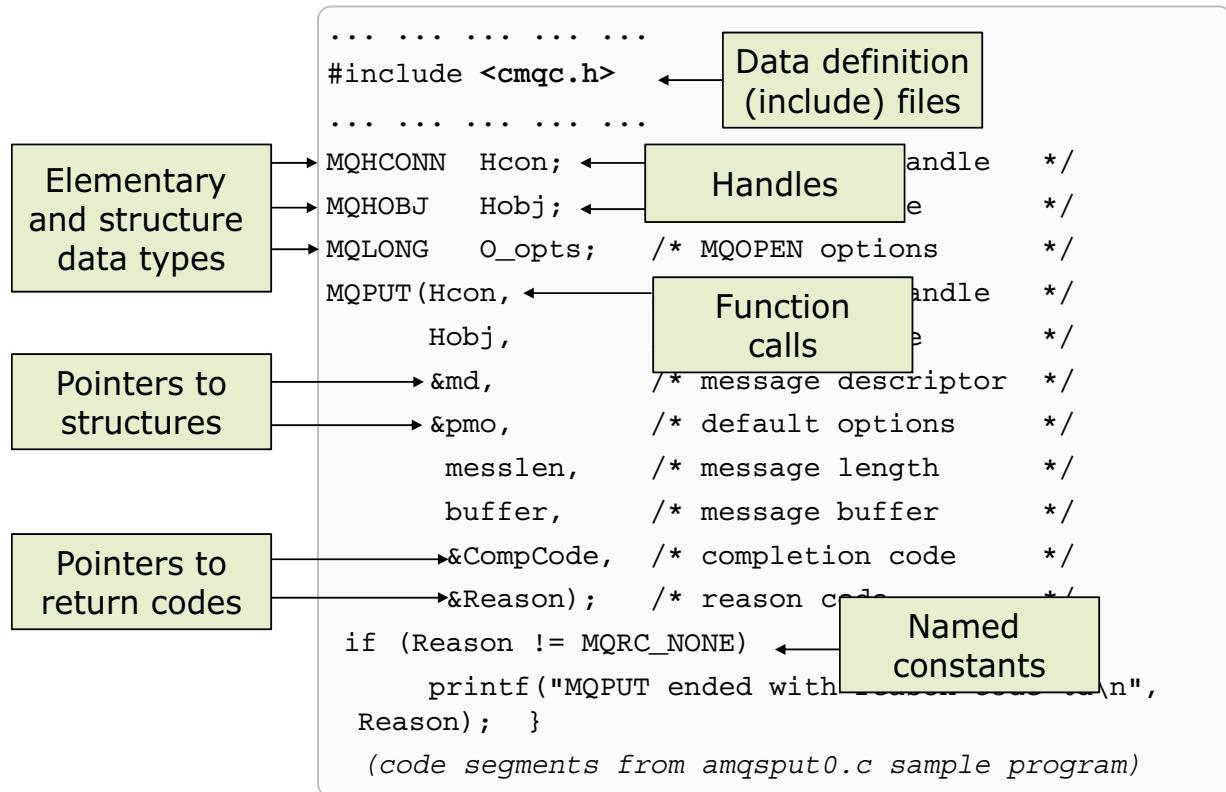
Basic design and development concepts

© Copyright IBM Corporation 2017

Figure 2-5. MQI programming options

IBM MQ applications can be coded in the languages and environments that are indicated in this visual.

Message Queue Interface (MQI)



Basic design and development concepts

© Copyright IBM Corporation 2017

Figure 2-6. Message Queue Interface (MQI)

The IBM MQ application message queue interface, or MQI, has several components to facilitate application development.

- Data definition, or include files. These files might have different names that depend on the platform. These files contain the definitions that are needed for your code. Different files exist for different IBM MQ components.
- Elementary and structure data types.
 - Elementary types are the basic types, which encompass character and numeric data.
 - Structure data types are made up of elementary basic types. Structure data types exist for different IBM MQ function calls and options.
- Named constants hold values and variables that are used in the MQI that you can use to set or question values in your code.
- Handles provide access to your defined instances of a structure.

MQI data definition files

Language	Referred to as:	Letter suffix	MQI data definition file name
Assembler language	Macros	a	cmqa
Visual Basic	Module files	b	cmqb
C	Include or header files	c	cmqc.h
COBOL (without initialized values)	Copy file or copybook	l	cmql
PL/I	Include files	p	cmqp
COBOL (with default values set)	Copy file or copybook	v	cmqv

Linux for C

Language header files:
`/opt/mqm/inc
ps -ef | grep c.h`

```
cmqbc.h
cmqcfc.h
cmqc.h
cmqec.h
cmqpsc.h
cmqstrc.h
cmqxch.h
cmqzc.h
```

Windows for C

Language header files:
`C:\<>\IBM\MQ
\Tools
\c
\include`

cmqbc.h
 cmqc.h
 cmqcfc.h
 cmqec.h
 cmqpsc.h
 cmqstrc.h
 cmqxch.h
 cmqzc.h

Basic design and development concepts

© Copyright IBM Corporation 2017

Figure 2-7. MQI data definition files

The MQI has data definition files, which contain the structures, fields, values, and other components you use in your code.

These data definition files might be referred to with different names, which depend on the language and platform, for example, header files for the C language versus copybooks for COBOL.

You might need to use one or more of these files. This rest of this course refers to files that are used for the C language.

The most commonly used file, which contains most of the needed definitions, is `cmqc.h`. However, sometimes you might need to include more than one file. For example, if you are connected to a queue manager, and use an MQCONN option to create your originating channel in the code, you must also include the `cmqxch.h` header. This header contains the needed channel definition, or `MQCD` structure.

A look at selected lines of the cmqc.h header file

```

25  /* FUNCTION: This file declares the functions, */
26  /* structures and named constants for the main MQI */
...
896 /* Values Related to MQMD Structure
900 #define MQMD_STRUC_ID           "MD"
908 #define MQMD_CURRENT_VERSION     2
913 #define MQMD_CURRENT_LENGTH      364
...
916 #define MQRO_EXCEPTION          0x01000000
...
3589 /* Byte Datatypes */
3590 typedef unsigned char MQBYTE;
3591 typedef MQBYTE MQPOINTER PMQBYTE;
...
4413 /* MQMD Structure -- Message Descriptor
4421 struct tagMQMD {
4422     MQCHAR4    StrucId;    /* Structure identifier */
4423     MQLONG     Version;    /* Structure version number */
4424     MQLONG     Report;    /* Options for report messages */

```

Named
constants

Elementary
data types

Structure
data types

Basic design and development concepts

© Copyright IBM Corporation 2017

Figure 2-8. A look at selected lines of the cmqc.h header file

This display contains selected definitions from the `cmqc.h` header file. The data types that are used in the MQI (or in exit functions) can be elementary data types, or arrays or structures derived from the elementary data types.

You can further investigate by looking at the `cmqc.h` header file:

```

/* Character Datatypes */
typedef char MQCHAR;
typedef MQCHAR MQPOINTER PMQCHAR;
typedef PMQCHAR MQPOINTER PPMQCHAR;
typedef MQCHAR MQCHAR4[4];
...

```

Elementary data types: Partial list

Elementary data type	Data type	Description
MQBOOL	Boolean	Represents a boolean value.
MQBYTE	Byte	Represents a single byte of data.
MQBYTEn	String of bytes	A string of bytes either 8, 16, 24,32, 40, or 128.
MQCHAR	Character	Represents a single-byte character, or 1 byte of a double-byte or multi-byte character.
MQCHARn	String of characters	A string of n characters. n can be: 4, 8, 12, 20, 28, 32,48, 64, 128, or 256.
MQFLOAT32 MQFLOAT64	32 – 64 bit floating point number	MQFLOAT32 or MQFLOAT64 data types are 32-bit or 64-bit floating-point numbers.
MQHCONN	Connection handle	Represents a connection to a queue manager for use in further MQI calls. No other meaning.
MQLONG	32-bit signed integer	Can take any value in the range -2 147 483 648 through +2 147 483 647.
MQHOBJ	Object handle	An object handle that gives access to an object.
PMQMD	Pointer	Pointer to structure of type MQMD.

Basic design and development concepts

© Copyright IBM Corporation 2017

Figure 2-9. Elementary data types: Partial list

Elementary data types are the most basic IBM MQ development fields. These data types are used in the IBM MQ structure definitions.

The subsets of elementary data types that are presented in this slide are explained in the Description field. IBM MQ has numerous elementary data types. You can learn more about the rest of the elementary data types in the coming units of this course. You can learn about data types at the IBM Knowledge Center, or the IBM MQ Reference manual for the IBM MQ version you need to work with. A copy of the IBM MQ Reference PDF file is available in the VMware workstation for this course.

A later exercise in this course uses two fields that use the MQPTR and MQLONG data types. These fields are defined in the `cmqc.h` header. Use the IBM MQ Reference to check the following fields:

- VSPtr
- VSLength

Structure data types that are used on MQI calls: Partial list

Structure	Description	Calls where used
MQOD	Object descriptor	MQOPEN, MQPUT1
MQMD	Message descriptor	MQBUFMH, MQMHBUF, MQCB, MQGET, MQPUT, MQPUT1
MQPMO	Put message options	MQPUT, MQPUT1
MQCNO	Connection options	MQCONN
MQCSP	Security parameters	MQCONN
MQGMO	Get message options	MQGET
MQSD	Subscription descriptor	MQSUB
MQBO	Begin options	MQBEGIN
MQCBD	Callback descriptor	MQCB
MQCTLO	Callback options	MQCTL

Basic design and development concepts

© Copyright IBM Corporation 2017

Figure 2-10. Structure data types that are used on MQI calls: Partial list

The various structure data types are documented in the IBM Knowledge Center and in the IBM Reference manual.

This slide contains some of the most commonly used structure data types, described within the corresponding row in the slide.

As you work through the units in this course, you become more familiar with the object descriptor, message descriptor, connection options, put message options, get message options, and many of the other structures.

Structure data types that are used with message data: Partial list

Structure	Calls where used
MQCIH	CICS information header
MQCFH	PCF header
MQDLH	Dead-letter header (undelivered message)
MQIIH	IMS information header
MQMDE	Message descriptor extension
MQRFH	Rules and formatting header
MQRFH2	Rules and formatting header 2
MQTM	Trigger message
MQTMC2	Trigger message (character format 2)
MQXQH	Transmission queue header

Figure 2-11. Structure data types that are used with message data: Partial list

The data types that are used in the MQI are either:

- Elementary data types, or
- Aggregates of elementary data types (arrays or structures)

MQMD message descriptor structure (partial)

```

MQCHAR4  StrucId;      /* Structure identifier          */
MQLONG    Version;     /* Structure version number    */
MQLONG    Report;      /* Report message options      */
MQLONG    MsgType;     /* Message type                */
MQLONG    Expiry;      /* Message lifetime            */
MQLONG    Feedback;    /* Feedback or reason code    */
MQLONG    Encoding;    /* Msg data numeric encoding */
MQLONG    CodedCharSetId; /*Msg data char set idnt */
MQCHAR8   Format;      /* Message data format name   */
MQLONG    Priority;    /* Message priority            */
MQLONG    Persistence; /* Message persistence         */
MQBYTE24  MsgId;      /* Message identifier          */
MQBYTE24  CorrelId;   /* Correlation identifier      */
...

```

Basic design and development concepts

© Copyright IBM Corporation 2017

Figure 2-12. MQMD message descriptor structure (partial)

A good example of a structure is the MQMD, or message descriptor structure. You use this structure in several labs. Some of the uses of the structure in the labs are:

- Use of message expiry
- Generating report messages
- Requesting that a message survives restarts of the queue manager
- Correlating requests with replies

You learn more about the MQMD in an upcoming unit.

amqspput0.c elementary and structure data definitions

```

MQOD      od = {MQOD_DEFAULT};          /* Object Descriptor      */
MQMD      md = {MQMD_DEFAULT};          /* Message Descriptor     */
MQPMO     pmo = {MQPMO_DEFAULT};        /* put message options   */
MQCNO     cno = {MQCNO_DEFAULT};        /* connection options    */
MQCSP     csp = {MQCSP_DEFAULT};        /* security parameters   */
MQHCONN   Hcon;                      /* connection handle     */
MQHOBJ    Hobj;                      /* object handle         */
MQLONG    O_options;                  /* MQOPEN options        */
MQLONG    C_options;                  /* MQCLOSE options       */
MQLONG    CompCode;                  /* completion code       */
MQLONG    OpenCode;                  /* MQOPEN completion code */
MQLONG    Reason;                   /* reason code           */
MQLONG    CReason;                  /* reason code for MQCONNX */
MQLONG    messlen;                  /* message length        */
char      buffer[65535];             /* message buffer         */
char      QMName[50];                /* queue manager name    */
char      *UserId;                  /* UserId for authentication */
char      Password[MQ_CSP_PASSWORD_LENGTH + 1] = {0}; /* auth */

```

Basic design and development concepts

© Copyright IBM Corporation 2017

Figure 2-13. amqspput0.c elementary and structure data definitions

The display shows another look at the code segment that initializes some of the IBM MQ structures. Note how each structure has a corresponding initialization structure. In some cases, a structure can have one or more initialization structures, and you need to know the correct one to select.

The items signaled by the letter are elementary data types.

The letter **b** is pointing to the initialization structures corresponding to each declared structure. You must be careful to review the initialized values to ensure the version number that is used is adequate for the feature you need to code for. For example, if you do not set the version of the MQCNO structure correctly, your code that is marked with the letter **d** is ignored.

The letter **c** points to various declarations and fields that the function calls require.

Named constants

- Fixed, default, and initial values for to assist with programming tasks
- Provided in the include or header files
- Different types of constants that include values such as:
 - Named, user readable options to use in function calls
 - Named values to query a process
 - Return codes to determine status of a call
 - Initialization values for structures

```
/* Open Options */
#define MQOO_BIND_AS_Q_DEF ... ...
#define MQOO_BROWSE     0x00000008
/* Get Message Options */
#define MQGMO_WAIT      0x00000001
/* Persistence Values */
#define MQPER_PERSISTENT    1
#define MQPER_PERSISTENCE_AS_Q_DEF 2
```

```
/* Completion Codes */
#define MQCC_OK          0
#define MQCC_FAILED       2
#define MQRC_NO_MSG_AVAILABLE 2033
#define MQRC_NOT_AUTHORIZED 2035
#define MQRC_Q_FULL        2053
/* Put Application Types */
#define MQAT_IMS_BRIDGE   19
#define MQAT_AMQP          37
```

[Basic design and development concepts](#)

© Copyright IBM Corporation 2017

Figure 2-14. Named constants

A completion code and a reason code are returned as output parameters by each call.

To show whether a call is successful, each call returns a *completion code* when the call is complete. The completion code is typically either MQCC_OK indicating success, or MQCC_FAILED indicating failure. 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. Reason codes denote such circumstances as a queue full, get operations that are not 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.

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. See Call descriptions and select the appropriate call from the list.

MQI function calls (1 of 3)

- Commonly used calls to put or get messages

Call	Description
MQCONN	Connect queue manager
MQCONNX	Connect queue manager (extended)
MQOPEN	Open object
MQPUT	Put message
MQPUT1	Put one message
MQGET	Get message
MQCLOSE	Close object
MQDISC	Disconnect from queue manager
MQSUB	Register subscription
MQSUBRQ	Subscription request

Basic design and development concepts

© Copyright IBM Corporation 2017

Figure 2-15. MQI function calls (1 of 3)

The MQI provides a number of function calls to conduct messaging and queuing actions. This first slide contains the most common function calls.

MQI function calls (2 of 3)

- Synchronization, callback, inquire and set object, and status retrieval calls

Call	Description
MQBEGIN	Begin unit of work
MQBACK	Back out changes
MQCMIT	Commit changes
MQCB_FUNCTION	Callback function
MQCB	Manage callback
MQCTL	Control callbacks
MQINQ	Inquire object attributes
MQSET	Set object attributes (queues only)
MQSTAT	Obtain status information

[Basic design and development concepts](#)

© Copyright IBM Corporation 2017

Figure 2-16. MQI function calls (2 of 3)

This slide shows the second set of function calls.

MQI function calls (3 of 3)

- Calls to work with message properties

Call	Description
MQCRTMH	Create message handle
MQDLTMH	Delete message handle
MQSETPM	Set message property
MQINQMP	Inquire message property
MQDLTMP	Delete message property
MQBUFMH	Convert buffer into message handle
MQMHBUF	Convert message handle into buffer

Basic design and development concepts

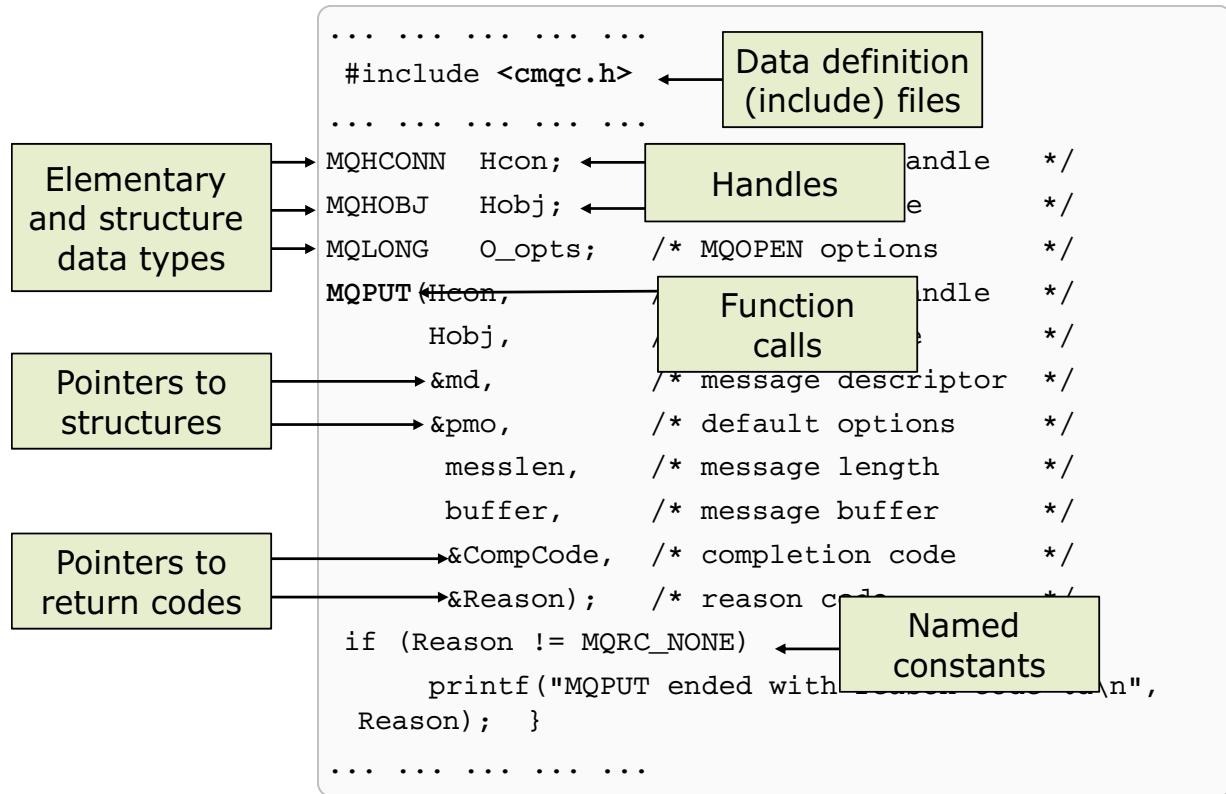
© Copyright IBM Corporation 2017

Figure 2-17. MQI function calls (3 of 3)

Handles

- Connection handle: Hconn
 - Unique identifier that allows a program to communicate with the queue manager
 - Returned as an output by an MQCONN or MQCONNX function call
 - Used as an input parameter for calls that follow MQCONN or MQCONNX
- Object handle: Hobj
 - Unique identifier that allows a program to work with an object
 - Returned as an output by the MQOPEN function call
 - Used as an input parameter for subsequent MQPUT, MQGET, MQINQ, MQSET, and MQCLOSE calls
- Subscription handle: Hsub
 - Identify a specific subscription
 - Used as an input parameter for subsequent MQGET, MQCB, or MQSUBRQ calls
- Message handle, or Hmsg is used by some calls when working with a message property

Message Queue Interface review



Basic design and development concepts

© Copyright IBM Corporation 2017

Figure 2-19. Message Queue Interface review

This graphic is a review of the MQI components.

Recurring considerations in the MQI



- Structures might have several version numbers
- If the version number is not set correctly, the code might not work as expected



- Many attributes that are specified in the code override the attributes of an object definition
- Ensure that you code according to application needs and organization standards



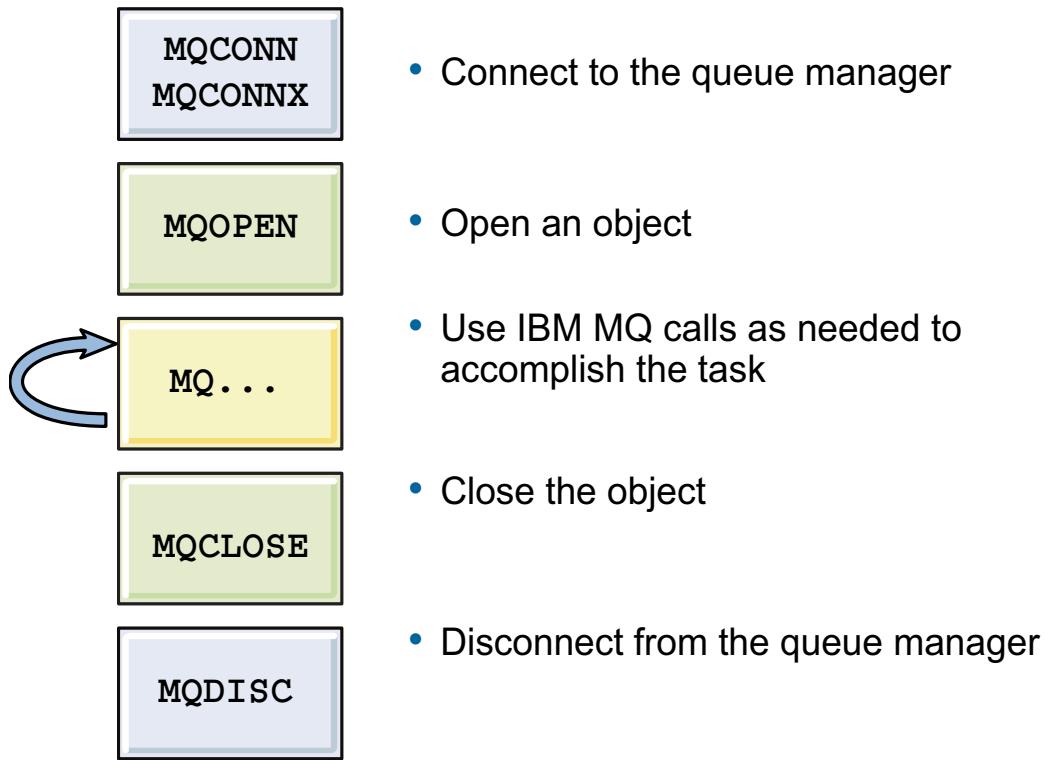
- Some problems that are perceived to be code errors might be queue manager or object constraints
- Develop good habits to research any extra details

Basic design and development concepts

© Copyright IBM Corporation 2017

Figure 2-20. Recurring considerations in the MQI

Common function call sequence



Basic design and development concepts

© Copyright IBM Corporation 2017

Figure 2-21. Common function call sequence

MQCONN

```
MQHCONN    Hcon;           /* connection handle      */
MQLONG     CompCode;       /* completion code       */
MQLONG     CReason;        /* reason code          */
char       QMName[50];     /* queue manager name   */

MQCONN (QMName,       /* queue manager         */
&Hcon,             /* connection handle    */
&CompCode,          /* completion code      */
&CReason);          /* reason code          */
```

Basic design and development concepts

© Copyright IBM Corporation 2017

Figure 2-22. MQCONN

MQCONNX

```

MQCNO  cno = {MQCNO_DEFAULT} /* connection options*/
MQCSP   csp = {MQCSP_DEFAULT};/* security parms      */
MQHCONN Hcon;           /* connection handle      */
MQLONG  CompCode;        /* completion code       */
MQLONG  CReason;         /* reason code          */
char    QMName[50];       /* queue manager name   */
char    *UserId;          /* Id for authentication */

```

**MQCONNX (QMName, ➔ /* queue manager */
 &cno, ➔ /* connection options */
 &Hcon, ➙ /* connection handle */
 &CompCode, ➙ /* completion code */
 &CReason); ➙ /* reason code */**

Basic design and development concepts

© Copyright IBM Corporation 2017

Figure 2-23. MQCONNX

Version-dependent attributes in structures: Example MQCNO

```

MQCHAR4  StrucId;          /* Structure identifier      */
MQLONG   Version;          /* Structure version number */
MQLONG   Options;          /* MQCONNX actions control opts */
/*Ver:1 */ < Following fields recognized if MQCNO_VERSION_2 or higher
MQLONG   ClientConnOffset; /* Client conn MQCD struct offset*/
MQPTR    ClientConnPtr;   /* Client conn MQCD struct addr */
/* Ver:2 */ < Following fields recognized if MQCNO_VERSION_3 or higher
MQBYT128 ConnTag;         /* Queue-manager connection tag */
/* Ver:3 */ < Following fields recognized if MQCNO_VERSION_4 or higher
PMQSCO   SSLConfigPtr;    /* Client conn MQSCO struct addr */
MQLONG   SSLConfigOffset; /* Client conn MQSCO str offset */
/* Ver:4 */ < Following fields recognized if MQCNO_VERSION_5 or higher
MQBYTE24 ConnectionId;    /* Unique Connection Identifier */
MQLONG   SecurityParmsOffset; /* Offset of MQCSP structure */
PMQCSP   SecurityParmsPtr; /* Address of MQCSP structure */

```

Basic design and development concepts

© Copyright IBM Corporation 2017

Figure 2-24. Version-dependent attributes in structures: Example MQCNO

When you work with structures, always consider:

- What version number needs to be set in the structure to accomplish the required tasks? Failure to set the correct version can cause otherwise correct code to be in error. IBM MQ ignores any requests that require the specific version. Omitting the version number is a common mistake.
- Use the correct initialization structure for your declared structures. However, use of the initialization structure does not ensure use of the needed structure version number. Check the `cmqc.h` header file to confirm the values set by the initialization structure. Then, if needed, update the structure version number.

The initialization structure for the MQCNO structure is shown in the example display that follows. Notice that the version number is 1. If you need to code any of the connection authentication options, you must set the version number to `MQCNO_VERSION_5`, or IBM MQ ignores any code that uses the security fields. You confirm this behavior in one of the exercises.

1+1=2 Example

```
#define MQCNO_DEFAULT {MQCNO_STRUC_ID_ARRAY}, \
    MQCNO_VERSION_1, \
    MQCNO_NONE, \
    0, \
    NULL, \
    {MQCT_NONE_ARRAY}, \
    NULL, \
    0, \
    {MQCONNID_NONE_ARRAY}, \
    0, \
    NULL
```

MQCNO options

Default option:

MQCNO_NONE

Binding options:

- MQCNO_STANDARD_BINDING
- MQCNO_FASTPATH_BINDING
- MQCNO_SHARED_BINDING
- MQCNO_ISOLATED_BINDING
- MQCNO_LOCAL_BINDING
- MQCNO_CLIENT_BINDING

Handle-sharing options:

- MQCNO_HANDLE_SHARE_NONE
- MQCNO_HANDLE_SHARE_BLOCK
- MQCNO_HANDLE_SHARE_NO_BLOCK

Conversation sharing options:

- MQCNO_NO_CONV_SHARING
- MQCNO_ALL_CONVS_SHARE

Reconnection options:

- MQCNO_RECONNECT_AS_DEF
- MQCNO_RECONNECT
- MQCNO_RECONNECT_DISABLED
- MQCNO_RECONNECT_Q_MGR

Conversation-sharing options:

- MQCNO_NO_CONV_SHARING
- MQCNO_ALL_CONVS_SHARE

Accounting options:

- MQCNO_ACCOUNTING_MQI_ENABLED
- MQCNO_ACCOUNTING_MQI_DISABLED
- MQCNO_ACCOUNTING_Q_ENABLED
- MQCNO_ACCOUNTING_Q_DISABLED

Channel-definition options:

- MQCNO_CD_FOR_OUTPUT_ONLY
- MQCNO_USE_CD_SELECTION

Tag and trace options in notes

Basic design and development concepts

© Copyright IBM Corporation 2017

Figure 2-25. MQCNO options

Other options are:

- MQCNO_SERIALIZE_CONN_TAG_Q_MGR
- MQCNO_SERIALIZE_CONN_TAG_QSG
- MQCNO_RESTRICT_CONN_TAG_Q_MGR
- MQCNO_RESTRICT_CONN_TAG_QSG
- MQCNO_ACTIVITY_TRACE_ENABLED
- MQCNO_ACTIVITY_TRACE_DISABLED

Connection security (MQCSP) parameter structure

```

typedef struct tagMQCSP MQCSP;
typedef MQCSP MQPOINTER PMQCSP;
struct tagMQCSP {
    MQCHAR4 StrucId;      /* Structure identifier      */
    MQLONG Version;       /* Structure version number */
    MQLONG AuthenticationType; /* Type of auth      */
    MQBYTE4 Reserved1;     /* Reserved          */
    MQPTR CSPUserIdPtr;   /* Address of user ID   */
    MQLONG CSPUserIdOffset; /* Offset of user ID   */
    MQLONG CSPUserIdLength; /* Length of user ID   */
    MQBYTE8 Reserved2;     /* Reserved          */
    MQPTR CSPPasswordPtr; /* Password address   */
    MQLONG CSPPasswordOffset; /* Password offset   */
    MQLONG CSPPasswordLength; /* Password length   */ };

```

[Basic design and development concepts](#)

© Copyright IBM Corporation 2017

Figure 2-26. Connection security (MQCSP) parameter structure

The MQCSP structure is one of the newer structures for connection authentication. The MQCNO structure points to this structure. However, if the MQCNO version number is not set correctly, the MQCSP structure is ignored.

Implementing connection authentication in MQCONN

```
MQCNO cno = {MQCNO_DEFAULT}; /* connection options*/
MQCSP csp = {MQCSP_DEFAULT}; /* security parameters*/

cno.SecurityParmsPtr = &csp;
cno.Version = MQCNO_VERSION_5;
csp.AuthenticationType = MQCSP_AUTH_USER_ID_AND_PWD
csp.CSPUserIdPtr = UserId;
csp.CSPUserIdLength = strlen(UserId);
csp.CSPPasswordPtr = Password;
csp.CSPPasswordLength = strlen(csp.CSPPasswordPtr);
```

Basic design and development concepts

© Copyright IBM Corporation 2017

Figure 2-27. Implementing connection authentication in MQCONN

The code snippet in the display implements connection authentication in your application.

- The MQCSP fields are set to the required values.
- The address of the MQCSP structure is provided to the MQCNO structure.
- The MQCNO structure is set to MQCNO_VERSION_5.

MQCONN, MQCONNXX reason code examples for MQCC_FAILED

MQRC_HCONN_ERROR

(2018, X'7E2') Connection handle not valid

MQRC_MAX_CONNS_LIMIT_REACHED

(2025, X'7E9') Maximum number of connections reached

MQRC_NOT_AUTHORIZED

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

MQRC_Q_MGR_NAME_ERROR

(2058, X'80A') Queue manager name not valid or not known

MQRC_Q_MGR_NOT_AVAILABLE

(2059, X'80B') Queue manager not available for connection

MQRC_Q_MGR QUIESCING

(2161, X'871') Queue manager quiescing

MQRC_Q_MGR_STOPPING

(2162, X'872') Queue manager is shutting down

MQRC_RESOURCE_PROBLEM

(2102, X'836') Insufficient system resources available

MQCONNX-related environment variables

MQ_CONNECT_TYPE

- Works with the MQCONNX bindings option STANDARD
- If bindings option is other than STANDARD, variable is ignored

MQSAMP_USER_ID

- Used in IBM MQ server and client sample programs
- Passes a user ID to be used with the authentication capability
- Dependent on the AUTHINFO CHCKLOCL and CHCKCLNT settings
 - If an ID is provided when the CHCKLOCL or CHCKCLNT is set to OPTIONAL, a password must be provided
 - If no ID is used, no password is needed
 - If CHCKLOCL or CHCKCLNT has any of the required settings, then ID and password must be provided

Basic design and development concepts

© Copyright IBM Corporation 2017

Figure 2-29. MQCONNX-related environment variables

STANDARD bindings in the connection imply two threads, where the application and the queue manager run in two separate spaces. They are in contrast to a fast path binding, in which a “trusted application” runs in the same space as the queue manager, which is not generally advisable.

MQSAMPU_ID and connection authentication settings

AMQ8408: Display Queue Manager details.

```
QMNAME (NEWQM)
CONNAUTH (SYSTEM.DEFAULT.AUTHINFO.IDPWOS)
```

```
dis authinfo(SYSTEM.DEFAULT.AUTHINFO.I*) CHCKCLNT CHCKLOCL
4 : dis authinfo(SYSTEM.DEFAULT.AUTHINFO.I*) CHCKCLNT CHCKLOCL
```

AMQ8566: Display authentication information details.

```
AUTHINFO (SYSTEM.DEFAULT.AUTHINFO.IDPWOS)
AUTHTYPE (IDPWOS) CHCKCLNT (REQADM)
CHCKLOCL (OPTIONAL)
```

AMQ8566: Display authentication information details.

```
AUTHINFO (SYSTEM.DEFAULT.AUTHINFO.IDPWLDAP)
AUTHTYPE (IDPWLDAP) CHCKCLNT (REQUIRED)
CHCKLOCL (OPTIONAL)
```

- NONE
- OPTIONAL
- REQUIRED
- REQADM

```
C:>set MQSAMPU_ID=INGMUSR
C:>amqspput ORDERS.AT.MQ00 MQ00
Sample AMQSPUT0 start
Enter password: VALIDPWD
target queue is ORDERS.AT.MQ00
Check message that the application
now works
Sample AMQSPUT0 end
```

[Basic design and development concepts](#)

© Copyright IBM Corporation 2017

Figure 2-30. MQSAMPU_ID and connection authentication settings

Connection authentication can be coded in the application. The connection authentication behavior is configured in the queue manager:

- First, find the queue manager CONNAUTH attribute to determine which authentication information (AUTHINFO) object is in force for the queue manager.
- After you confirm the correct AUTHINFO record, display the CHCKLOCL and CHCKCLNT attributes. The attributes work in a similar way, one for local applications, the other for client applications correspondingly.
 - NONE, as implied, means that no authentication is done.
 - OPTIONAL means that a password is required only if an ID is provided. If no ID, no password is required.
 - REQUIRED means that an ID and password are always required.
 - REQADM means that if the ID is that of a user with administrative rights, then a password must be supplied.

A first look at MQOPEN

```

MQOD      od = {MQOD_DEFAULT};      /* Object descriptn*/
MQLONG   O_options;                /* MQOPEN options */
MQHOBJ   Hobj;                   /* object handle */
MQLONG   OpenCode;                /* MQOPEN comp code*/
MQLONG   Reason;                 /* reason code */

O_options = MQOO_OUTPUT /* open queue for output */
            | MQOO_FAIL_IF_QUIESCING
                           /* unless MQ is stopping */

MQOPEN(Hcon,           →/* connection handle */
       &od,          ←/* queue object descriptor*/
       O_options,    →/* open options */
       &Hobj,        ←/* object handle */
       &OpenCode,    ←/* MQOPEN completion code */
       &Reason);    ←/* reason code */

```

Basic design and development concepts

© Copyright IBM Corporation 2017

Figure 2-31. A first look at MQOPEN

A first look at MQPUT

```

MQOD    od = {MQOD_DEFAULT}; /* Object Descriptor */
MQMD    md = {MQMD_DEFAULT}; /* Message Descriptor */
MQPMO   pmo = {MQPMO_DEFAULT}; /* put message options*/
      . . . . .
MQBYTE buffer[65536];
MQLONG buflen;
MQLONG messlen;           /* message length */
MQPUT (Hcon,             /* connection handle */
       Hobj,             /* object handle */
       &md,              /* message descriptor */
       &pmo,              /* default options (datagram) */
       messlen,           /* message length */
       buffer,            /* message buffer */
       &CompCode,          /* completion code */
       &Reason);          /* reason code */

```

Basic design and development concepts

© Copyright IBM Corporation 2017

Figure 2-32. A first look at MQPUT

A first look at MQGET

```

MQOD    od = {MQOD_DEFAULT}; /* Object Descriptor */
MQMD    md = {MQMD_DEFAULT}; /* Message Descriptor */
MQGMO   gmo = {MQGMO_DEFAULT}; /* get message options*/
MQBYTE buffer[65536];        /* object handle */
MQLONG buflen;               /* reason code */
MQLONG messlen;              /* message length */

MQGET(Hcon,      /* connection handle */
      Hobj,      /* object handle */
      &md,       /* message descriptor */
      &gmo,       /* get message options */
      buflen,     /* buffer length */
      buffer,     /* message buffer */
      &messlen,   /* message length */
      &CompCode,  /* completion code */
      &Reason);  /* reason code */

```

Basic design and development concepts

© Copyright IBM Corporation 2017

Figure 2-33. A first look at MQGET

MQINQ and MQSET

- MQINQ is used to obtain object attribute information for either a queue manager, queue, namelist, or process definition objects



- MQSET can be used to change a small subset of queue attributes
- MQINQ and MQSET use selectors that map to the object attributes
- Six of the parameters that are used by MQINQ and MQSET are for selectors
 - Selectors are of two main types: integer and character
 - For the MQINQ call, the integer and character selectors are grouped according to the type of object they represent (queue manager, queues, a namelist, or process definition objects)

Basic design and development concepts

© Copyright IBM Corporation 2017

Figure 2-34. MQINQ and MQSET

The MQINQ and MQSET calls can be used for situations where the developer, with permissions, must interrogate and set IBM MQ object attributes. They are used with a number of attributes followed by an array that contains the specified attributes. You look at the syntax of MQINQ and MQSET in a later slide.

MQINQ and MQSET declarations

```

MQHCONN Hcon;           /* Connection handle      */
MQHOBJ Hinq;           /* Object handle        */
MQLONG Select[3];      /* Attrib selectors array */
MQLONG IAV[3];          /* Integer attributes array */
MQLONG CompCode;        /* Completion code       */
MQLONG Reason;          /* Completion code reason */

/* fields below for illustration only, omitted from sample */
MQLONG SelectorCount; /* Count of selectors   */
MQLONG IntAttrCount;   /* Integer attributes count */
MQLONG CharAttrLength; /* Length of character attributes buffer */
MQCHAR CharAttrs[n];   /* Character attributes */

```

[Basic design and development concepts](#)

© Copyright IBM Corporation 2017

Figure 2-35. MQINQ and MQSET declarations

You use the MQI Elementary data types to define the declarations that the MQINQ and MQSET function calls require.

MQINQ invocation

```

O_options = MQOO_INQUIRE <= Option used with the MQOPEN call
Select[0] = MQIA_INHIBIT_GET; /* attribute selectors */
Select[1] = MQIA_CURRENT_Q_DEPTH;
Select[2] = MQIA_OPEN_INPUT_COUNT;

    MQINQ(Hcon,      ➔ /* connection handle */ *
           Hinq,       ➔ /* object handle */ *
           3,          ➔ /* Selector count */ *
           Select,     ➔ /* Selector array */ *
           3,          ➔ /* integer attrib count */ /
Replies ===>IAV,    ➔ /* integer attrib array */ /
           0,          ➔ /* character attrib count */ /
           NULL,       ➔ /* character attrib array */ /
           &CompCode,   ➔ /* completion code */ *
           &Reason);  ➔ /* reason code */ /

```

Basic design and development concepts

© Copyright IBM Corporation 2017

Figure 2-36. MQINQ invocation

When you use the MQINQ function, you specify MQOO_INQUIRE in the open options for the object. Valid objects to query with MQINQ are the queue manager, queues, a namelist, or a process definition.

The parameters that follow the object handle are:

- The input, selector count is the total number of integer and character selectors.
- An input, selector array is the set of selector names that are specified in the call.
- An input, integer attribute count is the number of the attributes that are integer.
- An output, integer array contains the information that is returned for the integer selectors.
- The input, character attribute count is the number of the attributes that are character.
- An output, character array contains the information that is returned for the character selectors.

MQSET invocation

```

O_options = MQOO_SET    <= Option used with the MQOPEN call
Select[0] = MQIA_INHIBIT_PUT; /* attribute selector */
IAV[0]    = MQQA_PUT_INHIBITED; /* attribute value */

MQSET(Hcon,      →/* connection handle           */
      Hset,       →/* object handle             */
      1,          →/* Selector count           */
      Select,     →/* Selector array            */
      1,          →/* integer attribute count */
      IAV,        ←/* integer attribute array */
      0,          →/* character attribute count*/
      NULL,       ←/* character attribute array*/
      &CompCode,   ←/* completion code          */
      &Reason);   ←/* reason code              */

```

Basic design and development concepts

© Copyright IBM Corporation 2017

Figure 2-37. MQSET invocation

When you use the MQINQ function, you specify MQOO_SET in the open options for the object.

The parameters for the MQSET function call are similar to the parameters used for the MQINQ function.

However, MQSET is valid only for queue objects.

MQCLOSE and options

```

MQLONG    C_options;          /* MQCLOSE options      */
MQCLOSE(Hcon,           →   /* connection handle  */
&Hobj,    ←→   /* object handle       */
C_options, →   /* Closing options     */
&CompCode, ←   /* completion code    */
&Reason); ←   /* reason code        */

```

Default option:

MQCO_NONE

Dynamic queue options:

MQCO_DELETE

MQCO_DELETE_PURGE

Subscription options:

MQCO_KEEP_SUB

MQCO_REMOVE_SUB

Read-ahead options:

MQCO_IMMEDIATE

MQCO QUIESCE

Figure 2-38. MQCLOSE and options

The MQCLOSE call releases the resource that is obtained from the MQOPEN call. In most situations, the call does not require options. However, some options are for special situations, such as:

- If you need to delete a dynamic queue, you use the MQCO_DELETE, or use the MQCO_DELETE_PURGE to delete the queue and any messages in the queue. You use this option in a later exercise when you create a dynamic queue.
- When you work with publish/subscribe, you can use an option to keep or remove a subscription.
- If you work with an IBM MQ Client application, you might consider trying one of the read-ahead options to optimize processing of non-persistent messages.

MQCLOSE reason code examples for MQCC_FAILED

MQRC_CALL_IN_PROGRESS

(2219, X'8AB') MQI call entered before previous call complete

MQRC_CONNECTION_STOPPING

(2203, X'89B') Connection shutting down

MQRC_HCONN_ERROR

(2018, X'7E2') Connection handle not valid

MQRC_NOT_AUTHORIZED

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

MQRC_OBJECT_DAMAGED

(2101, X'835') Object damaged

MQRC_OPTION_NOT_VALID_FOR_TYPE

(2045, X'7FD') On an MQCLOSE call: option not valid for object type

MQRC_OPTIONS_ERROR

(2046, X'7FE') Options not valid or not consistent

MQRC_Q_NOT_EMPTY

(2055, X'807') Queue contains one or more messages or uncommitted put or get requests

MQRC_RESOURCE_PROBLEM

(2102, X'836') Insufficient system resources available

MQDISC and reason code examples for MQCC_FAILED

```
MQDISC(Hcon,      /* connection handle */
        &CompCode, /* completion code */
        &Reason); /* reason code */
```

MQRC_CALL_IN_PROGRESS

(2219, X'8AB') MQI call entered before previous call complete

MQRC_CONNECTION_STOPPING

(2203, X'89B') Connection shutting down

MQRC_HCONN_ERROR

(2018, X'7E2') Connection handle not valid

MQRC_Q_MGR_NOT_AVAILABLE

(2059, X'80B') Queue manager not available for connection

MQRC_Q_MGR_STOPPING

(2162, X'872') Queue manager shutting down

MQRC_RESOURCE_PROBLEM

(2102, X'836') Insufficient system resources available

MQRC_STORAGE_NOT_AVAILABLE

(2071, X'817') Insufficient storage available

MQI and object attribute override considerations

MQMD initial values (partial)

```
MQMD_DEFAULT
{MQMD_STRUC_ID_ARRAY}, \
MQMD_VERSION_1, \
MQRO_NONE, \
MQMT_DATAGRAM, \
... . . . . .
MQPER_PERSISTENCE_AS_Q_DEF, \
... . . . . .
```

QLOCAL attributes (partial)

```
QUEUE (EX2NONPERS)
TYPE (QLOCAL)
DEFBIND (OPEN)
DEFPSIST (NO)
DEFPRTY (0)
GET (ENABLED)
PUT (ENABLED)
... . . . . .
```

Disposition of messages upon restart of the queue manager

Persistence MQI code / queue attribute	DEFPSIST (NO)	DEFPSIST (YES)
MQPER_PERSISTENCE_AS_Q_DEF	Lost	Kept
MQPER_PERSISTENT	Kept	Kept
MQPER_NOT_PERSISTENT	Lost	Lost

Basic design and development concepts

© Copyright IBM Corporation 2017

Figure 2-41. MQI and object attribute override considerations

This display features the persistence attribute. However, the behavior that is shown here is similar for other attributes, such as binding options (DEFBIND), which enables an application to bind to a certain queue manager on the MQOPEN call. Most of the attributes that are set in the application override the attributes of the object definitions. As a developer, you have considerable control of what happens in the IBM MQ environment. With that control goes the responsibility to question requirements.

This slide shows the iterations that can occur with different queue definition attributes and persistence set in your code. You reproduce the behavior of this chart in the exercise that is associated with this unit.

DEFBIND(OPEN) means that the queue manager continues to use the same queue manager for that open handle. If you use IBM MQ clusters to distribute the load, this BIND attribute would interfere with the distribution of messages to other queue managers. A DEFBIND value on the MQOPEN of NOT_FIXED allows messages to be distributed among other different queue managers.

Manipulating character and binary strings with C language

- Character strings
 - Data strings returned to the application by the queue manager is padded with blanks to the defined field length
 - The queue manager does not null terminate strings
 - Use `strncpy`, `strcmp`, and `strncat` to copy, compare, or concatenate strings
 - Use `sizeof` to determine the length of a field
 - Do not use null-dependent functions such as `strcpy`, `strcmp`, `strcat`, or `strlen`
- Binary strings
 - Declare binary strings as one of the `MQBYTEn` elementary data types
 - To copy, compare, or set binary strings, use `memcpy`, `memcmp`, or `memset`, correspondingly
 - `strcpy`, `strcmp`, `strncpy`, or `strcmp` does not work well with `MQBYTEn` data types and should be avoided



Basic design and development concepts

© Copyright IBM Corporation 2017

Figure 2-42. Manipulating character and binary strings with C language

Examples of preparing C programs

Linux

(IBM MQ server application, simple format)

```
gcc -I/opt/mqm/inc -lmqm -o mqput mqput.c
```

(IBM MQ server application, 64-bit, non-threaded)

```
gcc -m64 -o amqspput_64 amqspput0.c -I MQ_INSTALLATION_PATH/inc -L
MQ_INSTALLATION_PATH/lib64 -Wl,-rpath=MQ_INSTALLATION_PATH/lib64
-Wl,-rpath=/usr/lib64 -lmqm
```

Windows

- Open a Visual Studio command prompt
- Include the C libraries
SET INCLUDE=C:\Program
Files\IBM\MQ\tools\c\include;%INCLUDE%
- Change to your source directory

```
cl mqput.c "c:\Program Files\IBM\MQ\tools\Lib\mqm.lib"
```

(With 64-bit libraries)

```
cl -MD amqspget0.c -Feamqspget.exe MQ_INSTALLATION_PATH\Tools\Lib64\mqm.lib
```

Basic design and development concepts

© Copyright IBM Corporation 2017

Figure 2-43. Examples of preparing C programs

In this course, you use a simple compile. The display for this slide shows an extra compile option.

IBM Knowledge Center shows how to compile your code for different situations. However, some of the flags that are used depend on the compiler and operating system that you use.

Unit summary (1 of 2)

- Describe common messaging patterns
- Explain key architecture and performance considerations for message and application design
- List the available programming options
- Describe the calls, structures, and elementary data types that compose the message queue interface
- Describe message types and message formats
- Explain how to use the MQCONN or MQCONNX calls, and the various options of the MQCONNX call, to connect to a queue manager
- Describe how the MQOPEN, MQPUT, and MQGET calls use the output of the MQCONN or MQCONNX calls

Basic design and development concepts

© Copyright IBM Corporation 2017

Figure 2-44. Unit summary (1 of 2)

Unit summary (2 of 2)

- Explain how to use the MQCNO and MQCSP structures with the MQCONN function call to implement connection authentication
- Describe the use of the MQINQ and MQSET calls and the differences between them
- Distinguish the superseding characteristics between object and MQI attributes
- Describe how to compile a C program in the Linux and Windows environments

Basic design and development concepts

© Copyright IBM Corporation 2017

Figure 2-45. Unit summary (2 of 2)

Review questions (1 of 2)

1. Which of the factors that are listed can have an impact on performance?
 - a. Large messages
 - b. Searching a queue for a specific message
 - c. Persistent messages
 - d. All of the above
2. Which of the items that are listed are not part of the MQI?
 - a. Object definition files
 - b. Elementary data items
 - c. Named constants
 - d. Data definition files
3. True or False: The header file that is most commonly used when coding C programs is `cmqpsc.h`.



Basic design and development concepts

© Copyright IBM Corporation 2017

Figure 2-46. Review questions (1 of 2)

Write your answers here:

- 1.
- 2.
- 3.

Review questions (2 of 2)

4. What named constant do you use to initialize a connection security parameter structure?
 - a. MQCNO_VERSION_5
 - b. MQCSP_DEFAULT
 - c. MQCNO_DEFAULT
 - d. MQCNO_STRUC_ID_ARRAY

5. Select all correct answers. Which of the IBM MQ objects that are listed can have its attributes changed with the MQSET call?
 - a. Queue manager
 - b. Queue
 - c. Namelist
 - d. Process



Basic design and development concepts

© Copyright IBM Corporation 2017

Figure 2-47. Review questions (2 of 2)

Write your answers here:

4.

5.

Review answers (1 of 3)



1. Which of the factors that are listed can have an impact on performance?
 - a. Large messages
 - b. Searching a queue for a specific message
 - c. Persistent messages
 - d. All of the above

The answer is: d, All of the above. Large messages, searching for a message, and use of persistent messages all affect performance.

2. Which of the items that are listed are not part of the MQI?
 - a. Object definition files
 - b. Elementary data types
 - c. Named constants
 - d. Data definition files

The answer is: a, Object definition files are used by administrators to define IBM MQ objects such as queues and channels.

Review answers (2 of 3)

3. True or False: The header file that is most commonly used when coding C programs is cmqpsc.h.

The answer is: False. The most commonly used C header file is cmqc.h. The cmqpsc.h file is used for queued publish/subscribe.

4. What named constant do you use to initialize a connection security parameter structure?

- a. MQCNO_VERSION_5
- b. MQCSP_DEFAULT
- c. MQCNO_DEFAULT
- d. MQCNO_STRUC_ID_ARRAY

The answer is: a, Use MQCSP_DEFAULT to initialize a connection security parameter, or “CSP” structure.



Review answers (3 of 3)

5. Select all correct answers. Which of the IBM MQ objects that are listed can have its attributes changed with the MQSET call?

- a. Queue manager
- b. Queue
- c. Namelist
- d. Process

The answer is: b, The MQSET call can change a selected set of queue attributes. MQSET cannot change the attributes of any other object types.



Exercise: Getting started with IBM MQ development

Basic design and development concepts

© Copyright IBM Corporation 2017

Figure 2-51. Exercise: Getting started with IBM MQ development

Exercise objectives

- Compile and test a copy of the put message sample program
- Review the cmqc.h structure and the initialization values for the message descriptor structure
- Review selected default values of a local queue definition
- Add MQOPEN, MQPUT, and MQCLOSE calls to an application
- Change persistence attributes in a program by using named constants
- Determine the outcome of persistence behavior when the queue definition attributes and the MQL attributes use different values
- Check the queue manager environment to determine the connection authentication settings
- Set a variable to test connection authentication with a program
- Determine the results of not setting the correct version number in a structure



Basic design and development concepts

© Copyright IBM Corporation 2017

Figure 2-52. Exercise objectives

Unit 3. MQOPEN, queue name resolution, and MQPUT

Estimated time

01:00

Overview

This unit provides a detailed look at the MQOPEN and MQPUT calls. You learn how MQOPEN facilitates queue name resolution and the creation of dynamic queues. You also learn about the fields in the message descriptor structure, and how to use these fields in your application.

How you will check your progress

Accountability:

- Review questions
- Lab Exercises

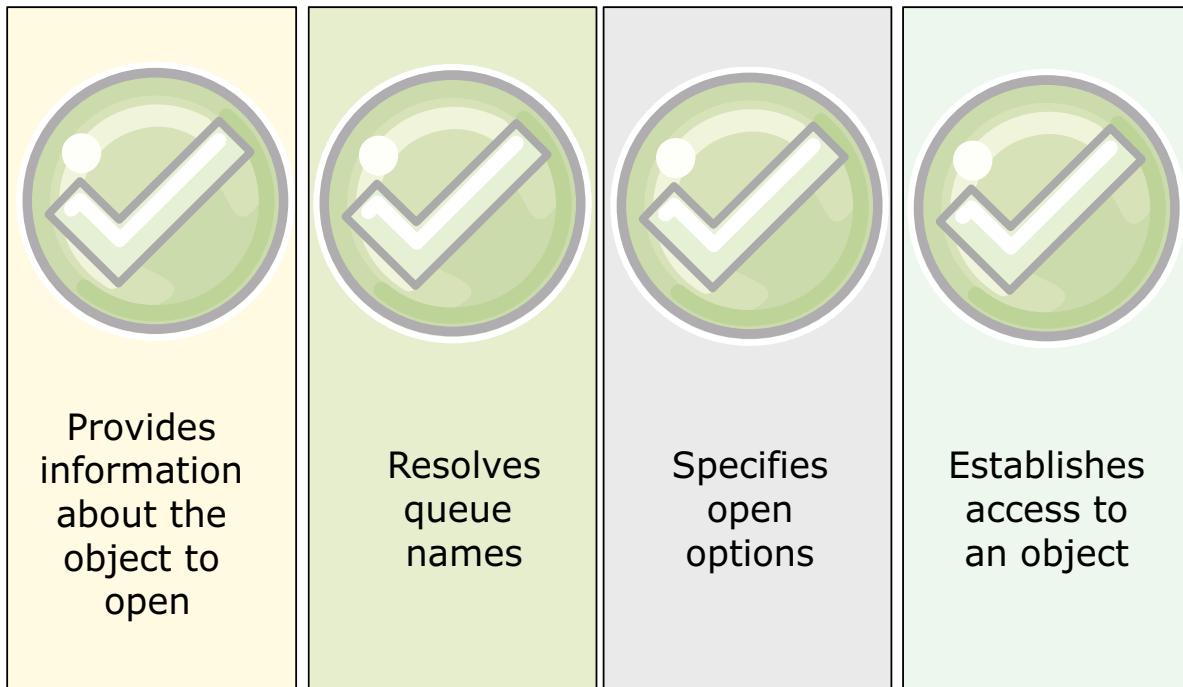
References

IBM Knowledge Center for IBM MQ V9 documentation

Unit objectives

- Describe the details that the MQOPEN call handles
- Identify the information in the object descriptor (MQOD) structure
- Describe the options that can be specified in the MQOPEN call
- Describe how the MQOPEN call processes queue name resolution
- Explain the use of fields in the message descriptor (MQMD) structure
- Describe how the IBM MQ V8.0.0.4 expiry cap overrides higher expiry specifications in the application
- Describe various uses of Report messages
- Describe the two types of context information and how context can be used to identify the user of an application
- Examine use of the MQPUT1 call and identify optimal scenarios for its use
- Explain how to create and remove temporary or permanent dynamic queues

The MQOPEN call



MQOPEN, queue name resolution, and MQPUT

© Copyright IBM Corporation 2017

Figure 3-2. The MQOPEN call

In this unit, you take a closer look at the MQOPEN option. MQOPEN has many roles, such as:

- Establishing information about the object to be open.
- Initiating queue name resolution.
- Specifying how the object, which might be a queue or a topic, should be open. Is it for getting messages (input), or for putting messages (output)?
- Being used to access the object.

MQOPEN

```

MQOD      od = {MQOD_DEFAULT};      /* Object descriptor*/
MQLONG   O_options;                /* MQOPEN options */
MQHOBJ   Hobj;                   object handle */
MQLONG   OpenCode;                MQOPEN comp code */
MQLONG   Reason;                 /* reason code */

O_options = MQOO_OUTPUT          /* open q for output*/
| MQOO_FAIL_IF_QUIESCING;

MQOPEN(Hcon,           →/* connection handle */
&od,                  ←→/* queue object descriptor*/
O_options,             →/* open options */
&Hobj,                ←/* object handle */
&OpenCode,              ←/* MQOPEN completion code */
&Reason);              ←/* reason code */

```

Setting multiple options

MQOPEN, queue name resolution, and MQPUT

© Copyright IBM Corporation 2017

Figure 3-3. MQOPEN

Before doing anything with any object such as MQINQ, MQSET, MQGET, and MQPUT, the object must be opened for the intended functions.

All processing options are established at open time. The only time that an MQOPEN is not required is when using the MQPUT1 call to place a single message on a queue.

As with all MQI calls, a series of parameters is passed between the application and the queue manager. The completion code and reason code can be checked to determine the status of the call when it completes.

MQOD: Information about the object to open (1 of 2)

```

MQCHAR4  StrucId;          /* Structure identifier */
MQLONG   Version;          /*Structure version number*/
MQLONG   ObjectType;       /* Object type */
MQCHAR48 ObjectName;       /* Object name */
MQCHAR48 ObjectQMgrName;  /* Object queue mgr name */
MQCHAR48 DynamicQName;    /* Dynamic queue name */
MQCHAR12 AlternateUserId; /* Alternate user Idntfr*/
/* Ver:1 */

MQLONG   RecsPresent;      /* Num of object recs pres*/
MQLONG   KnownDestCount;   /* Num local q opened OK */
MQLONG   UnknownDestCount; /* Num of remote q open*/
MQLONG   InvalidDestCount; /* Number q failed open*/
MQLONG   ObjectRecOffset;  /*Offset first object rec*/
MQLONG   ResponseRecOffset; /*Offset first resp rec*/

```

MQOPEN, queue name resolution, and MQPUT

© Copyright IBM Corporation 2017

Figure 3-4. MQOD: Information about the object to open (1 of 2)

The object descriptor (MQOD) is the structure that an application uses to identify the type of object (most often a queue) being opened and its name. The queue name is never used on subsequent MQGETs and MQPUTs, just the object handle.

As with many structures that are defined to IBM MQ, the object descriptor begins with a 4-byte constant. The structure identifier (MQOD_StrucId) contains the constant value “OD”. Other structures are examined later in the unit.

Some of the fields in the object descriptor are set by the application, and some are updated and returned to the application as a result of the MQOPEN call. Each field in the object descriptor has a specific length and purpose. The version 1 section of the structure has these fields:

- StrucId is the identifier.
- Version allows for potential different MQOD layouts. For this structure, the version can be 1, 2, 3, or 4. The default is 1. Other structures might have different version numbers.
- ObjectType specifies the IBM MQ object that is opened.
- ObjectName contains the actual name of the object that is opened.
- ObjectQMgrName can contain the name of the queue manager where the object is defined.

- DynamicQName is a special value that is used when dynamically creating a queue. Dynamic and model queues are covered later.
- AlternateUserId is used when the queue is being opened on behalf of a request from another program or user.

The MQOD initialization structure provides the values as shown:

```
#define MQOD_DEFAULT {MQOD_STRUC_ID_ARRAY}, \
    MQOD_VERSION_1, \
    MQOT_Q, \
    {" "}, \
    {" "}, \
    {"AMQ.*"}, \
    {" "}, \
```

MQOD: Information about the object to open (2 of 2)

```

MQPTR ObjectRecPtr; /* Address first object rec*/
MQPTR ResponseRecPtr; /* Address first resp rec */
/* Ver:2 */

MQBYTE40 AlternateSecurityId; /* Alt sec identifier */
MQCHAR48 ResolvedQName; /* Resolved queue name */
MQCHAR48 ResolvedQMgrName; /* Resolved queue mgr nm */
/* Ver:3 */

MQCHARV ObjectString; /* Object long name */
MQCHARV SelectionString; /* Message Selector */
MQCHARV ResObjectString; /* Resolved long obj name*/
MQLONG ResolvedType; /* Alias queue resolved
object type*/
/* Ver:4 */

```

MQOPEN, queue name resolution, and MQPUT

© Copyright IBM Corporation 2017

Figure 3-5. MQOD: Information about the object to open (2 of 2)

MQOPEN options that are used with MQPUT

Options for putting messages:

MQOO_OUTPUT

Options for cluster queue:

MQOO_BIND_NOT_FIXED

MQOO_BIND_ON_OPEN

MQOO_BIND_ON_GROUP

Options for queue manager quiescing

MQOO_FAIL_IF QUIESCING

Options for resolving to local queue names:

MQOO_RESOLVE_LOCAL_Q

Options that are related to message context

MQOO_PASS_IDENTITY_CONTEXT

MQOO_PASS_ALL_CONTEXT

MQOO_SET_IDENTITY_CONTEXT

MQOO_SET_ALL_CONTEXT

Options to run use a different user authority

MQOO_ALTERNATE_USER_Authority

MQOPEN, queue name resolution, and MQPUT

© Copyright IBM Corporation 2017

Figure 3-6. MQOPEN options that are used with MQPUT

Open options are the means by which the application declares its intent on the object that is opened. The open options that are listed are some of the most common.

It is valid to combine (add) multiple options when they do not conflict. For instance, a common pair of options is MQOO_INPUT_SHARED and MQOO_BROWSE. Generally, a browse is done to determine whether any messages meet some processing criteria. If so, the application wants to take the message off the queue for processing. However, combining MQOO_INPUT_EXCLUSIVE and MQOO_INPUT_SHARED is not valid.

The program can defer to the queue definition for the input option to use.

MQOPEN reason codes: Partial (1 of 2)

- **MQRC_ALIAS_BASE_Q_TYPE_ERROR** (2001, X'7D1') Alias base queue not a valid type
- **MQRC_CALL_IN_PROGRESS** (2219, X'8AB') MQI call entered before previous call complete
- **MQRC_CF_NOT_AVAILABLE** (2345, X'929') Coupling facility not available
- **MQRC_CLUSTER_PUT_INHIBITED** (2268, X'8DC') Put calls inhibited for all queues in cluster
- **MQRC_CONNECTION_BROKEN** (2009, X'7D9') Connection to queue manager lost
- **MQRC_CONNECTION_NOTAUTHORIZED** (2217, X'8A9') Not authorized for connection
- **MQRC_DEF_XMIT_Q_TYPE_ERROR** (2198, X'896') Default transmission queue not local
- **MQRC_DEF_XMIT_Q_USAGE_ERROR** (2199, X'897') Default transmission queue usage error
- **MQRC_DYNAMIC_Q_NAME_ERROR** (2011, X'7DB') Name of dynamic queue not valid

Figure 3-7. MQOPEN reason codes: Partial (1 of 2)

An MQOPEN might result in various return codes. The Application Programming Reference manual contains a description of each, along with suggested actions.

You look at other reason codes in the next slide.

MQOPEN reason codes: Partial (2 of 2)

- **MQRC_HANDLE_NOT_AVAILABLE** (2017, X'7E1') No more handles available
- **MQRC_HCONN_ERROR** (2018, X'7E2') Connection handle not valid
- **MQRC_HOBJ_ERROR** (2019, X'7E3') Object handle not valid
- **MQRC_MULTIPLE_REASONS** (2136, X'858') Multiple reason codes returned
- **MQRC_NAME_IN_USE** (2201, X'899') Name in use
- **MQRC_NOTAUTHORIZED** (2035, X'7F3') Not authorized for access
- **MQRC_OBJECT_NAME_ERROR** (2152, X'868') Object name not valid
- **MQRC_OBJECT_NOT_UNIQUE** (2343, X'927') Object not unique
- **MQRC_Q_MGR QUIESCING** (2161, X'871') Queue manager quiescing
- **MQRC_RECS_PRESENT_ERROR** (2154, X'86A') Number of records present not valid
- **MQRC_RESOURCE_PROBLEM** (2102, X'836') Insufficient system resources available
- **MQRC_STORAGE_NOT_AVAILABLE** (2071, X'817') Insufficient storage available
- **MQRC_UNKNOWN_OBJECT_NAME** (2085, X'825') Unknown object name

[MQOPEN, queue name resolution, and MQPUT](#)

© Copyright IBM Corporation 2017

Figure 3-8. MQOPEN reason codes: Partial (2 of 2)

You can use *The Application Programming Reference* manual for guidance on how to proceed for each situation.

Look at the MQRC_RESOURCE_PROBLEM return code 2102. This situation might be the result of a “misbehaved: application.”

Queue name resolution

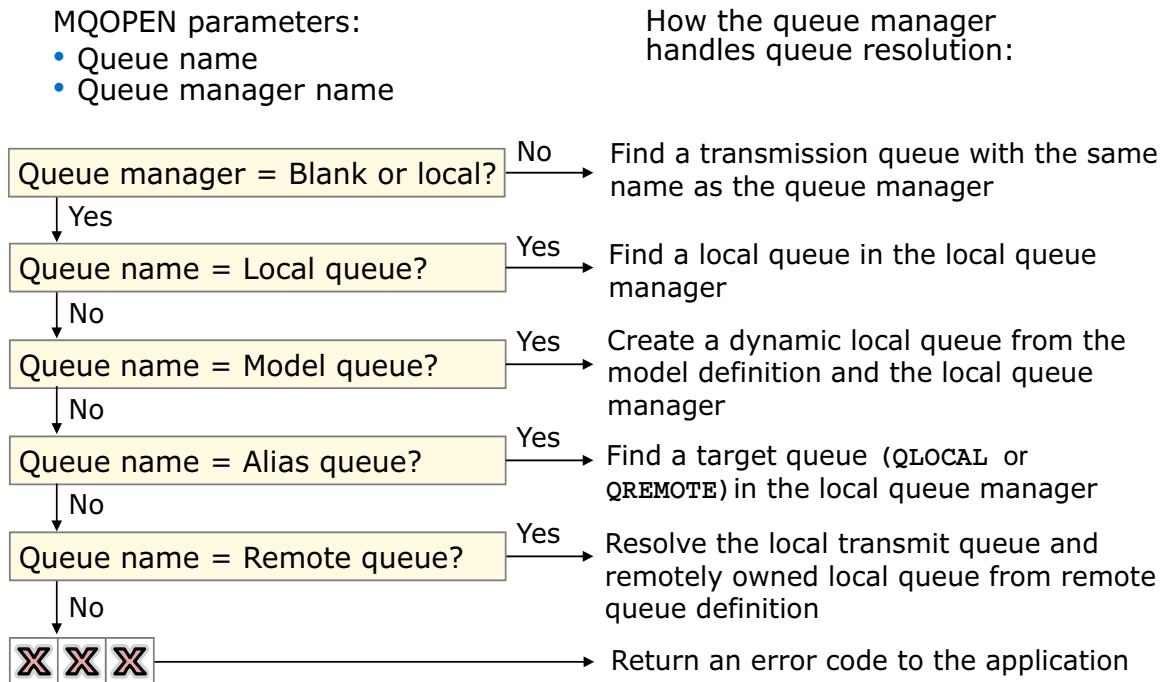
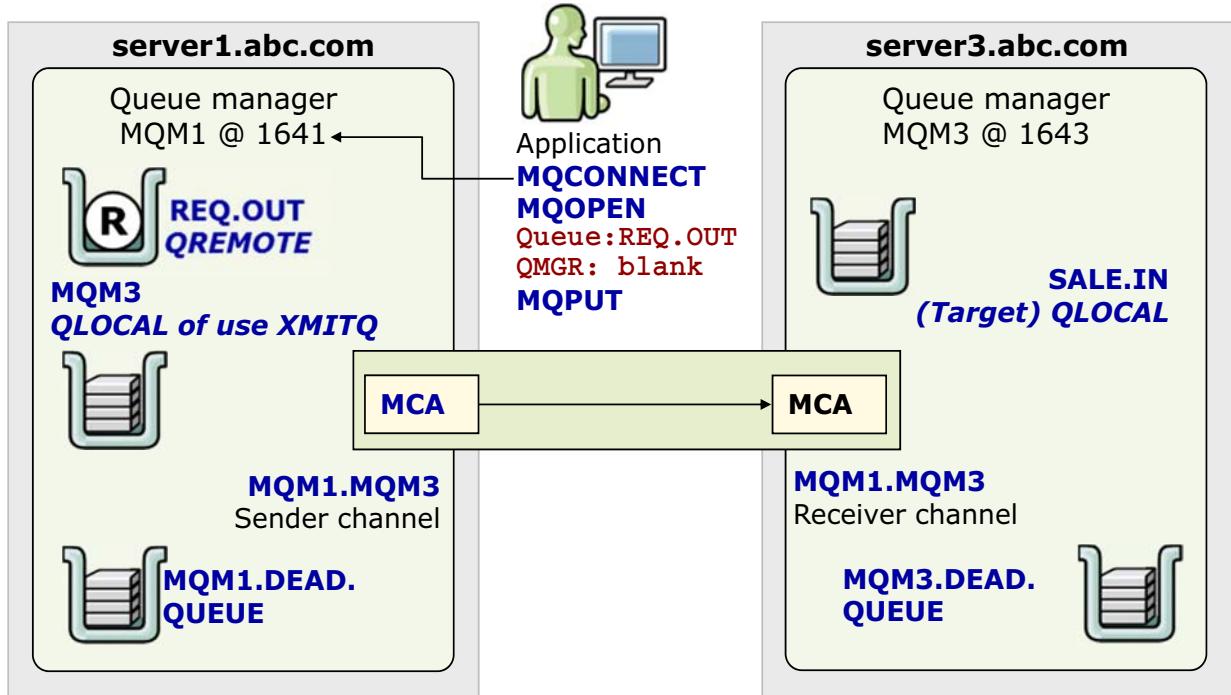


Figure 3-9. Queue name resolution

Different factors determine how the queue manager handles queue name resolution. The diagram summarizes some of the key concepts.

- The queue manager name that is passed in the MQOPEN call is checked.
- It can happen that the name of this queue manager is not omitted, and does not match the name of the local queue manager. In this case, queue name resolution looks for a locally defined transmission queue with a name that matches the name of the queue manager that is used in the MQOPEN call.
- If the queue manager name was blank, queue resolution checks whether a local queue that matches the name of the queue that is used in the MQOPEN exists. If such a local queue is found, queue resolution ends up placing the message in the local queue.
- The name of the queue that is used in the MQOPEN might not match the name of a model queue, or the name of an alias queue, in that order. In this case, queue name resolution looks for the name of a remote queue that matches the name that is used in the MQOPEN.
- If no match, an error is returned to the application, usually a return 2085, unknown object name.

Sender-receiver channel pair with remote queue



MQOPEN, queue name resolution, and MQPUT

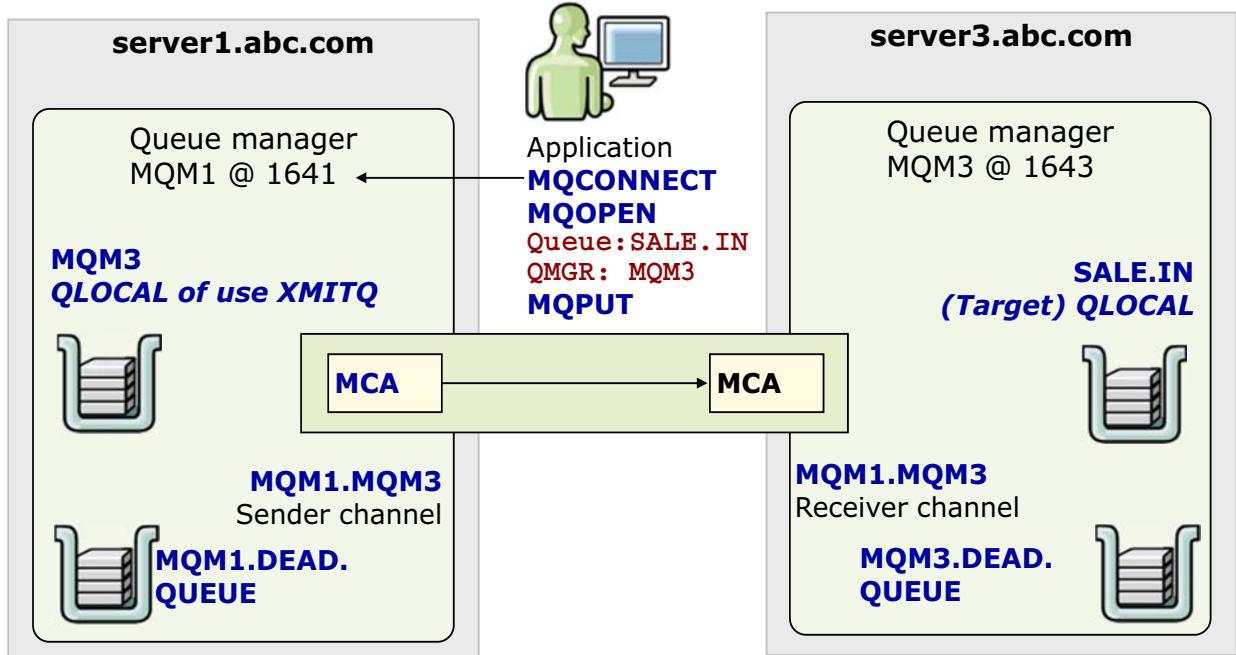
© Copyright IBM Corporation 2017

Figure 3-10. Sender-receiver channel pair with remote queue

This diagram shows the situation where the queue manager is left blank, and the put is done to a remote queue. It matches the following situation from the queue name resolution slide.

The name of the queue that is used in the MQOPEN might not match the name of a model queue, or the name of an alias queue, in that order. In this case, queue name resolution looks for the name of a remote queue that matches the name that is used in the MQOPEN.

Sender-receiver channel pair without remote queue



MQOPEN, queue name resolution, and MQPUT

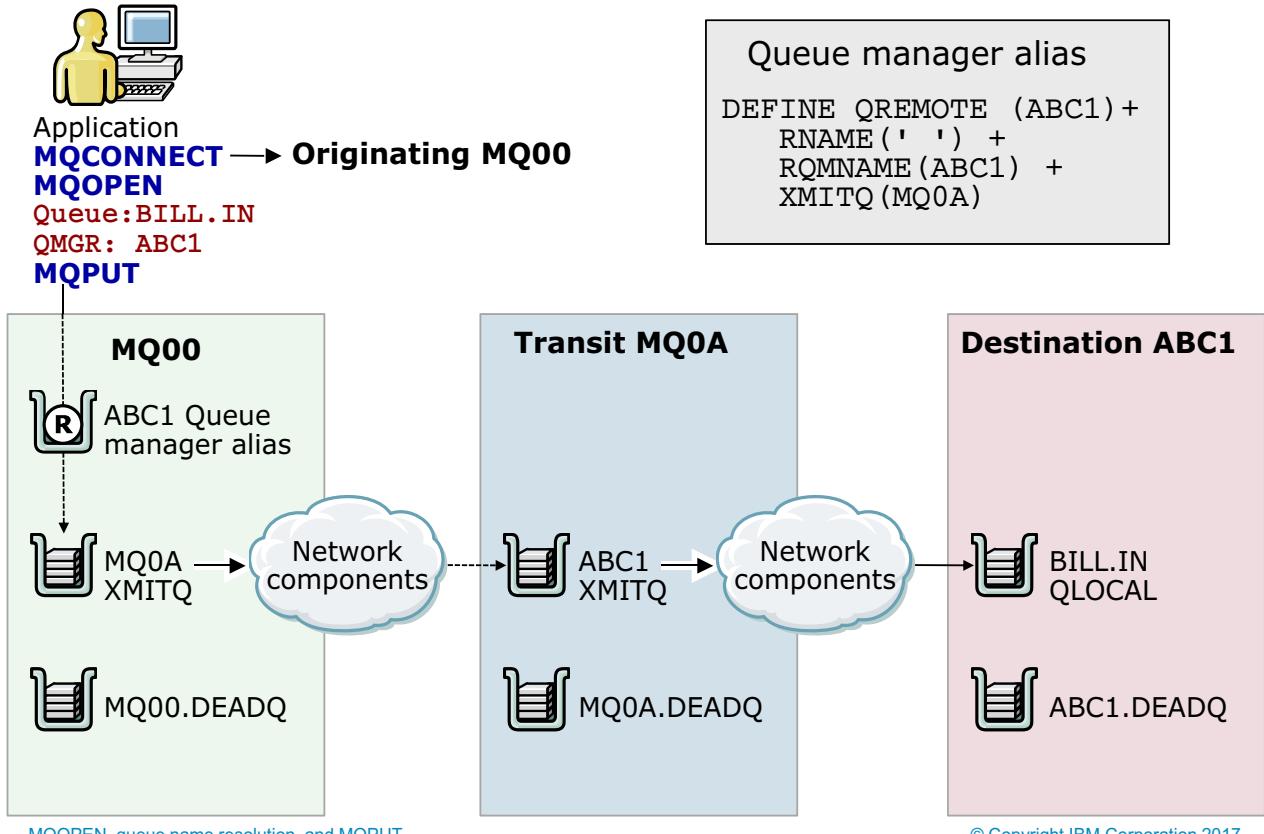
© Copyright IBM Corporation 2017

Figure 3-11. Sender-receiver channel pair without remote queue

The diagram on this slide matches the following situation from the queue name resolution slide.

It can happen that the name of this queue manager is not omitted, and does not match the name of the local queue manager. In this case, queue name resolution looks for a locally defined transmission queue with a name that matches the name of the queue manager that is used in the MQOPEN call.

Multi-hopping



In some IBM MQ infrastructures, a message might go across over other queue managers on the way to its target destination. Queue name resolution takes place in each queue manager. You see this diagram again.

MQPUT-related structures and options

```

MQOD    od = {MQOD_DEFAULT}; /* Object Descriptor */
MQMD    md = {MQMD_DEFAULT}; /* Message Descriptor */
MQPMO   pmo = {MQPMO_DEFAULT}; /* put message options*/
.....
MQBYTE buffer[65536];
MQLONG buflen;
MQLONG messlen;           /* message length */
MQPUT(Hcon,               /* connection handle */
      Hobj,               /* object handle */
      &md,                /* message descriptor */
      &pmo,                /* default options (datagram) */
      messlen,             /* message length */
      buffer,              /* message buffer */
      &CompCode,            /* completion code */
      &Reason);            /* reason code */

```

MQOPEN, queue name resolution, and MQPUT

© Copyright IBM Corporation 2017

Figure 3-13. MQPUT-related structures and options

The MQPUT call is used to place messages to a queue that is open for output. As you see in a later unit, MQPUT is also used to publish messages to a topic.

The term *input field*, which is shown as arrow that points to the right, means a parameter that is *provided to* the function call. *Output field*, which is shown as an arrow that points to the left, is a parameter that is *returned by* the call. As shown by the arrows, some of the parameters are for input, some for output, and some are both input and output. The last category becomes important when you use the same call to send several messages, as it is important to reset these fields before sending a new message.

This slide takes a first look at these parameters, and the rest of this presentation takes a closer look at the message descriptor and put message option parameters. You also look at selected reason codes that the MQPUT call might return.

The parameters that are passed in this call are:

- Hcon is an input field, which represents the connection handle from the MQCONN or MQCONNX call
- Hobj is an input field, which represents the object handle from the MQOPEN call.
- md is both an input and output field and represents the MQMD structure declared for this call.

- `pmo` is both an input and output field that represents the put message options that are specified for this MQPUT call.
- `messlen` is an input field that represents the length of the message.
- `buffer` is an input field that holds the message data, or payload.
- `CompCode` is an output field that holds the completion code.
- `Reason` is an output field that holds the reason code.



Important

When your code uses a loop to process many gets or puts, it is critical to reset fields that are used for both input and output before each function call is made.

MQMD (1 of 2)

```

StrucId : 'MD' Version : 2
Report : 0 MsgType : 8
Expiry : -1 Feedback : 0
Encoding: 546 CodedCharSetId : 1208
Format : 'MQSTR'
Priority: 0 Persistence : 1
MsgId :
X'414D51204D513031202020202020202020629AC356059B0220'
CorrelId :
X'00000000000000000000000000000000'
BackoutCount : 0 ←
ReplyToQ : ' '
ReplyToQMgr : 'MQ01'

```

- Output field for MQGET
- Used with MQGET UOW processing
- Ignored in MQPUT and MQPUT1

[MQOPEN, queue name resolution, and MQPUT](#)

© Copyright IBM Corporation 2017

Figure 3-14. MQMD (1 of 2)

The MQMD fields display was obtained with IBM MQ sample program `amqsbcg`. This sample application might come in handy when:

- You need to look at the MQMD fields, and
- You need to get your message contents in hexadecimal format for debugging purposes

The message that is processed by `amqsbcg` is of limited length, so you might need to recompile the program when your data is longer than the sample can handle.

The MQMD fields are detailed in the rest of this unit.

Each message has a message descriptor. An application or queue manager can update the MQMD.

- The MQMD starts with a structure identifier (StrucId). The value is “MD”.
- The next parameter is the version number, which can be 1 or 2. If the version is 2, the queue manager does extra checks for other structures.
- Message type is used to help IBM MQ and applications know what a particular message is used for.
- Encoding contains the representation of numeric data in the message. You learn more about encoding and data conversion in a later unit.

- CodedCharSetId contains the International Standards Organization code page number that is associated with the message. You learn more about encoding and data conversion in a later unit.
- Format is used to help the queue manager know what to do with data that requires conversion. You learn more about encoding and data conversion in a later unit.
- Priority can be used to control the sequence in which messages are received from a queue.
- Persistence determines whether the message is logged to enable its recovery.
- MsgId and CorrelId can be used to retrieve specific messages from a queue.
- BackoutCount tracks of the number of times a transaction is backed out when you work with units of work.
- ReplyToQ and ReplyToQmgr hold the name of the queue and queue manager that a reply, when working with reply messages, or report, when requesting a report message, should be sent to.

The MQMD field description continues in the next slide.



MQMD (2 of 2)

MQOPEN, queue name resolution, and MQPUT

© Copyright IBM Corporation 2017

Figure 3-15. MQMD (2 of 2)

The identity context fields allow an application that retrieves a message to:

- Check the authority of the sending application.
 - Process accounting functions, such as charge back.

Programmers might deal with IBM MQ security in two ways: the context fields of the message descriptor, and the alternate user authority.

Under normal circumstances, the queue manager checks the authority from the UserIdentifier in the identity context section. An alternate user might need to be checked instead. To use alternate user authority, two things must happen:

- The IBM MQ administrator needs to configure the correct authority to use alternate user.
 - The MQOPEN call needs to use the MQOO_ALTERNATE_USER_AUTHORITY option.

The origin context provides useful information, such as what type of application put the message. In the display, you see a numeric value; you can check the `cmqc.h` header file. An example of

application types from the `cmqc.h` file is shown. Note the last entry, `MQAT_AMQP`. You work with AMQP applications later in this course.

```
/* Put Application Types */
#define MQAT_CICS          1
#define MQAT_ZOS           2
#define MQAT_UNIX          6
#define MQAT_QMGR          7
#define MQAT_OS400         8
#define MQAT_WINDOWS        9
#define MQAT_IMS_BRIDGE    19
#define MQAT_AMQP          37
```

Next you have the `PutApplName`, which as the name implies, identifies the application that put the message.

`PutDate` and `PutTime` fields are given in `GMT`, and they depend on the accuracy of the originating system date and time setting. These fields can be useful for troubleshooting or auditing purposes.

`ApplOriginData` might provide extra information on the putting application suite.

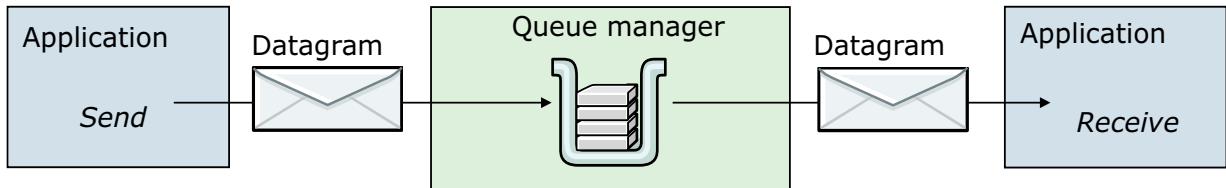
The fields with names that start with `GroupId` and below are part of a later unit in this course. These fields are ignored when the `MQMD` version number is less than 2.

The first few settings of the initialization structure for the `MQMD` are shown:

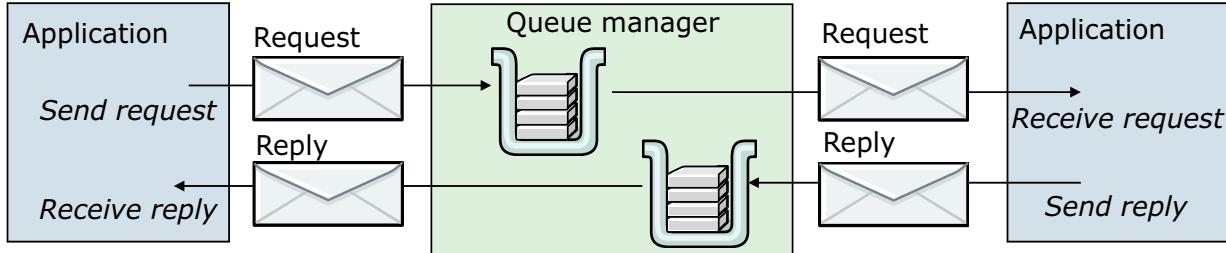
```
#define MQMD_DEFAULT {MQMD_STRUC_ID_ARRAY}, \
                      MQMD_VERSION_1, \
                      MQRO_NONE, \
                      MQMT_DATAGRAM, \
                      MQEI_UNLIMITED, \
                      MQFB_NONE, \
                      MQENC_NATIVE, \
                      MQCCSI_Q_MGR, \
                      ....
```

Message types: ReplyToQ and ReplyToQMgr

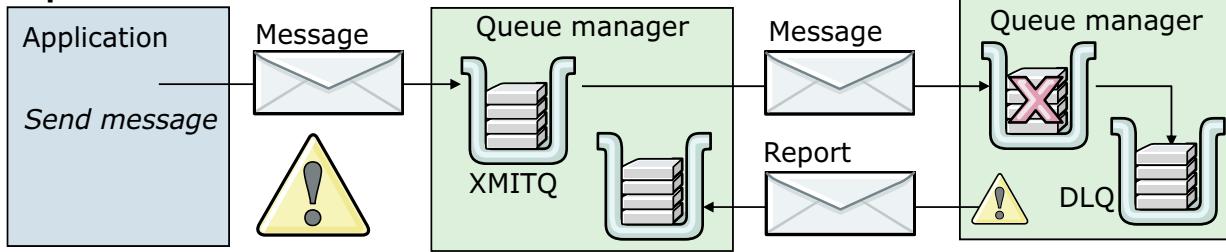
Datagram



Request/Reply



Report



MQOPEN, queue name resolution, and MQPUT

© Copyright IBM Corporation 2017

Figure 3-16. Message types: ReplyToQ and ReplyToQMgr

Message types are used to help IBM MQ and applications know what a particular message is used for.

A view at the message types from `cmqc.h` is shown.

IBM reserves use of message types 1 – 65535. Applications can use their own message types if they start with 65536 and above.

```

/* Message Types */
#define MQMT_SYSTEM_FIRST          1
#define MQMT_REQUEST                1
#define MQMT_REPLY                  2
#define MQMT_DATAGRAM               8
#define MQMT_REPORT                 4
#define MQMT_MQE_FIELDS_FROM_MQE   112
#define MQMT_MQE_FIELDS             113
#define MQMT_SYSTEM_LAST            65535
#define MQMT_APPL_FIRST             65536
#define MQMT_APPL_LAST              99999999
  
```

Report messages

- Messages that inform an application about events that pertain to the original message
 - Requested by setting the MQMD option when writing an application that puts messages to a queue
 - Delivered to the ReplyToQueue named in the MQMD
- Examples of report messages are:
 - Confirmation of arrival
 - Confirmation of delivery
 - Expiry report
- Consuming report messages should be part of the application design
- Report messages can be configured to contain:
 - The entire original message
 - The first 100 bytes of the original message
 - Just the report and MQMD but none of the original message



MQOPEN, queue name resolution, and MQPUT

© Copyright IBM Corporation 2017

Figure 3-17. Report messages

A report contains information about other messages and can be used for many purposes, such as:

- Auditing: Did a certain message get delivered to a certain location?
- System monitoring: Are there problems with the channels that process some messages?
- Correlating decoupled requests with replies
- Determining whether messages are expiring unexpectedly

Expiry

- Expiry is a message property set by a IBM MQ application that limits the time that a message can remain in a queue before it is processed
- When the expiry time set by the application expires, the message is discarded as follows:
 - The next time an MQGET is attempted on the expired message
 - (z/OS only) Explicitly by issuing MQSC command to expire the messages on the queue
 - (z/OS only) Periodically by configuring the queue manager to run expiry scans at set intervals
- Setting expiry is a good option for time-sensitive applications where messages become invalid if they remain in the queue longer than expected
- A special report message can be generated when a message expires



IBM MQ V8.0.0.4 and later: Queue definition expiry value can override a higher `MQMD` expiry value if using `CUSTOM CAPEXPRY`

MQOPEN, queue name resolution, and MQPUT

© Copyright IBM Corporation 2017

Figure 3-18. Expiry

Expiry is a period in tenths of a second that the application determines. After the expiry period, a message can be discarded when the next MQGET on that message is attempted. Expiry is a good option when messages are time sensitive, such as a stock quotation, and are considered invalid after a predetermined amount of time elapses.

When a message expires, a report message can be requested to track the expiration. Removal of expired messages is different in distributed than in z/OS. In z/OS, commands are available to remove expired messages.

The IBM MQ administrator can set a limit on the expiry value that can be set. You look at this new feature in the next slide.

MQMD expiry and the CAPEXPRY queue or topic attribute

- If the `MQMD.Expiry` is greater than the CUSTOM queue `CAPEXPRY` attribute, the `CAPEXPRY` value overrides the `MQMD.Expiry` value
- If the `MQMD.Expiry` value is less than the CUSTOM queue `CAPEXPRY` attribute, `MQMD.Expiry` value is used
- If more than one queue in the resolution path, such as an alias queue and its target queue, the `CAPEXPRY` with the lowest value is used

MQMD Expiry	CAPEXPRY attribute	Resulting expiry
md.Expiry = 3000	1000	1000
md.Expiry = 500	1000	500
md.Expiry = 900	Alias 1000 Target queue 3000	900
md.Expiry = 2000	Alias 1000 Target queue 3000	1000

MQOPEN, queue name resolution, and MQPUT

© Copyright IBM Corporation 2017

Figure 3-19. MQMD expiry and the CAPEXPRY queue or topic attribute

When `CAPEXPRY` is set, the *lowest* of the two expiry values between `CAPEXPRY` and the `MQMD.Expiry` is used. If the application has a higher expiry, the expiry set by the queue manager is taken, as shown in the display.

The queue name resolution might involve more than one queue, such as the case of an alias queue, and its target local queue. When each queue has a `CAPEXPRY` value that is specified, the lowest of the `CAPEXPRY` value is used. This lowest `CAPEXPRY` value is compared to the `MQMD.Expiry` field, and the lowest expiry is used.

Feedback

- Present for report message type MQMT_REPORT
- Feedback field can contain:
 - A feedback code prefixed with MQFB_
 - An MQRC reason code

Sample feedback codes

(partial list):

MQFB_NONE

MQFB_EXPIRATION

MQFB_COA

MQFB_COD

MQFB_PAN

MQFB_NAN

MQFB_QUIT

MQFB_SYSTEM_FIRST

Sample reason codes

(partial list):

MQRC_Q_FULL

MQRC_NOT_AUTHORIZED

MQRC_PUT_INHIBITED

MQRC_Q_SPACE_NOT_AVAILABLE

MQRC_PERSISTENT_NOT_ALLOWED

MQRC_MSG_TOO_BIG_FOR_Q_MGR

MQRC_MSG_TOO_BIG_FOR_Q

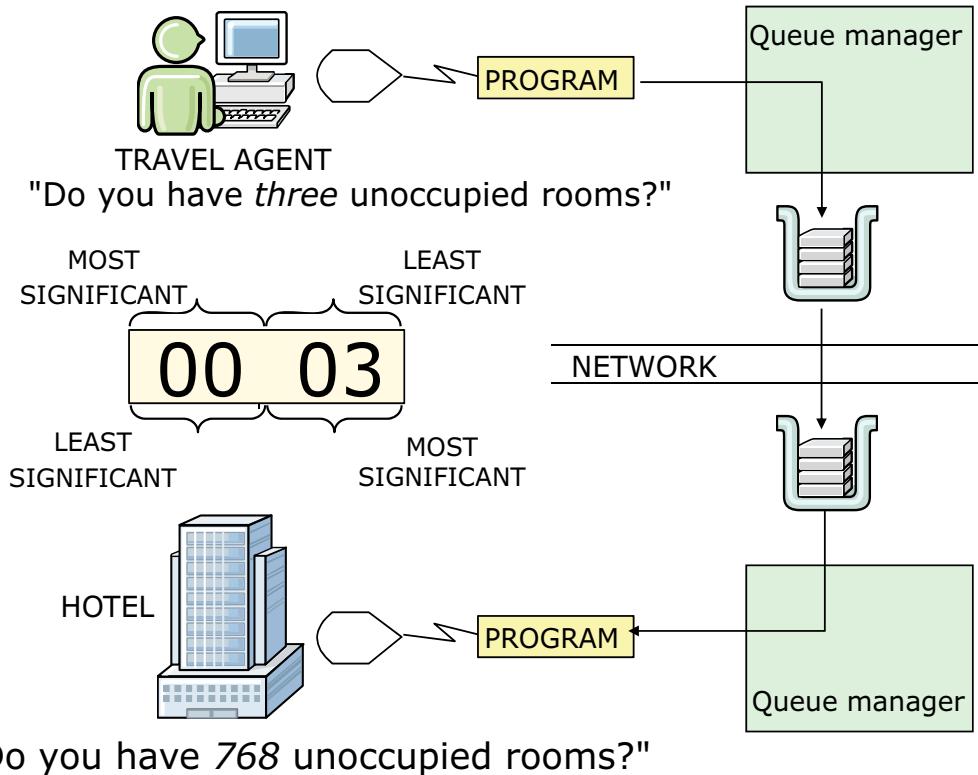
[MQOPEN, queue name resolution, and MQPUT](#)

© Copyright IBM Corporation 2017

Figure 3-20. Feedback

The feedback field is meaningful for message type MQMT_REPORT.

Encoding



MQOPEN, queue name resolution, and MQPUT

© Copyright IBM Corporation 2017

Figure 3-21. Encoding

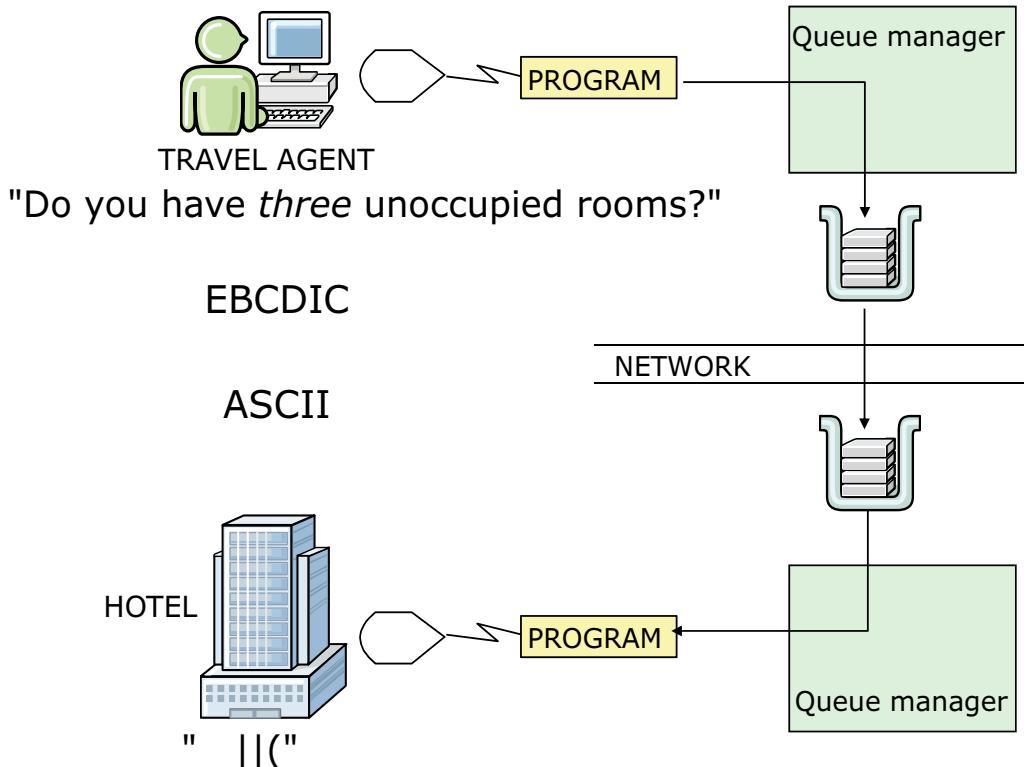
When floating point, decimal, or integer data is put into fields in the application data portion of a message, the data might be represented differently on different operating systems.

In the example, the putting platform is UNIX (that is, AIX) and the getting environment is Intel (that is, Windows). Because of platform differences, a number that is created as 3 in UNIX is interpreted as 768 on Windows. The Encoding field in the MQMD is used to avoid this problem.

The default value is MQENC_NATIVE. It enables the queue manager to put the value into the MQMD_Encoding field that is based on the platform where the program is running and the language in which that the program is written. The destination queue manager can inspect the encoding field in the incoming MQMD and, if requested, convert the numeric fields.

You look at data conversion in a later unit.

CodedCharSetId



MQOPEN, queue name resolution, and MQPUT

© Copyright IBM Corporation 2017

Figure 3-22. CodedCharsetId

Message data can come in many languages, and can be ASCII or EBCDIC.

In this example, you might be going from US English EBCDIC (MVS) to a UK English ASCII system. The receiving system must know that the data that is coming in is in a different form than its own. The coded character set ID field in the message descriptor (MQMD_CCSID) contains that information.

Again, the default (MQCCSI_Q_MGR) allows the correct value to be supplied by the queue manager at MQPUT time. The actual value is the International Standards Organization code page number for that particular language and platform.

A set of tables for all of the supported code pages is in the Application Programming Reference.

You look at data conversion in a later unit.

Message formats (1 of 2)

- Message formats as they appear in the `cmqc.h` header file

```
/* Formats */
#define MQFMT_NONE          "        "
#define MQFMT_ADMIN          "MQADMIN  "
#define MQFMT_AMQP            "MQAMQP   "
#define MQFMT_CHANNEL_COMPLETED "MQCHCOM  "
#define MQFMT_CICS             "MQCICS   "
#define MQFMT_COMMAND_1        "MQCMD1   "
#define MQFMT_COMMAND_2        "MQCMD2   "
#define MQFMT_DEAD_LETTER_HEADER "MQDEAD   "
#define MQFMT_DIST_HEADER      "MQHDIST   "
#define MQFMT_EMBEDDED_PCF     "MQHEPCF  "
#define MQFMT_EVENT             "MQEVENT   "
...
...    ...    ...    ...    ...
```

[MQOPEN, queue name resolution, and MQPUT](#)

© Copyright IBM Corporation 2017

Figure 3-23. Message formats (1 of 2)

The Encoding and CodedCharSetId fields of an incoming message enable the receiving queue manager to convert the MQMD because the queue manager knows the location of the numeric fields and the character fields.

However, the receiving queue manager does not know how the application data portion of the message is designed; it sees only a stream of bytes.

You learn how to handle the contents of your message data in the “Data conversion” unit.

This slide shows predefined formats from the `cmqc.h` header, continued on the next slide.

Message formats (2 of 2)

```
.....
#define MQFMT_IMS           "MQIMS      "
#define MQFMT_IMS_VAR_STRING "MQIMSVS   "
#define MQFMT_MD_EXTENSION  "MQHMDE    "
#define MQFMT_PCF            "MQPCF     "
#define MQFMT_REF_MSG_HEADER "MQHREF    "
#define MQFMT_RF_HEADER     "MQHRF     "
#define MQFMT_RF_HEADER_1   "MQHRF    "
#define MQFMT_RF_HEADER_2   "MQHRF2    "
#define MQFMT_STRING          "MQSTR     "
#define MQFMT_TRIGGER         "MQTRIG    "
#define MQFMT_WORK_INFO_HEADER "MQHWIH    "
```

MQOPEN, queue name resolution, and MQPUT

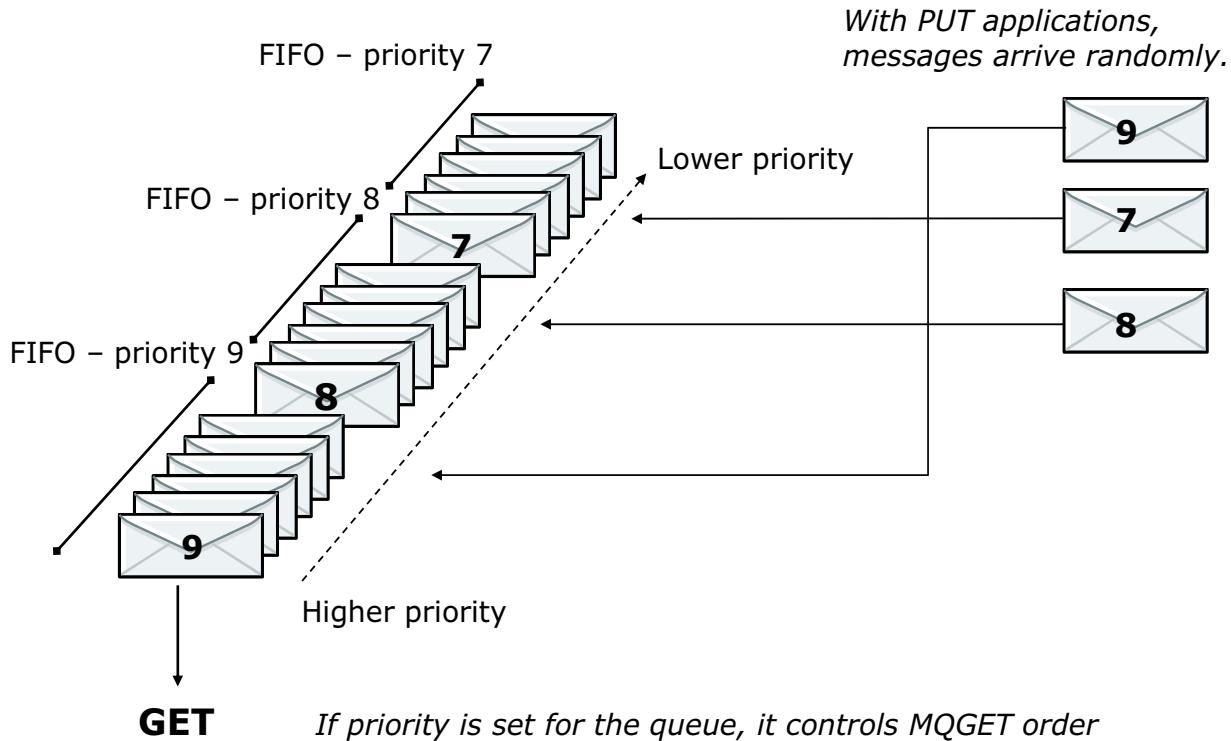
© Copyright IBM Corporation 2017

Figure 3-24. Message formats (2 of 2)

This slide has the continuation of predefined message formats from the `cmsgc.h` header file.

You learn how to handle the contents of your message data in the “Data conversion” unit.

Priority



MQOPEN, queue name resolution, and MQPUT

© Copyright IBM Corporation 2017

Figure 3-25. Priority

Use the Priority field in the message descriptor to set the priority of a message when it is put on a queue. The value can be in the range 0 – 9, or -1, which causes the message to inherit the default priority attribute of the queue.

The MsgDeliverySequence attribute of the queue, set by the administrator, determines how the message is stored on the queue by the queue manager.

- If the value of this attribute is MQMDS_FIFO, messages are enqueued with the default priority of the queue, irrespective of the priority that the application sets.
- If the value of this attribute is MQMDS_PRIORITY, the priority value in the message descriptor is used when enqueueing the message.

A convention when constructing replies is to use the priority value of the incoming request.

Persistence

- Controls whether messages survive restart of the queue manager

MQMD initial values (partial)

```
MQMD_DEFAULT
{MQMD_STRUC_ID_ARRAY}, \
MQMD_VERSION_1, \
MQRO_NONE, \
MQMT_DATAGRAM, \
... ...
MQPER_PERSISTENCE_AS_Q_DEF, \
... ... ...
```

QLOCAL attributes (partial)

```
QUEUE (EX2NONPERS)
TYPE (QLOCAL)
DEFBIND (OPEN)
DEFPSIST (NO)
DEFPRTY (0)
GET (ENABLED)
PUT (ENABLED)
... ... ...
```

Disposition of messages upon restart of the queue manager

Persistence MQI code Queue attribute	DEFPSIST (NO)	DEFPSIST (YES)
MQPER_PERSISTENCE_AS_Q_DEF	Lost	Kept
MQPER_PERSISTENT	Kept	Kept
MQPER_NOT_PERSISTENT	Lost	Lost

MQOPEN, queue name resolution, and MQPUT

© Copyright IBM Corporation 2017

Figure 3-26. Persistence

You might remember this slide from an earlier unit.

The persistence attribute of a message determines whether it survives a restart of the queue manager.

Persistent messages are logged to enable recovery, and this logging might affect performance and system resources such as disk space.

An example of a good reason to use a persistent message is an application that transfers funds between bank accounts.

Non-persistent messages are not logged, and do not survive queue manager restarts. An example of a non-persistent message is an inquiry on the price of a stock.

The application program can set the MQMD persistence field to:

- MQPER_NOT_PERSISTENT (value 0) for non-persistent.
- MQPER_PERSISTENT (value 1) for persistent.
- MQPER_PERSISTENCE_AS_Q_DEF (value 2). It inherits the default persistence characteristic of the queue (DEFPSIST), defined by the administrator.

Reply-to-Q and Reply-To-QMgr processing

- The ReplyToQ and ReplyToQMgr are used by an application to receive a reply message
 - From a responding application, or
 - From the queue manager
- **ReplyToQ**
 - If ReplyToQ specified is a QREMOTE owned by the **local** queue manager, it is changed to the name of the **target** queue (RNAME) specified in the local QREMOTE definition
 - If the ReplyToQ is **not** a QREMOTE, it is not changed
- **ReplyToQMgr**
 - If ReplyToQ is a QREMOTE owned by the **local** queue manager, ReplyToQMgr is set to the target queue manager (RQMNAME) specified in the local QREMOTE definition
 - If the ReplyToQ is **not** a QREMOTE, the ReplyToQMgr is set to the queue manager that the current application is connected to

[MQOPEN, queue name resolution, and MQPUT](#)

© Copyright IBM Corporation 2017

Figure 3-27. Reply-to-Q and Reply-To-QMgr processing

The ReplyToQ and ReplyToQMgr fields are used to specify which queue and queue manager.

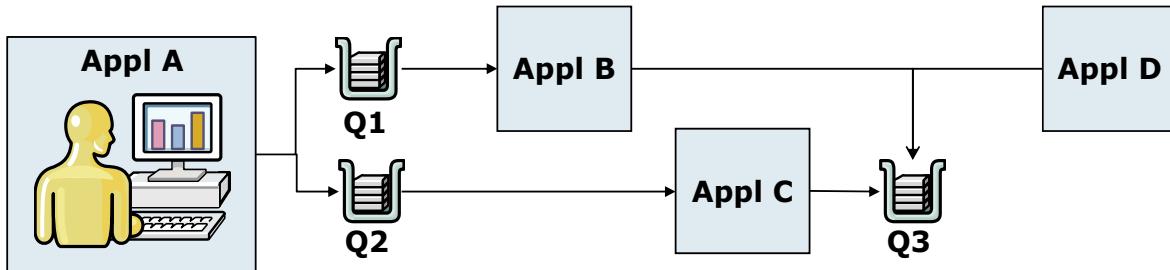
- A request message is sent to an application.
- A report message is sent.

Message context

- Contains information about the source of the message
 - Information about user who put message in the queue in the Identity section
 - Information about the application and timeline in the Origin section
- Context can be passed in a related message

```
** Identity Context
UserIdentifer : 'mqadmin8'
AccountingToken : X'033530 . . .
ApplIdentityData : ' . . . . .

** Origin Context
PutApplType : '6'
PutApplName : 'twoqput . . . '
PutDate: '20160218'
PutTime: '15031393'
ApplOriginData : ' '
```



MQOPEN, queue name resolution, and MQPUT

© Copyright IBM Corporation 2017

Figure 3-28. Message context

Message context allows an application that retrieves a message to find out about the originator of the message. The retrieving application can:

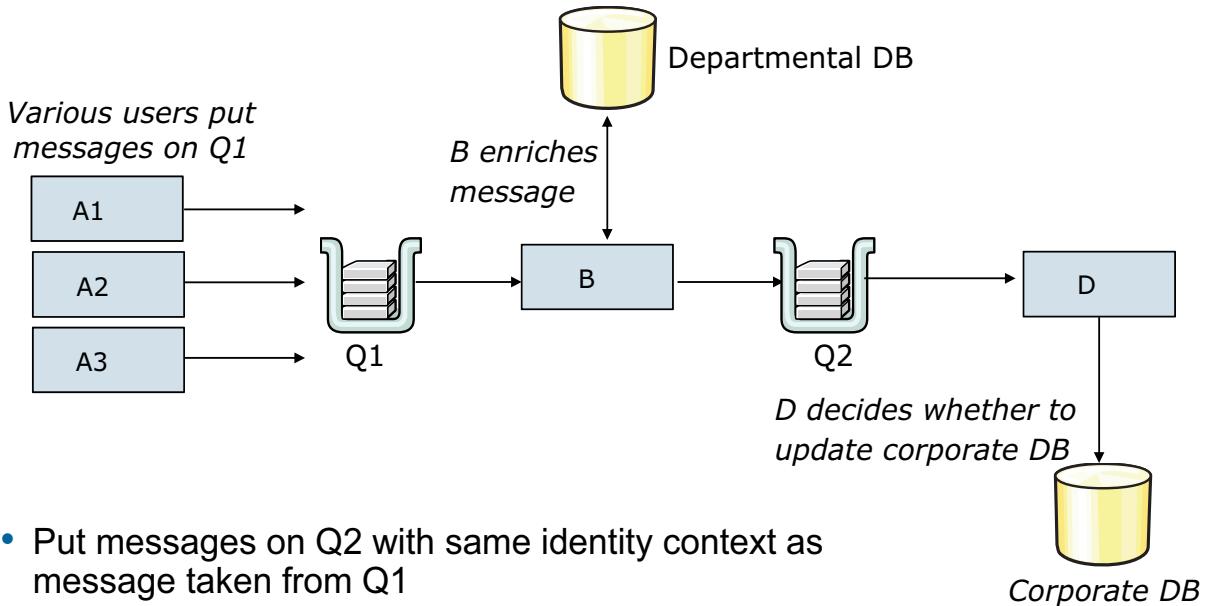
- Check the authority of the sending application
- Perform some accounting functions, such as charge back, by using the information of the initiator
- Keep an audit trail of messages that were processed

The information about the originating user and application is contained in eight fields in the message descriptor. As a group, these fields are called the context fields.

The setting and use of these fields should be carefully controlled. The information consists of two parts:

- Identity data
- Origin data

Passing context



- Put messages on Q2 with same identity context as message taken from Q1
- Open Q1 as “Save All Context”
- Put messages with “Pass Identity Context”

MQOPEN, queue name resolution, and MQPUT

© Copyright IBM Corporation 2017

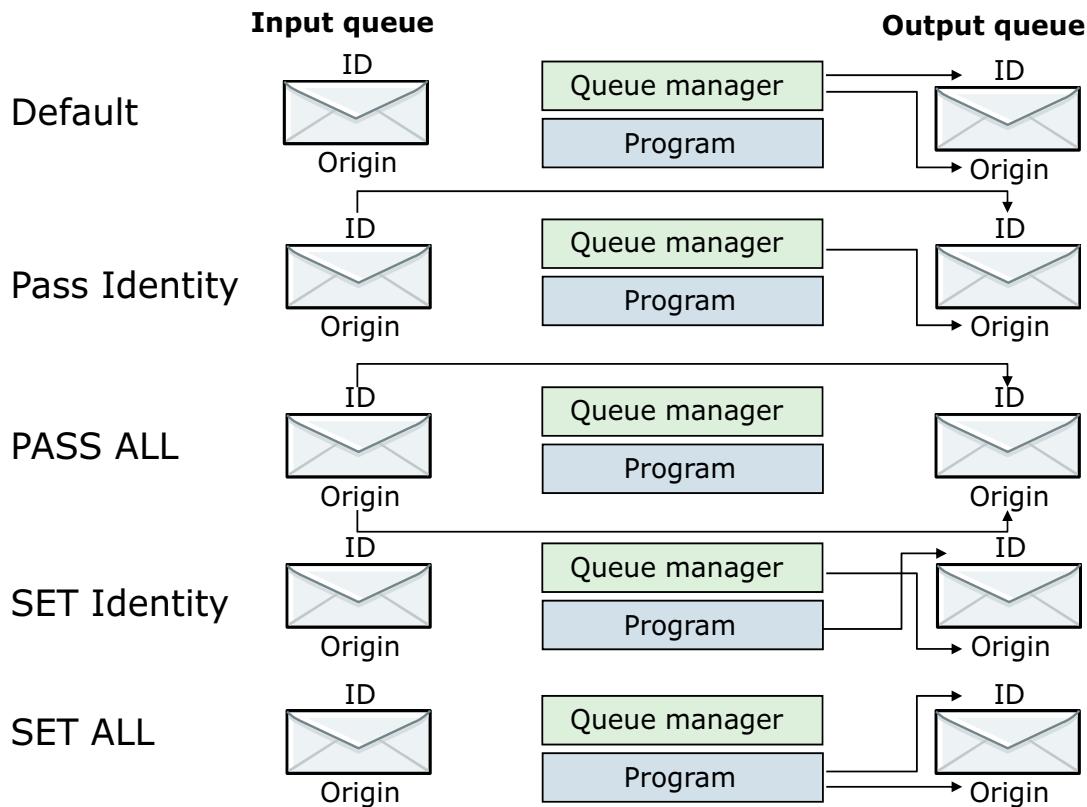
Figure 3-29. Passing context

In this example, various users (A1, A2, A3) initiate a business transaction by putting a message on Q1. The server program B gets the messages from Q1, enriches them with data from the departmental database, and passes them on to Q2. Program D gets the messages from Q2 and updates the corporate database.

If program D must verify the IDs of the requester before updating the corporate database, the context information in the Q2 messages is of no use. It shows that user under whose ID program B is running created each message. The solution is for program B to pass context information from Q1 to Q2.

Context information is passed by using an open option on Q1 of “Save All Context”, an open option on Q2 of “Pass Identity Context”, and the “Pass Identity Context” option on the MQPUT to Q2. The UserIdentifier, AccountingToken, and ApplIdentityData are passed from the input message to the output message. However, because the request was to pass identity context and not to pass all context, the queue manager updates the last five fields (the origin data).

Context handling



MQOPEN, queue name resolution, and MQPUT

© Copyright IBM Corporation 2017

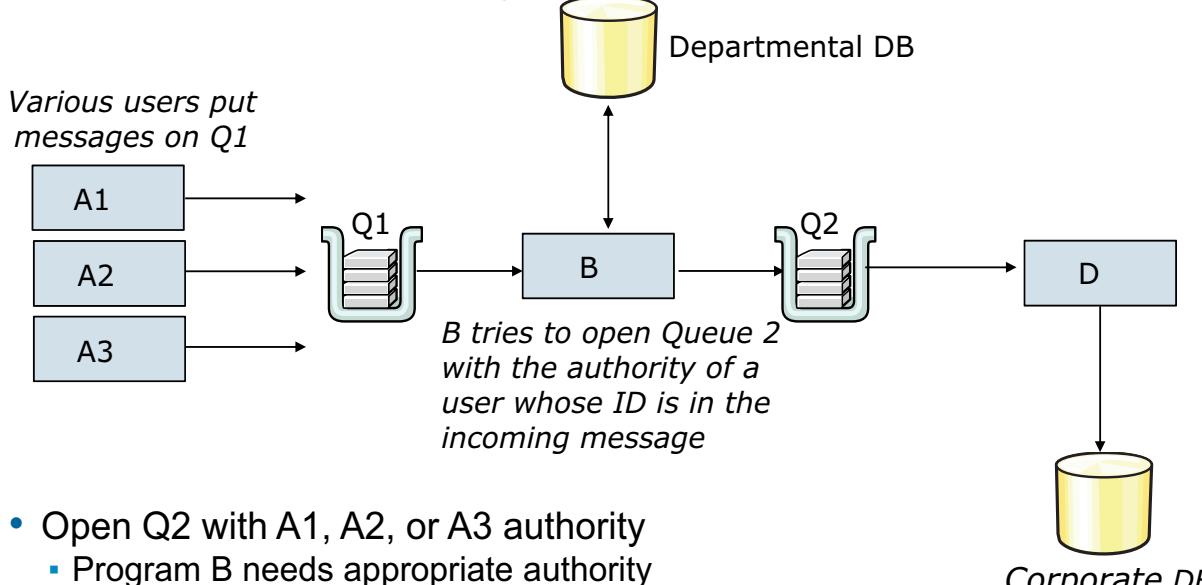
Figure 3-30. Context handling

Putting programs can stipulate how to populate the context fields. The possible options are:

- MQPMO_DEFAULT_CONTEXT: The queue manager supplies both the identity context and the origin context data.
- MQPMO_PASS_IDENTITY_CONTEXT: The queue manager supplies the origin context, but the identity context is copied from the input message. This option assumes that the input queue was opened with the open option of MQOO_SAVE_ALL_CONTEXT and that the output queue was opened with the open option of MQOO_PASS_IDENTITY_CONTEXT.
- MQPMO_PASS_ALL_CONTEXT: All eight of the context fields are passed from the input to the output message. Again, the correct open options are required for the input and output queues.
- MQPMO_SET_IDENTITY_CONTEXT: The queue manager supplies the origin context, but the application is to set the identity context. The queue must be opened for output by using the additional open option MQOO_SET_IDENTITY_CONTEXT.
- MQPMO_SET_ALL_CONTEXT: The program controls the content of all context fields.

Except for the first option, the options that are reviewed succeed only if the user and application that is attempting them have the appropriate authority.

Alternative user authority



- Open Q2 with A1, A2, or A3 authority
 - Program B needs appropriate authority
 - User ID taken from message context
- How it is requested:
 - **AlternateUserId** field in object descriptor
 - Option on MQOPEN or MQPUT1

[MQOPEN, queue name resolution, and MQPUT](#)

© Copyright IBM Corporation 2017

Figure 3-31. Alternative user authority

The scenario in the figure does not rely on application code in program D to decide whether to update the corporate database. Instead, you get the queue manager to prevent program B opening Q2 on behalf of users you do not trust. In this way, the request gets no further.

Alternative user authority enables program B to get a message from the input queue (Q1) and move the UserIdentifier from the context information of the input message to a field in the object descriptor called MQOD_AlternateUserId. If an explicit MQOPEN is done, then the MQOO_ALTERNATE_USER_AUTHORITY is included in the open options, or if an MQPUT1 is used, the more likely choice, MQPMO_ALTERNATE_USER_AUTHORITY, is included in the MQPMO_Options.

When the MQOPEN or MQPUT1 is run, the authority of program B is checked to see whether it is authorized to request alternative user authority. The authority of the user ID that is specified in the MQOD_AlternateUserId is checked to see whether it is authorized to open the queue for output.

Put message options (MQPMO) structure (1 of 2)

```

MQCHAR4  StrucId;          /* Structure identifier      */
MQLONG   Version;          /* Structure version number */
MQLONG   Options;          /* Options that control the */
                           /* action of MQPUT and MQPUT1*/
MQLONG   Timeout;          /* Reserved                 */
MQHOBJ   Context;          /* Object handle of input queue */
MQLONG   KnownDestCount;   /* Number of messages sent */
                           /* successfully to local queues */
MQLONG   UnknownDestCount; /* Number of messages sent */
                           /* successfully to remote queues*/
MQLONG   InvalidDestCount; /* Number of messages that could*/
                           /* not be sent             */
MQCHAR48 ResolvedQName;    /* Resolved name of destination */
                           /* queue                   */
MQCHAR48 ResolvedQMgrName; /* Resolved name of dest Qmgr */
/* Ver:1 */

```

MQOPEN, queue name resolution, and MQPUT

© Copyright IBM Corporation 2017

Figure 3-32. Put message options (MQPMO) structure (1 of 2)

The put message options (MQPMO) structure controls how a message is put on a queue.

The fields that are common to all versions of the MQPMO structure are:

- StrucId: The structure identifier.
- Version: Identifies the MQMD version. Regardless of the version, the layout of MQPMO is identical for the first 128 characters (through the field that is called ResolvedQMgrName).
- Options: Contains a value that is a combination of options that are added in the application. For example:
 - MQPMO_SYNCPOINT or MQPMO_NO_SYNCPOINT
 - MQPMO_FAIL_IF QUIESCING
 - MQPMO_PASS_IDENTITY_CONTEXT
- Timeout: Reserved (not used).
- Context: The object handle of an input queue.
- KnownDestCount, UnknownDestCount, and InvalidDestCount are reserved fields in Version 1 of the MQPMO. The distribution list support, which uses Version 2 of the structure sets these fields.

- ResolvedQName and ResolvedQMgrName contain the resolved names of the destination queue and queue manager.

Your application can use these fields to influence how a message is put on a queue.

Put message options (MQPMO) structure (2 of 2)

```

MQLONG RecsPresent;          /* Number of put message records      */
                           /* or response records present     */
MQLONG PutMsgRecFields;    /* Flags indicating which MQPMR      */
                           /* fields are present             */
MQLONG PutMsgRecOffset;    /* Offset of first put message      */
                           /* record from start of MQPMO     */
MQLONG ResponseRecOffset; /* Offset of first response record */
                           /* from start of MQPMO           */
MQPTR   PutMsgRecPtr;       /* Address of first put message rec*/
MQPTR   ResponseRecPtr;    /* Address of first response record*/
/* Ver:2 */
MQHMSG  OriginalMsgHandle; /* Original message handle         */
MQHMSG  NewMsgHandle;      /* New message handle             */
MQLONG  Action;            /* The action being performed     */
MQLONG  PubLevel;          /* Publication level             */
/* Ver:3 */

```

MQOPEN, queue name resolution, and MQPUT

© Copyright IBM Corporation 2017

Figure 3-33. Put message options (MQPMO) structure (2 of 2)

MQPMO options

Default options:

MQPMO_NONE

Syncpoint options:

MQPMO_SYNCPOINT

MQPMO_NO_SYNCPOINT

Message and correlation identifier options:

MQPMO_NEW_MSG_ID

MQPMO_NEW_CORREL_ID

Group and segment options:

MQPMO_LOGICAL_ORDER

Put response options:

MQPMO_ASYNC_RESPONSE

MQPMO_SYNC_RESPONSE

MQPMO_RESPONSE_AS_Q_DEF

MQPMO_RESPONSE_AS_

TOPIC_DEF

MQOPEN, queue name resolution, and MQPUT

Figure 3-34. MQPMO options

Publishing options are as follows:

- MQPMO_SCOPE_QUEUE_MANAGER
- MQPMO_SUPPRESS_REPLYTO
- MQPMO_RETAIN
- MQPMO_NOT_OWN_SUBS
- MQPMO_WARN_IF_NO_SUBS_MATCHED

Context options:

MQPMO_DEFAULT_CONTEXT

MQPMO_NO_CONTEXT

MQPMO_PASS_IDENTITY_CONTEXT

MQPMO_PASS_ALL_CONTEXT

MQPMO_SET_IDENTITY_CONTEXT

MQPMO_SET_ALL_CONTEXT

Publish/subscribe options:

(In publish/subscribe unit)

Other options:

MQPMO_MD_FOR_OUTPUT_ONLY

Other options:

MQPMO_ALTERNATE_
USER_AUTHORITY

MQPMO_FAIL_IF QUIESCING

MQPMO_RESOLVE_LOCAL_Q

© Copyright IBM Corporation 2017

MQPMO action

- The MQPMO Action field establishes the relationship between the `OriginalMsgHandle` field and the `NewMsgHandle` field
- The queue manager chooses message properties according to the value specified in the `Action` field
- Possible values for `Action` are:

`MQACTP_NEW`

`MQACTP_FORWARD`

`MQACTP_REPLY`

`MQACTP_REPORT`

Figure 3-35. MQPMO action

Put action indicators are used to indicate to the queue manager the type of put action and the relationship between the new message and a possible original message that it might receive earlier.

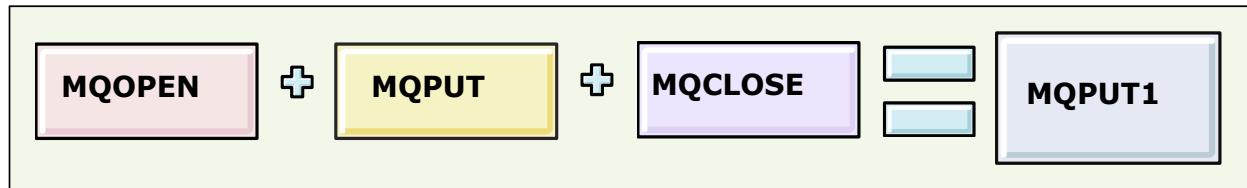
The queue manager uses the action indicators to validate and copy message properties and MQMD values from the original message to the new message handle according to the action.

A message handle stored in MQPMO in the `OriginalMsgHandle` field represents the original message. A message handle stored in the `NewMsgHandle` field represents the new message.

The types of actions that the action field can specify in the MQPMO put options in the MQPUT or MQPUT1 function calls are:

- `MQACTP_NEW`: A new message that is unrelated to any previous message is put to the queue.
- `MQACTP_FORWARD`: A previously retrieved message that was modified is put on the queue for forward processing.
- `MQACTP_REPLY`: The new message is a reply to a previously retrieved message.
- `MQACTP_REPORT`: A report message is generated as result of receiving a message.

The MQPUT1 call



- Which call to use: MQPUT, or MQPUT1?
 - MQPUT1 simplifies code when you need to put one message
 - Use MQPUT to put more than one message
- MQOO_BIND_NOT_FIXED behavior if MQOPEN detects **cluster** queue
- Can be used to put to a distribution

```

MQPUT1 (Hcon, /* connection handle */
        &od,      /* object descriptor */
        &md,      /* message descriptor */
        &pmo,     /* default options */
        messlen,   /* message length */
        buffer,    /* message buffer */
        &CompCode, /* completion code */
        &Reason); /* reason code */
  
```

MQOPEN, queue name resolution, and MQPUT

© Copyright IBM Corporation 2017

Figure 3-36. The MQPUT1 call

As with the MQPUT, the MQPUT1 is used to place the buffer as messages on queues. However, with MQPUT1, no associated MQOPEN or MQCLOSE calls are required as these housekeeping calls are implied in the MQPUT1. A common use for MQPUT1 is when a server program is servicing requests from many different users, and the message descriptor of each request specifies a different reply-to queue. Use of MQPUT1 avoids the additional processing of multiple MQOPEN and MQPUT calls.

The MQPUT1 parameters are a combination of the parameters that are used for object descriptor (MQOPEN), and the parameters that are used for the message descriptor and put message options (MQPUT). The parameters do not include the object handle that is used with MQPUT.

Since the open, put, and close are all done with one call, an object handle is not assigned.

MQPUT reason codes (partial)

- **MQRC_DATA_LENGTH_ERROR** (2010, X'7DA') Data length parameter not valid
- **MQRC_EXPIRY_ERROR** (2013, X'7DD') Expiry time not valid
- **MQRC_INCOMPLETE_GROUP** (2241, X'8C1') Message group not complete
- **MQRC_MSG_TOO_BIG_FOR_Q** (2030, X'7EE') Message length greater than maximum for queue
- **MQRC_MSG_TOO_BIG_FOR_Q_MGR** (2031, X'7EF') Message length greater than maximum for queue manager
- **MQRC_NOT_OPEN_FOR_OUTPUT** (2039, X'7F7') Queue not open for output
- **MQRC_PERSISTENCE_ERROR** (2047, X'7FF') Persistence not valid
- **MQRC_PUT_INHIBITED** (2051, X'803') Put calls inhibited for the queue, for the queue to which this queue resolves, or the topic
- **MQRC_REPORT_OPTIONS_ERROR** (2061, X'80D') Report options in message descriptor not valid
- **MQRC_RFH_ERROR** (2334, X'91E') MQRFH or MQRFH2 structure not valid
- **MQRC_SYNCPOINT_LIMIT_REACHED** (2024, X'7E8') No more messages can be handled within current unit of work

Creating dynamic queues

1. Determine whether you need a temporary or permanent dynamic queue
 - Use `SYSTEM.DEFAULT.MODEL.QUEUE` for a temporary dynamic queue
 - Use `SYSTEM.DURABLE.MODEL.QUEUE` for a permanent dynamic queue
 - Define your own `QMODEL` queue with the required attributes
2. Set `ObjectName` to the model queue name in the `MQOPEN MQOD`
3. Specify the dynamic queue name in the `MQOD DynamicQName`

MQOD DynamicQName	Resulting dynamic queue name
<code>MY.DYNAMIC.QUEUE</code>	<code>MY.DYNAMIC.QUEUE</code>
<code>MYPREFIX.*</code>	<code>MYPREFIX.56C39A6220002...</code>
<code>*</code>	<code>AMQ.B6330A6220002507...</code>

4. Use the `MQOD MQCO_DELETE` or `MQCO_DELETE_PURGE` options in the `MQCLOSE` to delete the dynamic queue

[MQOPEN, queue name resolution, and MQPUT](#)

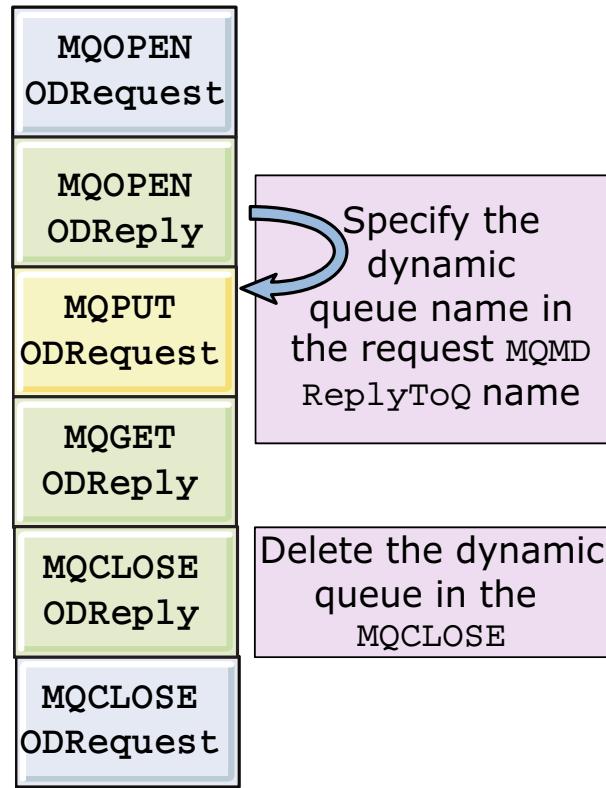
© Copyright IBM Corporation 2017

Figure 3-38. Creating dynamic queues

You create model queues in the lab exercise for this unit.

Common use for a dynamic queue

- A popular use for a dynamic queue is when the queue is not needed after the application ends, such as in a request-reply situation
- MQGET waits for reply
- After the response is received, the dynamic queue can be discarded



MQOPEN, queue name resolution, and MQPUT

© Copyright IBM Corporation 2017

Figure 3-39. Common use for a dynamic queue

Unit summary

- Describe the details that the MQOPEN call handles
- Identify the information in the object descriptor (MQOD) structure
- Describe the options that can be specified in the MQOPEN call
- Describe how the MQOPEN call processes queue name resolution
- Explain the use of fields in the message descriptor (MQMD) structure
- Describe how the IBM MQ V8.0.0.4 expiry cap overrides higher expiry specifications in the application
- Describe various uses of Report messages
- Describe the two types of context information and how context can be used to identify the user of an application
- Examine use of the MQPUT1 call and identify optimal scenarios for its use
- Explain how to create and remove temporary or permanent dynamic queues

Review questions (1 of 2)

1. True or False: Queue name resolution takes place during MQOPEN processing.
2. True or False: An application must always open a QREMOTE type queue to put a message to a remote queue manager.
3. The MQPUT, MQOPEN, MQCONN, MQCONN, and MQGET all require which one of these parameters in common?
 - a. Object handle
 - b. Message descriptor
 - c. Message handle
 - d. Connection handle



MQOPEN, queue name resolution, and MQPUT

© Copyright IBM Corporation 2017

Figure 3-41. Review questions (1 of 2)

Write your answers here:

- 1.
- 2.
- 3.

Review questions (2 of 2)

4. Which of the following MQOPEN options might result in queue manager affinities in the application?
 - a. MQOO_BIND_ON_OPEN
 - b. MQOO_PASS_IDENTITY_CONTEXT
 - c. MQOO_OUTPUT
 - d. MQOO_FAIL_IF_QUIESCING
5. An application wants to create a dynamic queue by setting the MQOD ObjectName field to SYSTEM.DEFAULT.MODEL.QUEUE, and the MQOD DynamicQName field to BILL.RECEIPTS. What are the results?
 - a. A permanent dynamic queue that is named BILL.RECEIPTS.xxxxx where xxxx is replaced with a unique identifier
 - b. A temporary dynamic queue that is named BILL.RECEIPTS.xxxxx where xxxx is replaced with a unique identifier
 - c. A temporary dynamic queue named BILL.RECEIPTS
 - d. A permanent dynamic queue named BILL.RECEIPTS

[MQOPEN, queue name resolution, and MQPUT](#)

© Copyright IBM Corporation 2017

Figure 3-42. Review questions (2 of 2)

Write your answers here:

4.

5.



Review answers (1 of 4)

1. True or False: Queue name resolution takes place during MQOPEN processing.

The answer is: True. The queue manager inspects the value that is placed in the MQOD queue manager name and the MQOD queue name to determine resolution.



2. True or False: An application must always open a QREMOTE type queue to put a message to a remote queue manager.

The answer is: False. If the queue manager locates a local transmission queue with the same name as the queue manager name that is specified in the MQOD, a message can be sent to a remote queue manager. (Note: The assumption is that a channel servicing that transmission queue is available and started.)

Review answers (2 of 4)

3. The MQPUT, MQOPEN, MQCONN, MQCONNX, and MQGET all require which one of these parameters **in common**?
- a. Object handle
 - b. Message descriptor
 - c. Message handle
 - d. Connection handle

The answer is: d, The MQPUT, MQOPEN, MQCONN, MQCONNX, and MQGET all require a connection handle.
All calls also require a return code and reason code, but depending on the application, the return code fields might not be shared, but allocated separately for some calls.



Review answers (3 of 4)

4. Which of the following MQOPEN options might result in queue manager affinities in the application?
- a. MQOO_BIND_ON_OPEN
 - b. MQOO_PASS_IDENTITY_CONTEXT
 - c. MQOO_OUTPUT
 - d. MQOO_FAIL_IF_QUIESCING



The answer is: a, MQOO_BIND_ON_OPEN causes all messages that are put after the MQOPEN call to be sent to the same queue manager, undermining cluster architecture objectives.

Review answers (4 of 4)



5. An application wants to create a dynamic queue by setting the MQOD ObjectName field to SYSTEM.DEFAULT.MODEL.QUEUE, and the MQOD DynamicQName field to BILL.RECEIPTS. What are the results?
- A permanent dynamic queue that is named BILL.RECEIPTS.xxxxx where xxxx is replaced with a unique identifier
 - A temporary dynamic queue that is named BILL.RECEIPTS.xxxxx where xxxx is replaced with a unique identifier
 - A temporary dynamic queue named BILL.RECEIPTS
 - A permanent dynamic queue named BILL.RECEIPTS

The answer is: c, SYSTEM.DEFAULT.MODEL.QUEUE is a QMODEL template for temporary queues. Without “*”, the name is created as specified in the MQOD DynamicQName field.

Working with MQOPEN and queue name resolution, MQPUT, and MQMD fields

MQOPEN, queue name resolution, and MQPUT

© Copyright IBM Corporation 2017

Figure 3-47. Working with MQOPEN and queue name resolution, MQPUT, and MQMD fields

Exercise objectives

- Code various combinations of queue manager and queue name in the object descriptor to test and confirm how queue name resolution takes place
- Code report options and review the results in the reply-to queue
- Create and display a dynamic queue where the queue manager determines the name
- Create and display a dynamic queue by specifying a partial prefix of the queue name
- Create and display a dynamic queue by specifying the exact queue name
- Request a confirm-on-arrival with data report message
- Set the expiry attribute in a message and request an expiry report
- Set the expiry attribute in a message for a queue with a lower expiry cap value in the queue definition



MQOPEN, queue name resolution, and MQPUT

© Copyright IBM Corporation 2017

Figure 3-48. Exercise objectives

Unit 4. Getting messages and retrieval considerations

Estimated time

01:00

Overview

This unit describes the various ways to retrieve messages from a queue.

How you will check your progress

Accountability:

- Review questions
- Lab Exercises

References

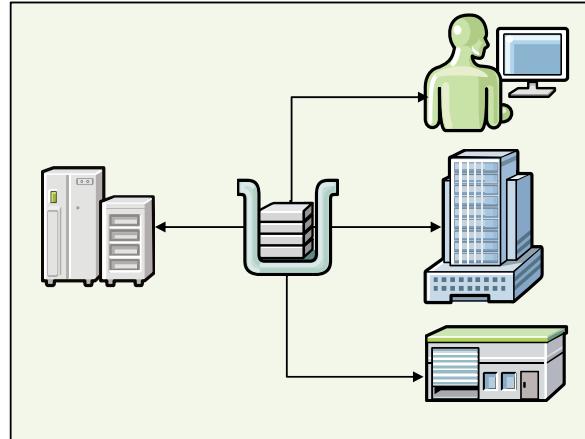
IBM Knowledge Center for IBM MQ V9 documentation

Unit objectives

- Describe the parameters that are required for the MQGET call
- Describe the MQGET call option groupings
- Explain how to associate requests with responses by using the message and correlation IDs
- Differentiate between the options that are used to browse messages
- Explain how to use message tokens to browse a queue
- Explain the use and need for message marks and cooperative browsing
- Describe how to write code that waits for responses

How are messages retrieved from a queue?

- First-in, first-out (FIFO) within priority
- Getting specific messages
 - Message ID
 - Correlation ID
 - Group ID, message sequence number, and offset
 - Message token
- Browsing (non-destructive)
- Cooperative browsing
- Waiting for a message



[Getting messages and retrieval considerations](#)

© Copyright IBM Corporation 2017

Figure 4-2. How are messages retrieved from a queue?

Applications have different requirements to retrieve messages. A request/reply application might need to retrieve a specific message, while other applications get messages off the queue in first-in first-out order according to message arrival or arrival within message priority.

Aside from the order in which messages are retrieved, you must consider other factors, such as:

- Does the message need to be removed from the queue or left in the queue? Sometimes whether to browse or remove is termed “destructive” or “non-destructive” retrieval.
- Are there applications that cooperate so they do not browse the same message that another application already browsed?
- If the message is not there, how long do you wait for it?

MQOPEN options used with MQGET

Browsing options:

MQOO_BROWSE

Options for removing messages:

MQOO_INPUT_AS_Q_DEF

MQOO_INPUT_SHARED

MQOO_INPUT_EXCLUSIVE

Read ahead options

MQOO_NO_READ_AHEAD

MQOO_READ_AHEAD

MQOO_READ_AHEAD_A_Q_DEF

Other options:

MQOO_FAIL_IF QUIESCING

Must use at least one of these options

Exclusive to IBM MQ client non-persistent message retrieval

Figure 4-3. MQOPEN options used with MQGET

Before you code MQGET, the MQOPEN function call for your queue must use one of these options:

- MQOO_BROWSE retrieves the contents of a message without removing the message from the queue.
- If an application uses a cooperative option, the MQOO_BROWSE must be used. You look at cooperative browsing later in this unit.

In addition to the use of one of the required options, you can also use one of the options that follow. These options are valid for local, alias, and model queues.

- MQOO_INPUT_AS_Q_DEF uses the input attributes of the queue definition.
- MQOO_INPUT_EXCLUSIVE provides exclusive access to the queue.
- MQOO_INPUT_SHARED allows more than one application to access the queue. If an outstanding MQOPEN that used MQOO_INPUT_EXCLUSIVE was done first, the call fails with MQRC_OBJECT_IN_USE.

MQGET declarations and parameters

```
MQOD    od = {MQOD_DEFAULT}; /* Object Descriptor */
MQMD    md = {MQMD_DEFAULT}; /* Message Descriptor */
MQGMO   gmo = {MQGMO_DEFAULT};/* get message options*/
MQCNO   cno = {MQCNO_DEFAULT};/* connection options */
...
MQGET(Hcon,          → /* connection handle */ */
       Hobj,          → /* object handle */ */
       &md,           ↔ /* message descriptor */ */
       &gmo,           ↔ /* get message options */ */
       buflen,         → /* buffer length */ */
       buffer,         ← /* message buffer */ */
       &messlen,        ← /* message length */ */
       &CompCode,       ← /* completion code */ */
       &Reason);      ← /* reason code */
```

Getting messages and retrieval considerations

© Copyright IBM Corporation 2017

Figure 4-4. MOGET declarations and parameters

MQGET is used to retrieve messages from a queue previously opened with one of the MQOO_INPUT options. Unless an MQOO_BROWSE option is used, messages are destructively removed from the queue and returned to the application buffer.

- The connection handle from the MQCONN or MQCONNXX call, and the object handle from the MQOPEN call, are the first two arguments.
 - The message descriptor is both an input and an output structure, as is the Get Message Options structure.
 - For the MQGET, the buffer is an output field.
 - BufferLength tells the queue manager the length of the buffer into which to return the application data portion of the message.
 - The buffer holds the message data.
 - The queue manager uses the DataLength attribute (declared as `messlen` in the display) to return the actual length of the message data that was retrieved. In this way, the program can use a general-purpose buffer that can contain small or large messages. The program can determine the actual length of each message that the DataLength value retrieves.

Get message options (MQGMO) structure (1 of 2)

```

MQCHAR4  StrucId;          /* Structure identifier */
MQLONG   Version;         /* Structure version nbr */
MQLONG   Options;         /* MQGET control options */
MQLONG   WaitInterval;    /* Wait interval */
MQLONG   Signal1;         /* Signal */
MQLONG   Signal2;         /* Signal identifier */
MQCHAR48 ResolvedQName;   /* Resolved target name */

/* Ver:1 */

MQLONG   MatchOptions;    /* Select criteria optns */
MQCHAR   GroupStatus;     /* Part of group flag */
MQCHAR   SegmentStatus;   /* Logical segment flag */
MQCHAR   Segmentation;    /* Further segmentation OK*/
MQCHAR   Reserved1;       /* Reserved */

/* Ver:2 */

```

[Getting messages and retrieval considerations](#)

© Copyright IBM Corporation 2017

Figure 4-5. Get message options (MQGMO) structure (1 of 2)

The IBM MQ Get Message Options (MQGMO) fields contain the options that an application specifies to control the behavior of the MQGET.

MQGMO begins with an identifier (GMO), followed by a version number.

- MQGMO_VERSION_1 is supported on all platforms.
- MQGMO_VERSION_2 contains information that is related to message groups and message segments.

This explanation is continued on the next slide.

Get message options (MQGMO) structure (2 of 2)

```
...  
MQBYTE16 MsgToken;          /* Message token */  
MQLONG   ReturnedLength; /* Msg data returned len */  
/* Ver:3 */  
MQLONG   Reserved2;        /* Reserved */  
MQHMSG   MsgHandle;        /* Message handle */  
/* Ver:4 */
```

Getting messages and retrieval considerations

© Copyright IBM Corporation 2017

Figure 4-6. Get message options (MQGMO) structure (2 of 2)

The following information is continued from the previous slide:

- MQGMO_VERSION_3 contains information about message tokens.
- MQGMO_VERSION_4 contains information about message handles.

MQGMO options (1 of 2)

MQGMO_DEFAULT options:

MQGMO_NO_WAIT
MQGMO_PROPERTIES_
AS_Q_DEF

Default option:

MQGMO_NONE

Wait options:

MQGMO_WAIT
MQGMO_NO_WAIT
MQGMO_SET_SIGNAL
MQGMO_FAIL_IF QUIESCING

Sync point options

MQGMO_SYNCPOINT
MQGMO_SYNCPOINT_
IF_PERSISTENT
MQGMO_NO_SYNCPOINT
MQGMO_MARK_SKIP_BACKOUT

Browse options

MQGMO_BROWSE_FIRST
MQGMO_BROWSE_NEXT
MQGMO_BROWSE_MSG_
UNDER_CURSOR
MQGMO_MSG_UNDER_CURSOR
MQGMO_MARK_BROWSE_HANDLE
MQGMO_MARK_BROWSE_CO_OP
MQGMO_UNMARK_BROWSE_CO_OP
MQGMO_UNMARK_BROWSE_HANDLE
MQGMO_UNMARKED_BROWSE_MSG

Message-data options

MQGMO_ACCEPT_TRUNCATED_MSG
MQGMO_CONVERT

Lock options:

MQGMO_LOCK
MQGMO_UNLOCK

[Getting messages and retrieval considerations](#)

© Copyright IBM Corporation 2017

Figure 4-7. MQGMO options (1 of 2)

The following information pertains to MGMO options:

- MQGMO_WAIT: Wait until a message arrives on the queue. This option is generally used with the WaitInterval field, a value that specifies how long to wait for a message to arrive before a failure code is returned.
- MQGMO_SYNCPOINT or MQGMO_NO_SYNCPOINT: Establish whether the message should be part of a unit of work, or immediately deleted from the input queue.
- MQGMO_ACCEPT_TRUNCATED_MESSAGE: Fit as much of the message into the buffer as fits and discard the remainder. If not specified and a message is larger than the buffer length as declared in the MQGET, the call fails.
- MQGMO_CONVERT: If MQMD_ENCODING or MQMD_CCSID on the inbound message differs from the values of the environment where the MQGET is done, use the value in MQMD_FORMAT to allow data to be converted to a usable state.
- MQGMO_BROWSE_FIRST, MQGMO_BROWSE_NEXT: Retrieve a copy of the requested message into the application buffer, leaving the message on the input queue.

The MQGMO structure is an input and output structure. If an application does an MQGET on an alias queue, the name of the base queue is returned in the ResolvedQName field when the queue manager retrieves the message.

The MatchOptions allow the application to choose which fields in the MsgDesc parameter are used to select the message that the MQGET call returns. The application sets the required options in this field, and then sets the corresponding fields in the MsgDesc parameter to the values required for those fields. Only messages that have those values in the MQMD for the message are candidates for retrieval by using that MsgDesc parameter on the MQGET call.

MQGMO options (2 of 2)

Group and segment options:

MQGMO_LOGICAL_ORDER
MQGMO_COMPLETE_MSG
MQGMO_ALL_MSGS_AVAILABLE
MQGMO_ALL_SEGMENTS_AVAILABLE

Property options

MQGMO_PROPERTIES_FORCE_MQRFH2
MQGMO_NO_PROPERTIES
MQGMO_PROPERTIES_IN_HANDLE
MQGMO_PROPERTIES_COMPATIBILITY
MQGMO_PROPERTIES_AS_Q_DEF

[Getting messages and retrieval considerations](#)

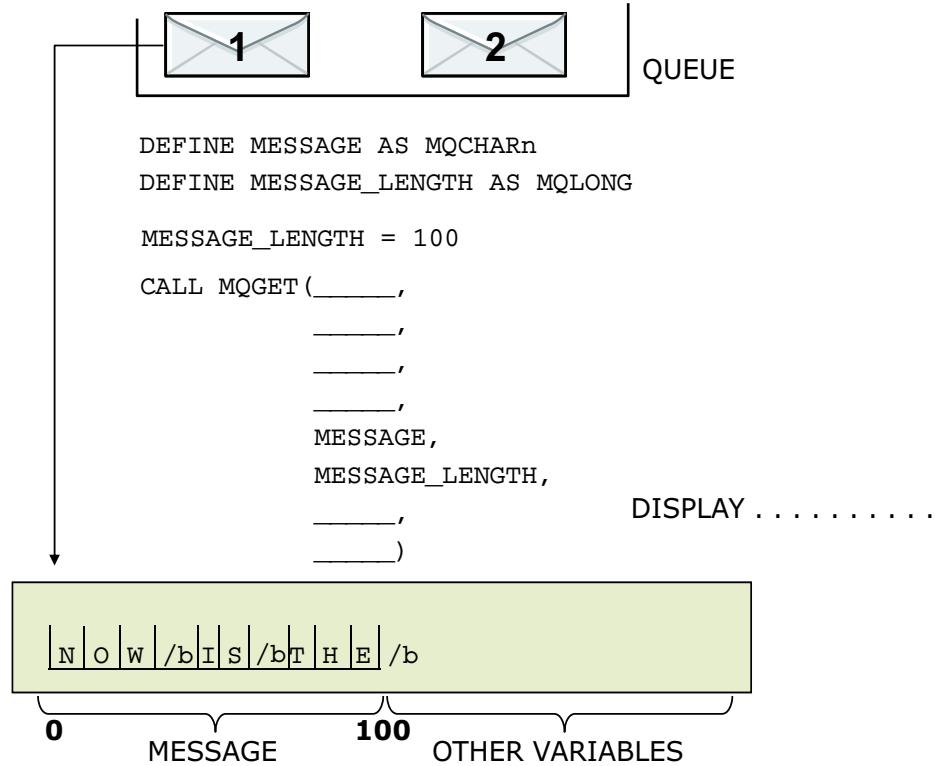
© Copyright IBM Corporation 2017

Figure 4-8. MQGMO options (2 of 2)

You look at the group and segment options in a later unit.

A message property is data that is associated with a message, consisting of a textual name and a value of a particular type. Message properties are used by message selectors to filter publications to topics or to selectively get messages from queues. Message properties can be used to include business data or state information without having to store it in the application data.

Buffer length (1 of 2)



Getting messages and retrieval considerations

© Copyright IBM Corporation 2017

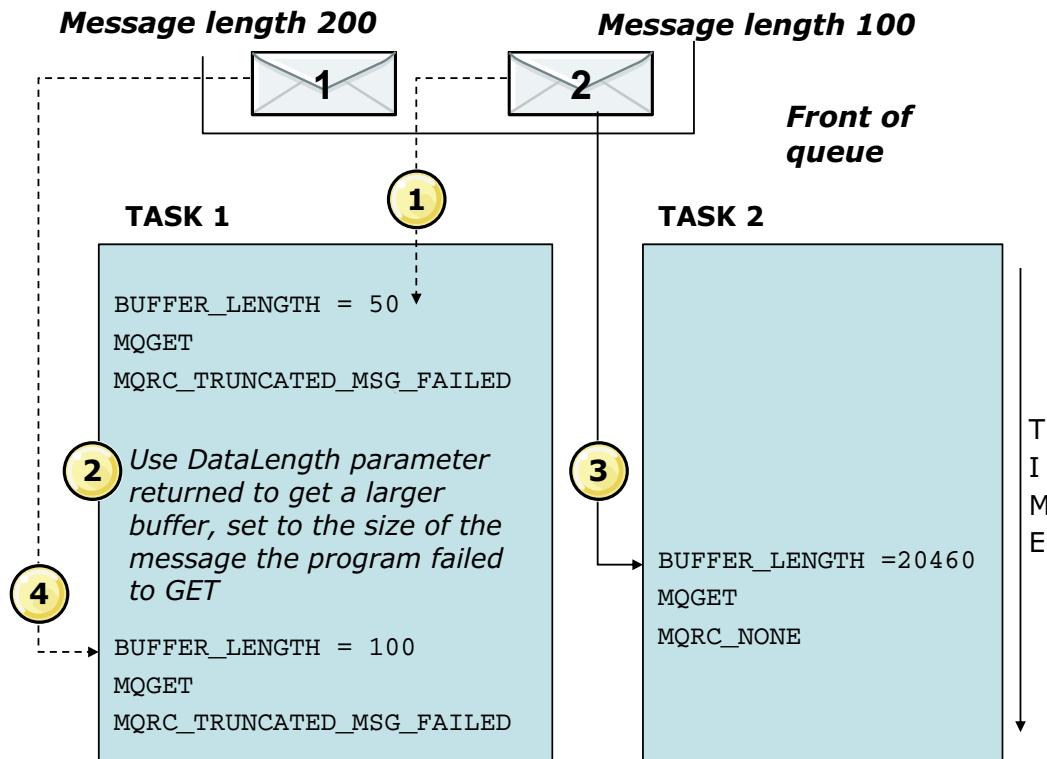
Figure 4-9. Buffer length (1 of 2)

It is important that the value that is used for buffer length is not larger than the actual buffer in the program. If it is, the MQGET can return a message that might be larger than the work area you established, and non-buffer storage can be overwritten.

It is safest to ensure that the buffer length and the buffer agree on the length. It is possible to set the length parameter to less than the actual size of the buffer. If the message is no larger than the stated buffer length, it is placed into the buffer. It is important to remember that you might have data that remains from a previous GET, which goes beyond the current message length.

If a message might be larger than the buffer length specified, it is possible to retrieve that portion of the message that fits in the buffer and discard any excess by using the MQGMO_ACCEPT_TRUNCATED_MESSAGE MQGMO option. In this case, the message is removed from the queue. If this option is not specified, the MQGET fails if a message larger than the buffer is encountered. However, the buffer contains the portion of the message that fits even though that message remains on the queue. In both cases, the length of the complete application data portion of the message is returned in DataLength.

Buffer length (2 of 2)



Getting messages and retrieval considerations

© Copyright IBM Corporation 2017

Figure 4-10. Buffer length (2 of 2)

In this example, numbers 1 – 4 represent the sequence of events.

1. Task 1 attempts to get the first message, which has a length of 100. Because the MQGET specified a buffer length of only 50, the MQGET fails.
2. Using the value that the queue manager returns in the DataLength parameter, Task1 acquires a buffer of 100 by using the provided mechanism of the programming language.
3. Meanwhile, Task 2 specifies a large buffer size and issues an MQGET to successfully retrieve the message.
4. By the time Task 1 reissues the MQGET, the first message is no longer on the queue, and the MQGET fails again because the message is too large.

This example also illustrates the effect of opening a queue with MQOO_INPUT_SHARED as opposed to MQOO_INPUT_EXCLUSIVE.

MQGET reason codes CompCode MQCC_WARNING (partial)

- **MQRC_CONVERTED_MSG_TOO_BIG** (2120, X'848') Converted data too large for buffer
- **MQRC_INCOMPLETE_GROUP** (2241, X'8C1') Message group not complete
- **MQRC_INCONSISTENT_CCSIDS** (2243, X'8C3') Message segments have differing CCSID
- **MQRC_INCONSISTENT_UOW** (2245, X'8C5') Inconsistent unit-of-work specification
- **MQRC_MSG_TOKEN_ERROR** (2331, X'91B') Invalid use of message token
- **MQRC_NOT_CONVERTED** (2119, X'847') Message data not converted
- **MQRC_OPTIONS_CHANGED** (2457, X'999') Options that were required to be consistent are changed
- **MQRC_SOURCE_BUFFER_ERROR** (2145, X'861') Source buffer parameter not valid
- **MQRC_TARGET_DECIMAL_ENC_ERROR** (2117, X'845') Packed-decimal encoding specified by receiver is not recognized
- **MQRC_TRUNCATED_MSG_ACCEPTED** (2079, X'81F') Truncated message returned (processing completed)

[Getting messages and retrieval considerations](#)

© Copyright IBM Corporation 2017

Figure 4-11. MQGET reason codes CompCode MQCC_WARNING (partial)

The figure shows some of many possible reason codes that can be returned after an MQGET. A complete list of the reasons is contained in the Application Programming Reference manual.

MQRC_NONE is returned only if the completion code is MQCC_OK.

The remaining reasons can be a result of a completion code of MQCC_WARNING or MQCC_FAILED.

This explanation is continued in the next slide.

MQGET reason codes CompCode MQCC_FAILED (partial)

- **MQRC_BUFFER_ERROR** (2004, X'7D4') Buffer parameter not valid
- **MQRC_CICS_WAIT_FAILED** (2140, X'85C') Wait req rejected by CICS
- **MQRC_CORREL_ID_ERROR** (2207, X'89F') Correlation-identifier error
- **MQRC_GET_INHIBITED** (2016, X'7E0') Gets inhibited for the queue
- **MQRC_GLOBAL_UOW_CONFLICT** (2351, X'92F') Global units of work conflict
- **MQRC_GMO_ERROR** (2186, X'88A') Get-message options structure not valid
- **MQRC_INCONSISTENT_BROWSE** (2259, X'8D3') Inconsistent browse specification
- **MQRC_NO_MSG_AVAILABLE** (2033, X'7F1') No message available
- **MQRC_NOT_OPEN_FOR_BROWSE** (2036, X'7F4') Queue not open for browse
- **MQRC_UNEXPECTED_ERROR** (2195, X'893') Unexpected error occurred

Getting messages and retrieval considerations

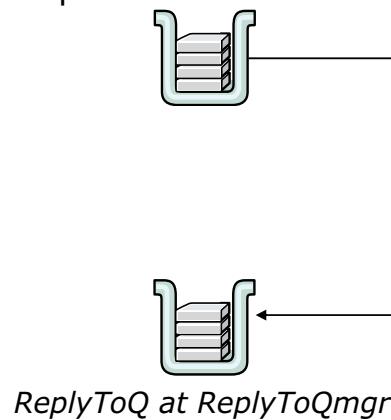
© Copyright IBM Corporation 2017

Figure 4-12. MQGET reason codes CompCode MQCC_FAILED (partial)

This slide shows some additional reason codes that the MQGET call might return.

Sending replies to the reply-to queue

Incoming request message contains the names of the ReplyToQ and ReplyToQmgr in the message descriptor



Responding application

MQGET

IF the message is a request

Process the data

Process the data from the MQGET MQMD_ReplyToQ and MQMD_ReplyToQmgr

END-IF

[Getting messages and retrieval considerations](#)

© Copyright IBM Corporation 2017

Figure 4-13. Sending replies to the reply-to queue

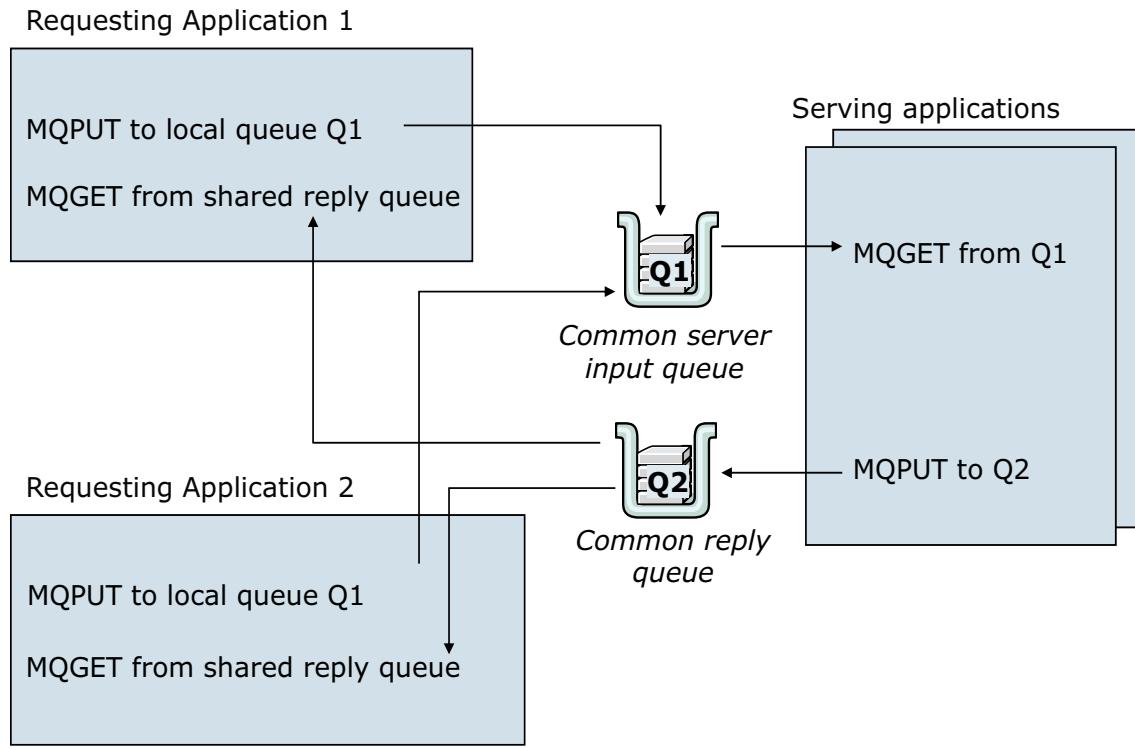
When a program issues an MQGET, it might be designed to interrogate the message descriptor MsgType field to validate that the message is a request. If not, an error process might be started if the responding application always expects request type messages.

After the message type is validated and business processing is completed, a reply message can be constructed on the ReplyToQ at the ReplyToQmgr.

Since the queue that is receiving the responses might not be on the local queue manager, the ReplyToQMgr is significant.

The queue manager looks for a transmission queue with the name of the ReplyToQMgr, and that is where the message is placed (with the correct transmission header that contains routing information) for transmission by the channel.

Message ID and correlation ID



[Getting messages and retrieval considerations](#)

© Copyright IBM Corporation 2017

Figure 4-14. Message ID and correlation ID

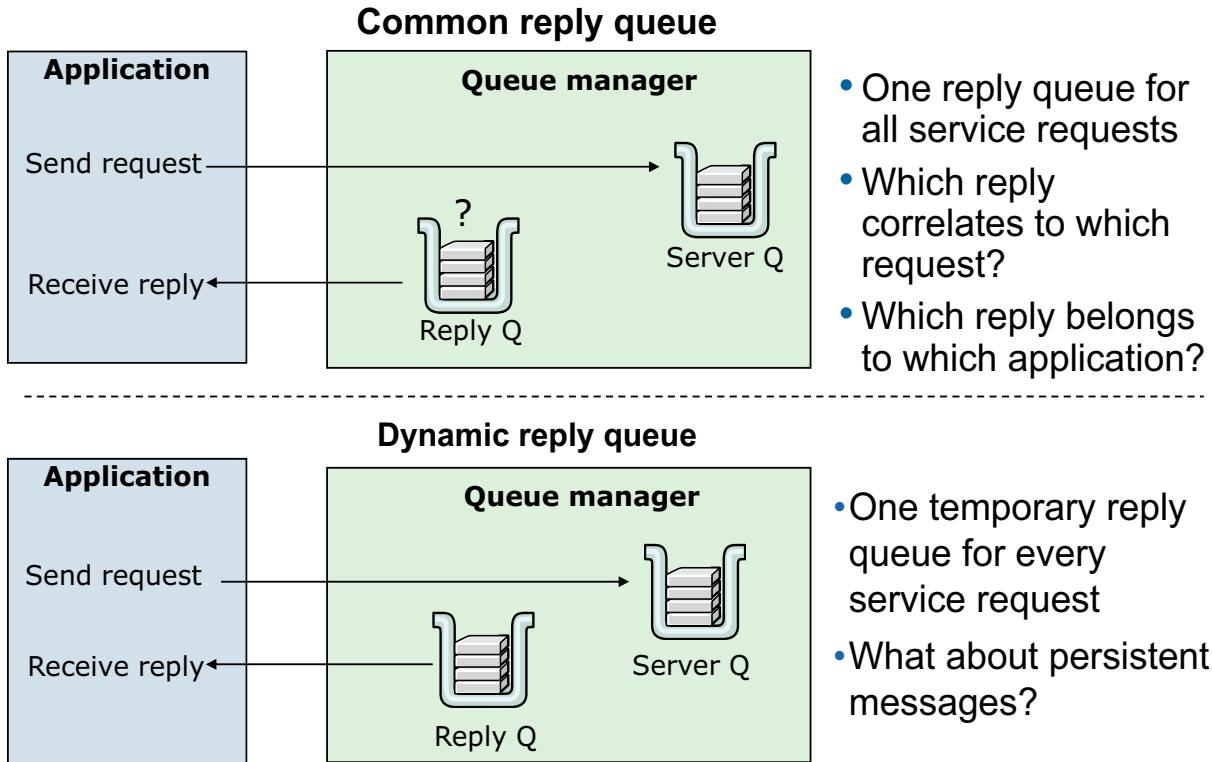
In the figure, a continuously running server is servicing requests for information:

- Several copies of this application can be running simultaneously.
- The reply-to queue is the same for all requesting applications.
- The requesting application uses MQGET to get replies sent by the server. It must ensure that the reply it gets relates to the request it sent, ignoring any others.

If an application issues an MQGET, the message is destructively taken from the queue. The application should not remove other replies that are not for its request. It is possible to use the BROWSE function and check each message, but that can be wasteful since more MQGETs would be issued just to examine each message.

Two fields in the message descriptor allow an application to selectively retrieve messages. The message ID and correlation ID fields can be used in this scenario to ensure that the reply message that is retrieved matches the request that was sent.

Request and reply queue consideration



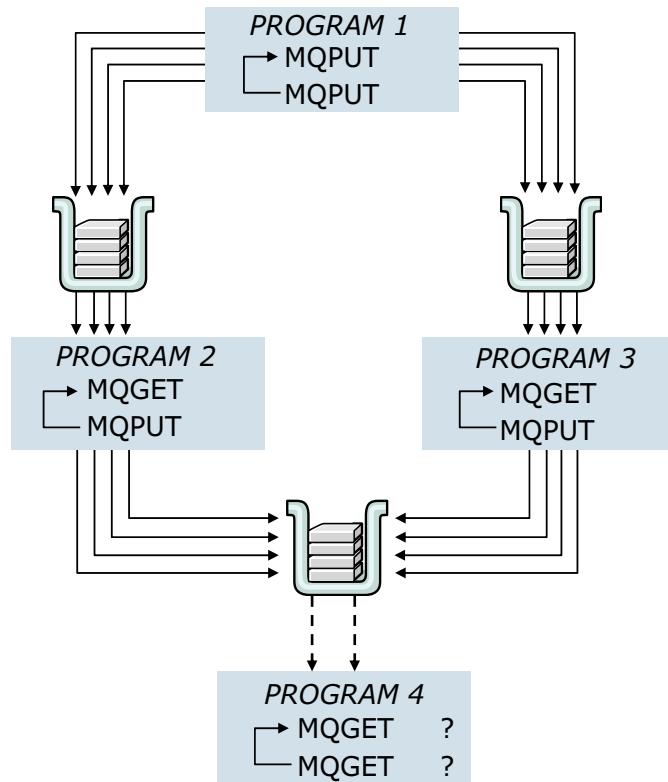
Getting messages and retrieval considerations

© Copyright IBM Corporation 2017

Figure 4-15. Request and reply queue consideration

It is possible to use a separate reply-to queue for each separate requesting application. These queues are usually dynamic queues for easier administration. When dynamic reply queues are used, a message ID and correlation ID still might be necessary to correlate the replies when the requesting application sends several requests before receiving the replies.

MsgId, CorrelId, and application parallelism



Getting messages and retrieval considerations

© Copyright IBM Corporation 2017

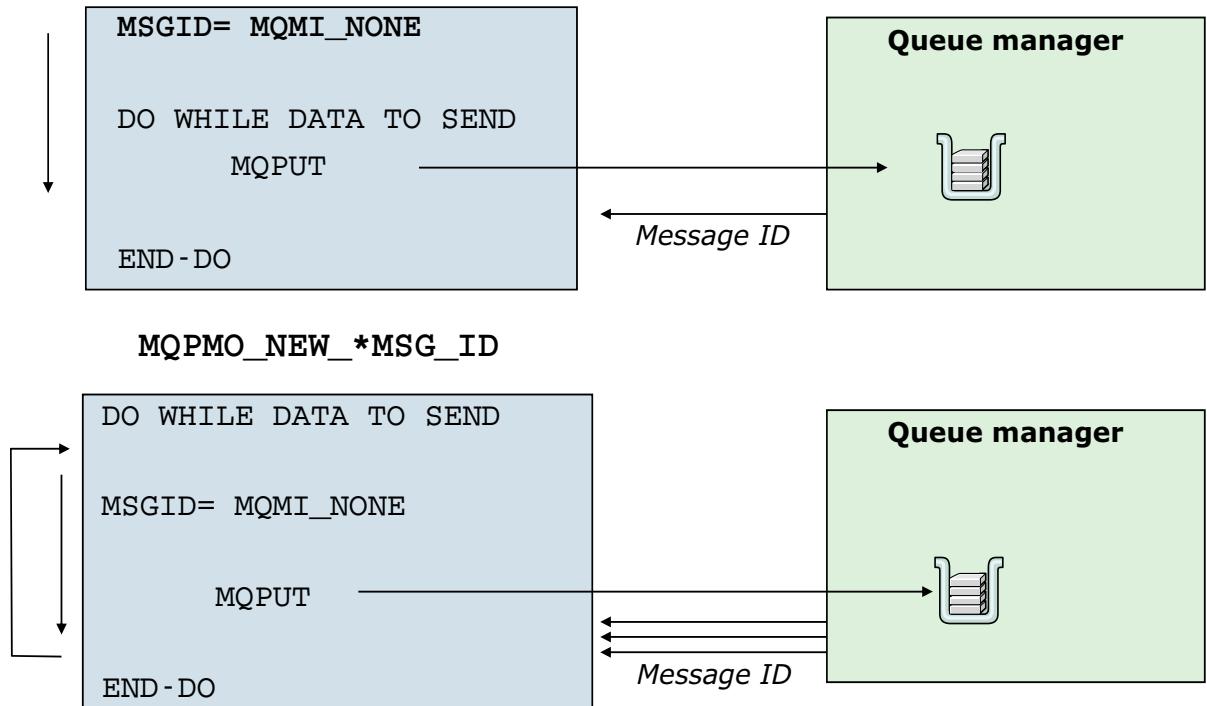
Figure 4-16. MsgId, Correlld, and application parallelism

In this example, you might be sending out requests to different applications for information (perhaps bank account balance and a credit check for a loan application). Again, you must be able to collect all the responses that are related to one particular request from the many replies on the reply-to queue. You cannot be sure about the order of the replies either. One time you might get the account balance information ahead of the credit check information. The next time, the replies might be reversed.

By issuing the MQGET with the message ID or correlation ID or both, the correct messages can be retrieved while others are left intact.

The application that issues the MQPUT of the initial message must set up some unique values in either or both fields. A program can put any specified value into either or both fields. The queue manager keeps the value that is in the Correlld field at the time of the MQPUT. The content of the MsgId field at MQPUT time determines whether the queue manager assigns a unique value and places it in the message ID field.

MQPUT and MQPUT1: Message ID and correlation ID



Getting messages and retrieval considerations

© Copyright IBM Corporation 2017

Figure 4-17. MQPUT and MQPUT1: Message ID and correlation ID

If the application specifies MQMI_NONE or MQPMO_NEW_MSG_ID for the MQPUT and MQPUT1 calls, the queue manager generates a unique message identifier 2 when the message is put. The queue manager also places the message identifier in the message descriptor that is sent with the message, and it returns this message identifier in the message descriptor that belongs to the sending application. The application can use this value to record information about particular messages, and to respond to queries from other parts of the application.

A common use of the generated MsgId and CorrelId is to relate PUT messages with their replies, as shown in the previous figures.

The sending application can also specify a value for the message identifier other than MQMI_NONE, which stops the queue manager from generating a unique message identifier. An application that is forwarding a message can use this technique to propagate the message identifier of the original message.

The queue manager does not use this field except to:

- Generate a unique value
- Deliver the value to the application that issues the get request for the message
- Copy the value to the CorrelId field of any report message that it generates about this message (depending on the report options)

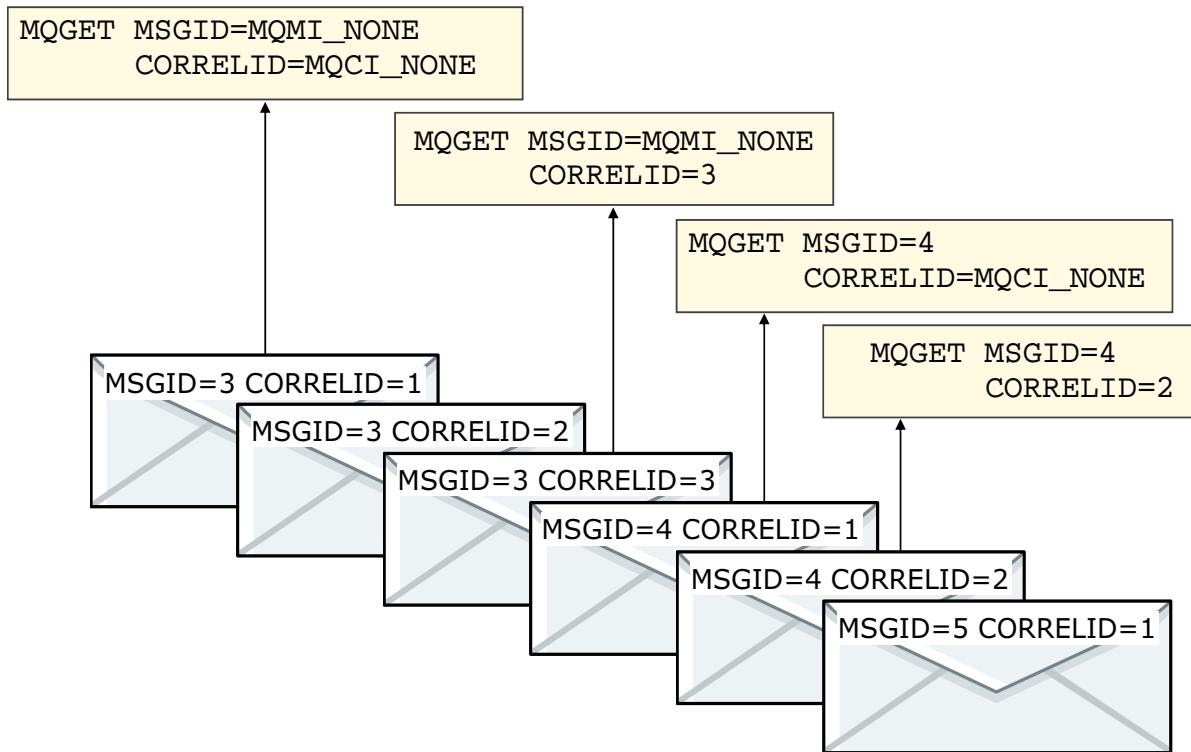
When the queue manager or an MCA generates a report message, it sets the message ID field that is based on the Report field value of the original message, either MQRO_NEW_MSG_ID or MQRO_PASS_MSG_ID. Applications that generate report messages must also use this same technique.

Because the MsgId and CorrelId identifiers are byte fields and not character strings, they are not converted between character sets when the message flows from one queue manager to another.

In the example in the figure, you see the use of symbolic constants to set the message ID and correlation ID to nulls (low values). The first case sets the message ID and correlation ID before going into the MQPUT loop while the second case has the setting of these fields within the loop.

In all implementations, if an application sets the message ID field to nulls (low values), the queue manager then assigns a unique value to the message ID. Any other value that the application assigns results in no action by the queue manager; in this way an application can specify a message ID explicitly.

Retrieval by Message ID and Correlation ID



Getting messages and retrieval considerations

© Copyright IBM Corporation 2017

Figure 4-18. Retrieval by Message ID and Correlation ID

Examine the examples; in each case, fields in the MQMD are being initialized before the MQGET:

1. Asks for the first available message, regardless of the values of message ID and correlation ID.
2. Requests a message that has a correlation ID of three, regardless of the value of the message ID field.
3. Succeeds when it finds a message with message ID of four, regardless of the value in correlation ID.
4. Unless a message with message ID of four and correlation ID of two is found, the call would not succeed.

On completion of the four MQGETs in the example, if no other messages arrived, two messages would remain on the queue. Which ones?

After all MQGETs are complete, the two remaining messages are the second and the last.

Queues with many messages can cause a performance problem when an application chooses to use MQGET with message ID and correlation ID. In general, queues that hold less than approximately a hundred messages do not cause any problems. However, if a queue has thousands of messages, the queue manager scans the queue to find a match.

On z/OS, it is possible for the IBM MQ administrator to define either message ID or correlation ID as an index. For example, if IndexType on a queue is the message ID, an index of all MsgIds on the queue is maintained. Message retrieval by message ID is faster even if many messages are on the queue. Using a match of CorrelId results in slower processing since it is not possible to specify the IndexType of both.

It is possible to specify MQMO_MATCH_MSG_ID and MQMO_MATCH_CORREL_ID in the MatchOptions field and avoid having to reset the message ID and correlation ID. Conversely, not specifying MQMO_MATCH_MSG_ID results in retrieval on the next available message regardless of the value in message ID. The same holds true when the MQMO_MATCH_CORREL_ID is not specified. In other words, if no match options are specified, the message ID and correlation ID field are ignored on the MQGET, essentially behaving as if they are set to MQMI_NONE and MQCI_NONE.

The default on these platforms is MQMO_MATCH_MSG_ID and MQMO_MATCH_CORREL_ID. No program changes are necessary for programs that are moved from earlier releases. If the program was looking for a match, it continues to do so. However, new programs have less to consider. They merely set the MatchOptions to MQMO_NONE unless a match is required.

Retrieving every message

Problem:

Setting the message ID and correlation ID before going into the MQGET loop might get one message only

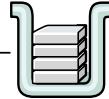
```
MSGID= MQMI_NONE
```

```
CORRELID=MQCI_NONE
```

```
DO WHILE NOT QEMPTY
```

```
    MQGET
```

```
END - DO
```



Solution 1:

Set the message ID and correlation ID in the loop to reset them before every MQGET

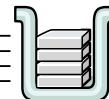
```
DO WHILE NOT QEMPTY
```

```
    MSGID= MQMI_NONE
```

```
    CORRELID=MQCI_NONE
```

```
    MQGET
```

```
END - DO
```



Solution 2:

Specify MQGMO_NONE to retrieve the next available message regardless of the value in message ID and correlation ID

[Getting messages and retrieval considerations](#)

© Copyright IBM Corporation 2017

Figure 4-19. Retrieving every message

The application controls which message it retrieves with MQGET. By setting both message ID and correlation ID in the MQMD to nulls (low values) before the MQGET, the next available message on the queue is retrieved. However, after the MQGET is completed, the values in the message ID and correlation ID are initialized to the values of the message retrieved. If not continuously reset to nulls (low values), the application might receive a failing completion code with the reason set to MQRC_NO_MSG_AVAILABLE even though messages remain on the queue. Using the MQMI_NONE and MQCI_NONE symbolics is the same as if the fields are set by moving nulls or low values to them.

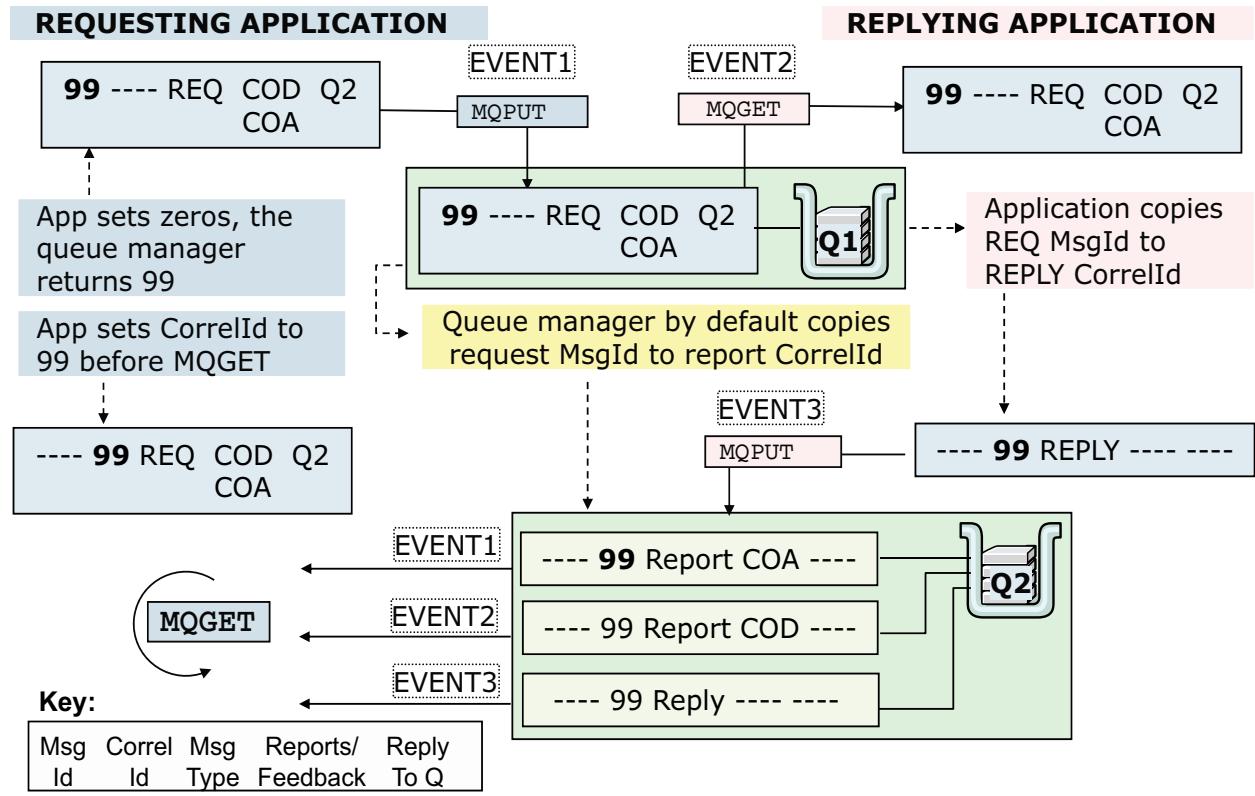
In the example, you see the use of symbolic constants to set message ID and correlation ID to nulls (low values). Notice that the first case sets the message ID and correlation ID before going into the MQGET loop while the second case has the setting of these fields within the loop.

An extra field available in the version 2 MQGMO structure is called MatchOptions. It is possible to specify MQMO_NONE and avoid having to reset the message ID and correlation ID, which results in retrieval on the next available message regardless of the value in message ID and correlation ID.

IBM Training



MsgId, CorrelId, reports, and replies



Getting messages and retrieval considerations

© Copyright IBM Corporation 2017

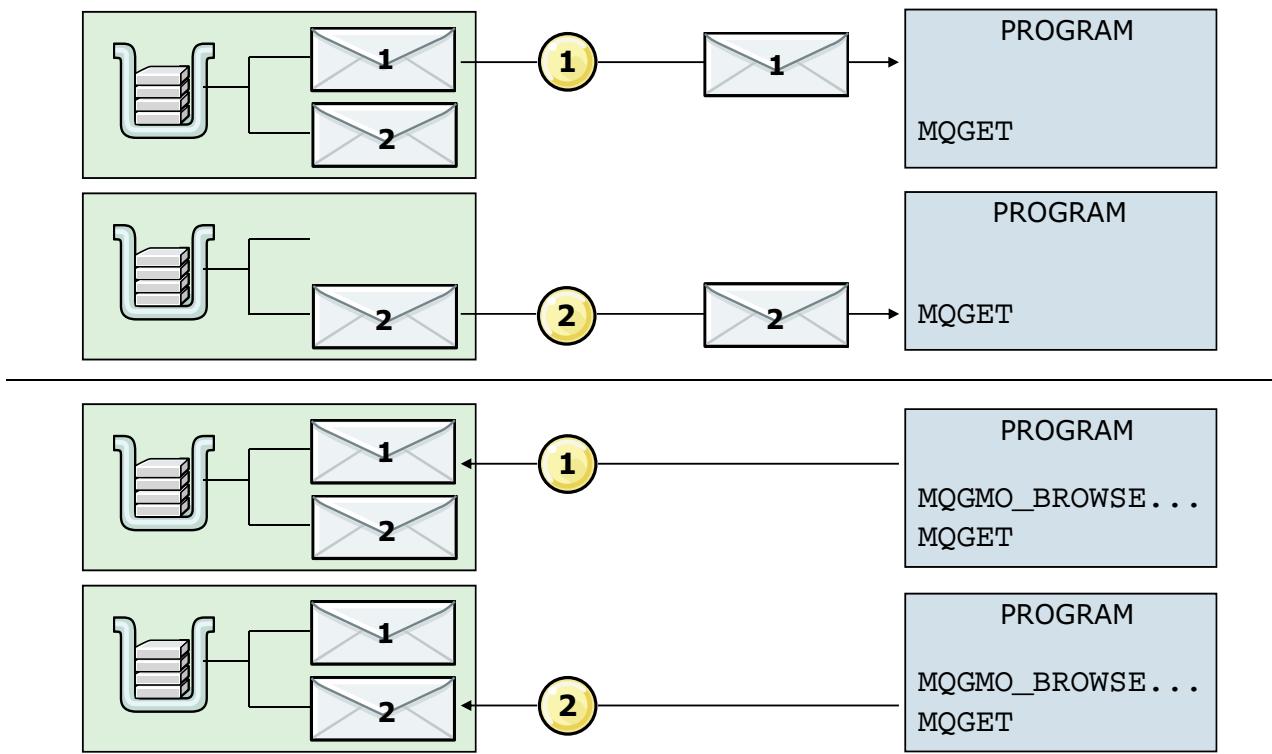
Figure 4-20. MsgId, CorrelId, reports, and replies

This figure relates the concepts of message ID, correlation ID, reports, replies, and selective MQGETs.

1. The requesting application creates an MQMD, setting the MsgType to request(REQ), the reply-to queue to Q2, the message ID to MQMI_NONE, and asking for the target queue manager to generate COA and COD reports.
2. Responding to MQMI_NONE in the MQMD messageId field, the queue manager creates a unique message ID for the new message and returns this value to the program in the MQMD. The message ID is represented as 99 in this example.
3. When the message is MQPUT to Q1, the queue manager creates a COA report message as requested, sending it to the reply-to queue Q2. By default, the request message ID is copied to the report correlation ID.
4. The replying application MQGET causes the queue manager to generate another report message, this time with a feedback value of COD. The CorrelId is constructed in the same way as for the COA report.
5. By convention, when the replying application creates its reply message, it emulates the action of the queue manager in copying the incoming message ID to the outgoing correlation ID.
6. Three messages are on Q2:

- Two reports that the reply queue manager of the application generates
 - Reply that the responding application generates
7. To retrieve only the replies and reports that relate to request 99, the requesting application sets the MQMD message ID field to MQMI_NONE and the MQMD correlation ID field to 99.
 8. Subsequent MQGETs retrieve only relevant messages. The requesting application can distinguish between reports and replies by inspecting the MsgType and the type of report by inspecting the feedback field.

Browsing messages



Getting messages and retrieval considerations

© Copyright IBM Corporation 2017

Figure 4-21. Browsing messages

If an application wants to look at a message without removing it from the queue, the MQGMO_Options field can include either MQGMO_BROWSE_FIRST or MQGMO_BROWSE_NEXT. The browse is unidirectional. It is not possible to browse backward on a queue.

To browse, a queue must be opened with the MQOO_BROWSE open option. Then, when the application uses the MQGMO_BROWSE_ option, instead of the destructive removal of messages, the application moves through the messages in the queue. The application then retrieves a copy of the next available message into its buffer.

When the open option for browsing is used, the queue manager creates a unique pointer that is associated with the object handle. This pointer is called a browse cursor. It is the browse cursor that monitors which message is being pointed to by the application. The browse cursor is not something that the program can manipulate except by issuing MQGET with one of the MQGMO_BROWSE_ options.

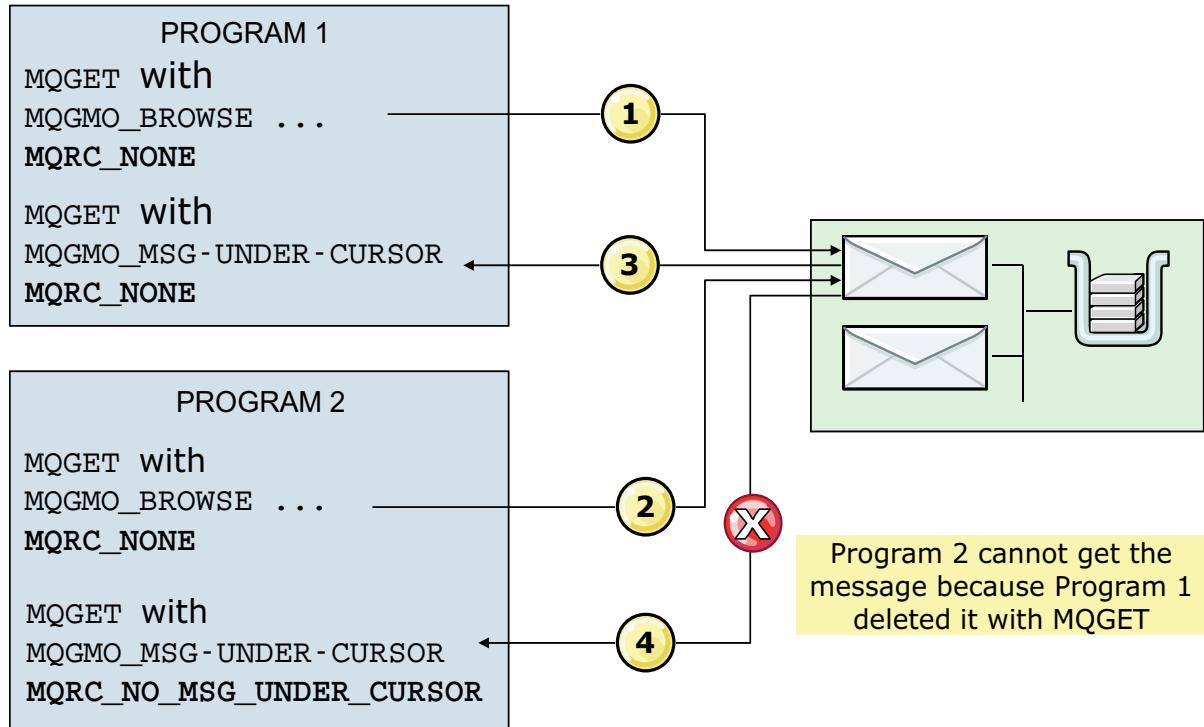
It is possible to combine the MQGMO_WAIT option with the browse. If planning to use this option, it is advised that you specify a wait interval.

IBM MQ browsing is unidirectional. An application can be browsing messages on a queue with a message delivery sequence of priority and reach the end of available messages (MQRC_NO_MSG_AVAILABLE) while new messages of a higher priority are placed on the queue.

In this situation, issue an MQGET with the MQGMO_BROWSE_FIRST option to reposition your browse cursor at the start of the queue.

When a message is found that the application wants to remove from the queue, a further MQGET must be issued with the MQGMO_MSG_UNDER_CURSOR option. This option results in a destructive removal of the message pointed to by the browse cursor.

Browsing the same message



Getting messages and retrieval considerations

© Copyright IBM Corporation 2017

Figure 4-22. Browsing the same message

The numbers in the figure represent a sequence of events.

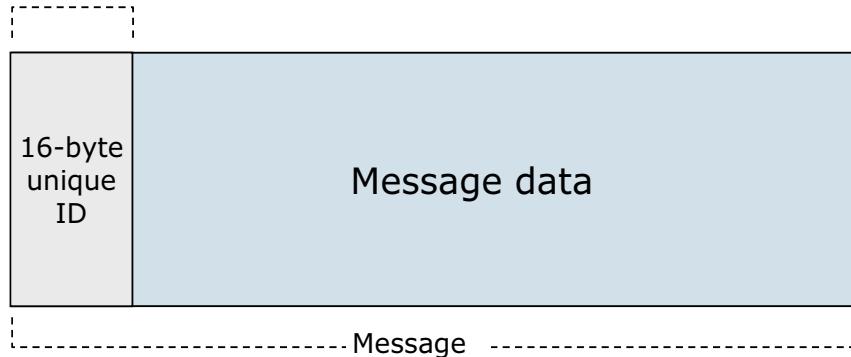
If an application combined the `MQOO_BROWSE` and `MQOO_INPUT_SHARED` options when the application was opened, it is possible that other programs might also be browsing and removing messages from the queue.

In fact, two applications might browse the same message at the same time. Either one might then decide to use the `MQGMO_MSG_UNDER_CURSOR` to remove the message. If the second program then attempts the same, that call fails with `MQRC_NO_MSG_UNDER_CURSOR`. In effect, the cursor is pointing to an empty slot in the queue.

This problem can be avoided by using another option that can be combined with the `MQGMO_BROWSE` and the `MQGMO_LOCK`. This option makes the browsed message invisible and unavailable to any other application that might also be browsing or getting messages from the queue. The `MQGMO_UNLOCK` provides for an explicit unlock of a previously locked message (must use the same handle). The message is also unlocked by a successful `MQCLOSE` that uses the same handle.

Message tokens

Message token



- Uniquely identifies a message on a queue
- Generated by the queue manager when a message is placed on a queue
- Returned after an MQGET in the get message options (GMO) data structure in field MsgToken
- Remains with the message until it is permanently removed from the queue or the queue manager is restarted

[Getting messages and retrieval considerations](#)

© Copyright IBM Corporation 2017

Figure 4-23. Message tokens

The message token is a 16-byte unique identification that the queue manager generates when a message is placed on a queue.

It does not follow the message when transferring from one queue to another queue.

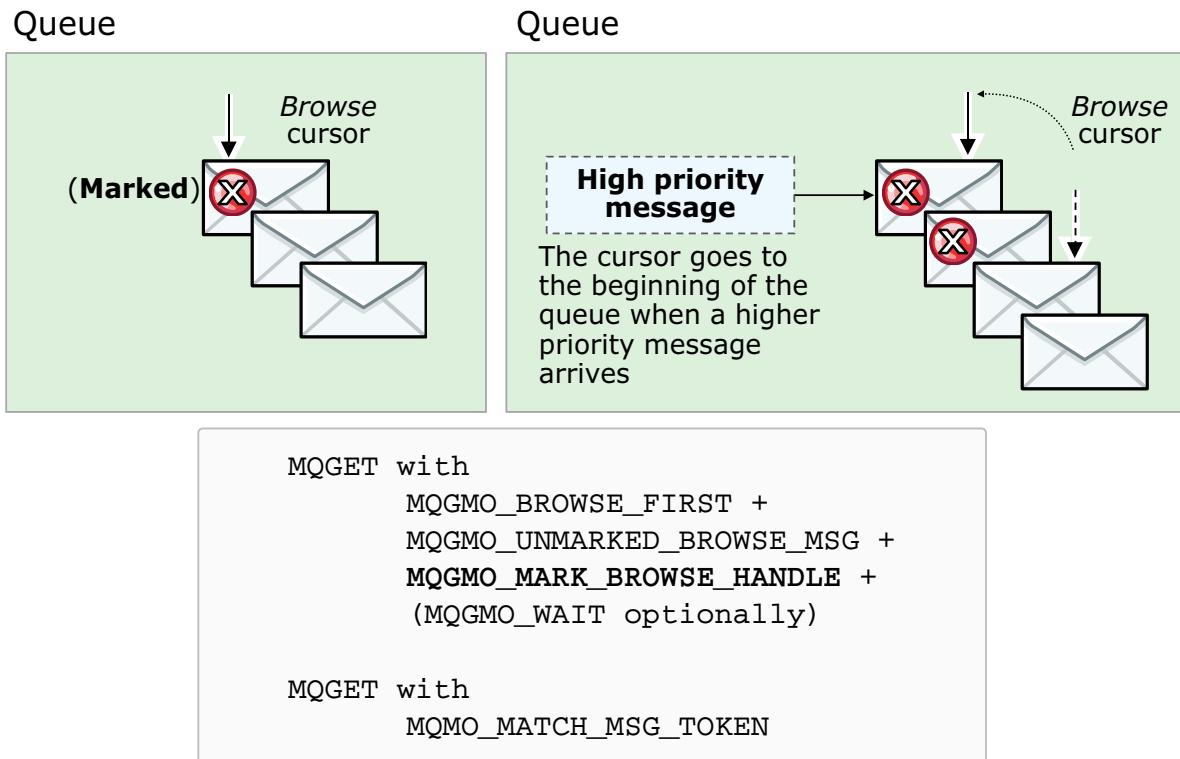
It is returned after the MQGET function call in the get message options (MQGMO) data structure in field MsgToken.

The message token uniquely identifies a message on a queue.

The message token remains with the message until it is permanently removed from the queue or the queue manager is restarted.

It can be used as a match option, MQMO_MATCH_MSG_TOKEN for MQGET.

Browse and mark



Getting messages and retrieval considerations

© Copyright IBM Corporation 2017

Figure 4-24. Browse and mark

Browse and mark are options in the MQGET function call that enable the queue manager to track which messages were browsed and who browsed the messages.

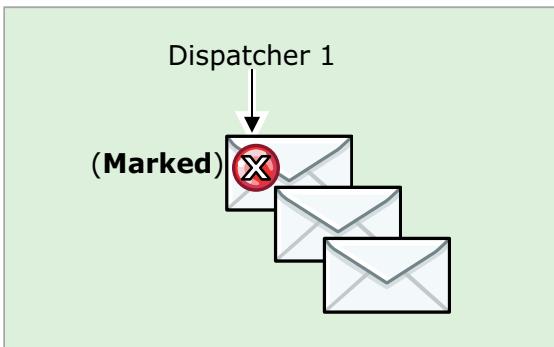
The browse and mark options resolve the issue of skipping messages during a browse loop when high priority messages arrive and the browse cursor is on a lower priority message.

The queue manager returns high priority messages as soon as they arrive to the queue because they can be identified as not marked.

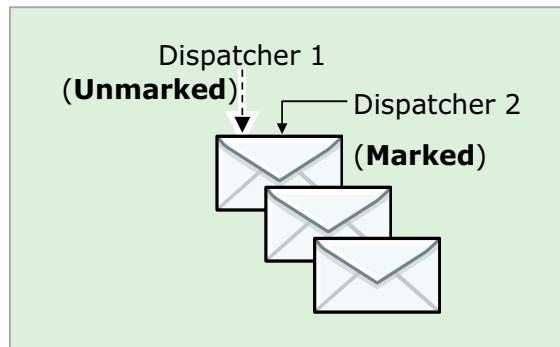
`MQGMO_MARK_BROWSE_HANDLE` marks a message token as browsed by a single application.

Cooperative applications

Queue



Queue



MQGET with
MQGMO_BROWSE_FIRST +
MQGMO_UNMARKED_BROWSE_MSG +
MQGMO_MARK_BROWSE_CO_OP +
(MQGMO_WAIT optionally)

MQGET with
MQMO_MATCH_MSG_TOKEN
MQGMO_UNMARK_BROWSE_CO_OP

Getting messages and retrieval considerations

© Copyright IBM Corporation 2017

Figure 4-25. Cooperative applications

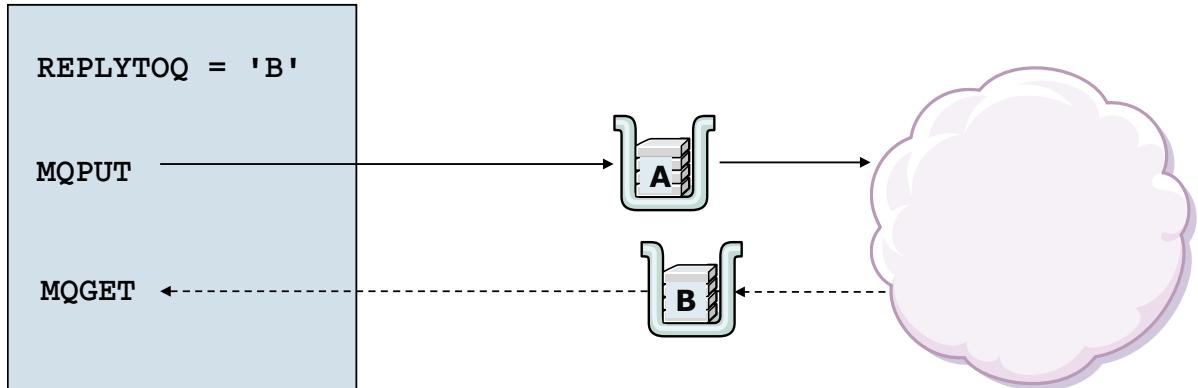
Cooperative dispatchers are programs that browse the same queue and provide the same dispatching service in parallel; therefore, they are not interested in messages that other cooperative dispatchers browse on the same queue.

You might run multiple copies of a dispatcher application to browse messages, then the dispatcher initializes a consumer and passes the message token of the selected message.

MQOPEN option MQOO_COOP indicates that the program is a cooperative browser for the queue that is being opened.

MQGMO_MARK_BROWSE_CO_OP marks a message as browsed by one of a cooperating set of applications.

WAIT



`MQRC_NO_MSG_AVAILABLE` is returned if no eligible message exists on the queue at the time that the `MQGET` is issued

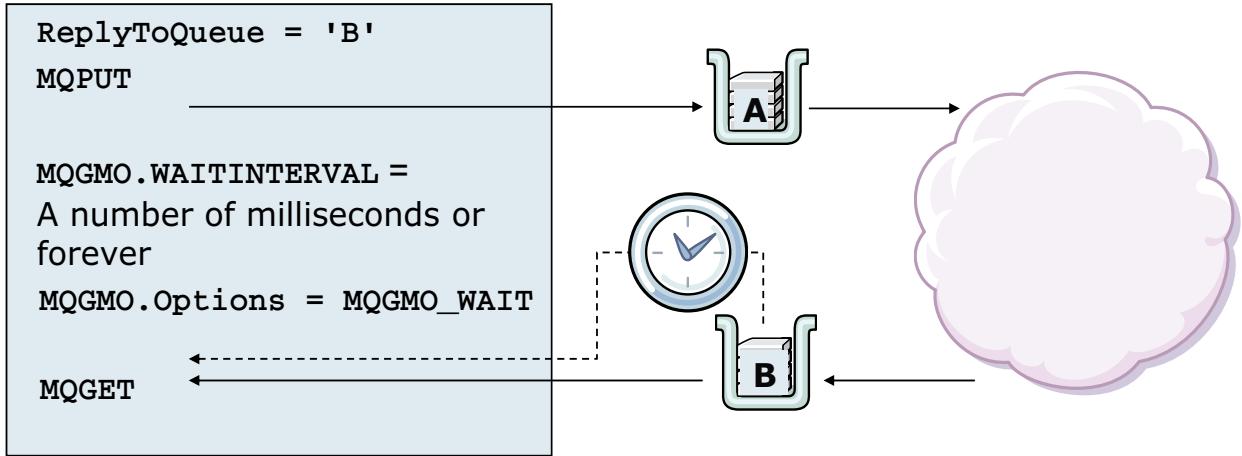
Figure 4-26. WAIT

Because IBM MQ applications frequently communicate across a network or await a response from a triggered program, a reply is not invariably available immediately after the request message is put. In these situations if the application is synchronous, the `MQGMO_WAIT` option can be useful.

If an application uses `MQGMO_WAIT`, the `MQGMO_FAIL_IF QUIESCING` option should be used too. This combination allows the call to be ended if the queue manager is quiescing, returning `MQRC_Q_MGR QUIESCING`.

If the ability of the queue to deliver messages changes, the wait can also end. For instance, if the IBM MQ administrator changes the get attribute for the queue from allowed to inhibited, the call would then fail with `MQRC_GET_INHIBITED`.

WAIT with WaitInterval



[Getting messages and retrieval considerations](#)

© Copyright IBM Corporation 2017

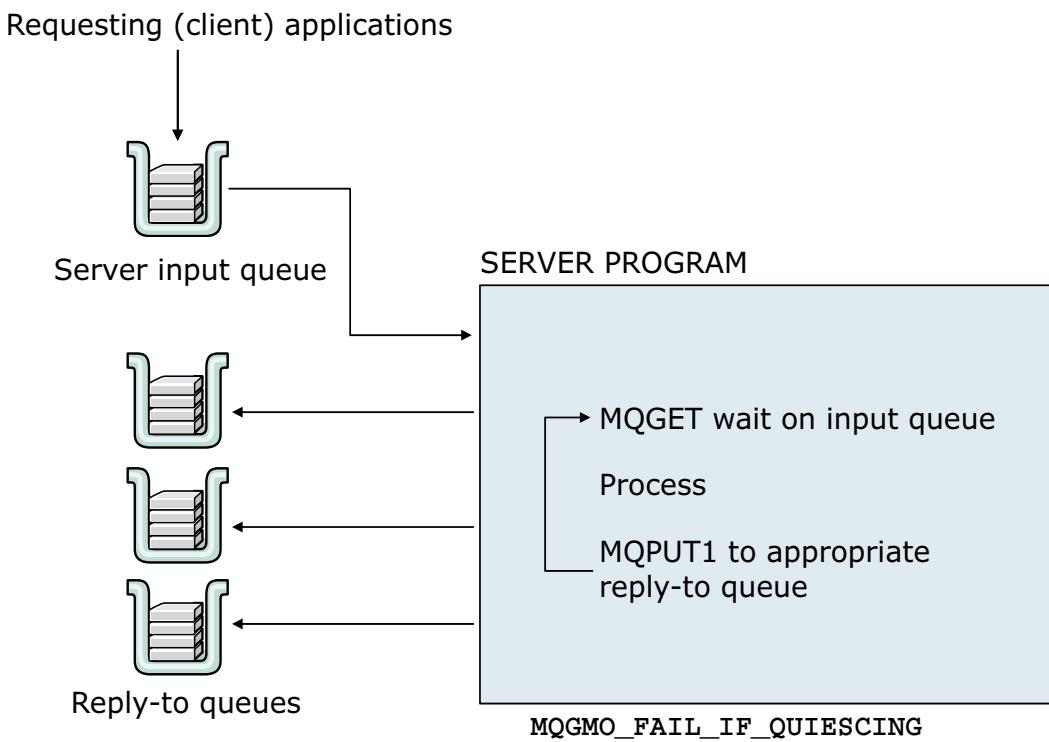
Figure 4-27. WAIT with WaitInterval

Within the get message option structure, a field called `WaitInterval` is used to set a time period (in milliseconds) during which the wait is in effect. If a message is available when the `MQGET` is issued, the wait never takes effect. If not, the `MQGMO_WAIT` option (along with a value in the `WaitInterval` field) establishes a reasonable time to wait for a message. If none arrives during that time, the `MQGET` completes with a failing completion code (`MQCC_FAILED`) and a reason of `MQRC_NO_MSG_AVAILABLE`. During the `WaitInterval`, if a message arrives, the wait is immediately satisfied.

The default wait interval is `MQWI_UNLIMITED`, which means forever. This option should be used only with extreme discretion, and always with `MQGMO_FAIL_IF_QUIESCING`.

If you are using a `WaitInterval`, consider a similar expiry value.

Typical server application



Getting messages and retrieval considerations

© Copyright IBM Corporation 2017

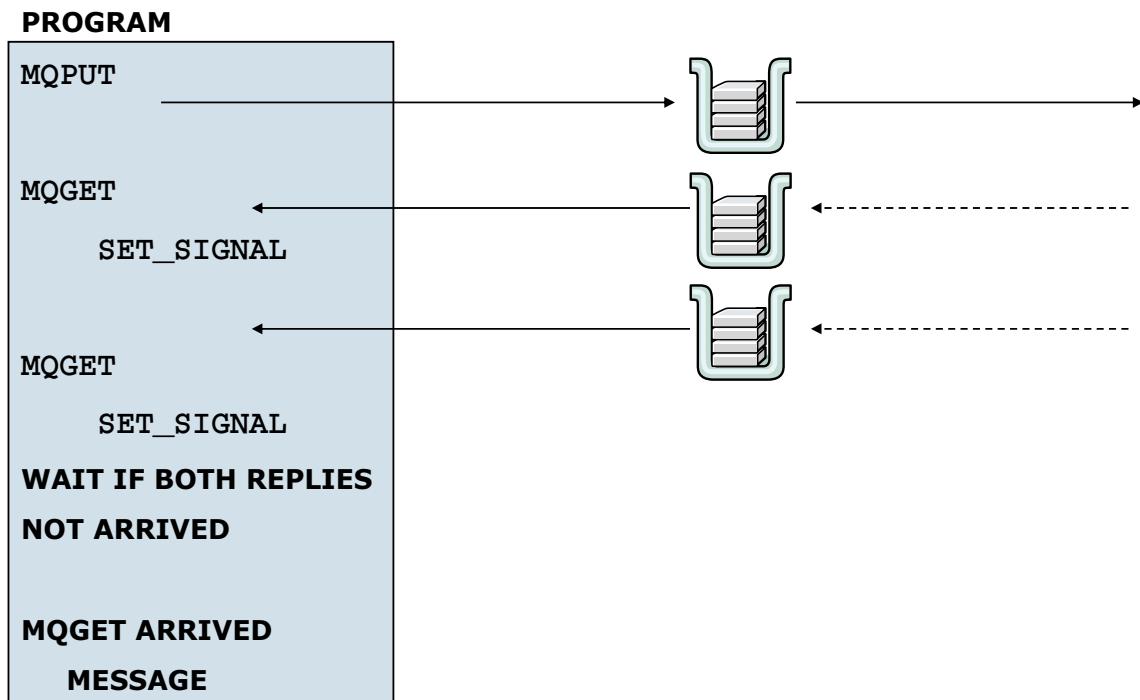
Figure 4-28. Typical server application

An application might have a loop that includes the MQGET with the MQGMO_WAIT option. This operation might be typical for a server type program; it just waits for more work to come along. As one final reminder, use a WaitInterval and MQGMO_FAIL_IF_QUIESCING so that the MQGET can somehow be ended if the queue manager is ending.

An example is an z/OS IMS environment. An IMS system cannot be ended while any of its dependent regions are active. An IMS application that is waiting for a message is considered active by the IMS region controller. If it was not possible to interrupt the wait, the IMS system cannot be ended normally.

Use the WaitInterval or at least the MQGMO_FAIL_IF_QUIESCING. Otherwise, the application can keep the system from being shut down in some cases.

MQGET with SET SIGNAL option (z/OS only)



Getting messages and retrieval considerations

© Copyright IBM Corporation 2017

Figure 4-29. MQGET with SET SIGNAL option (z/OS only)

Unit summary

- Describe the parameters that are required for the MQGET call
- Describe the MQGET call option groupings
- Explain how to associate requests with responses by using the message and correlation IDs
- Differentiate between the options that are used to browse messages
- Explain how to use message tokens to browse a queue
- Explain the use and need for message marks and cooperative browsing
- Describe how to write code that waits for responses

Getting messages and retrieval considerations

© Copyright IBM Corporation 2017

Figure 4-30. Unit summary

Review questions

1. Select the correct answers. What two options are used by an application to code an MQGET with a wait?
 - a. MQGMO_WAIT_FOR_ALL_MESSAGES
 - b. MQGMO_WAIT
 - c. MQGMO_ALL_MSGS_AVAILABLE
 - d. MQGMO_WAIT_INTERVAL
2. What statements are true of cooperative browse?
 - a. Cooperative applications optimize the workload by deleting browsed messages.
 - b. They use MQOO_COOP as the MQOPEN option.
 - c. They can use the MQGMO_MARK_BROWSE_CO_OP option to mark a message as already browsed.
 - d. They improve efficiency by alerting another cooperating application that a specific message has already been browsed.



Getting messages and retrieval considerations

© Copyright IBM Corporation 2017

Figure 4-31. Review questions

Write your answers here:

1.

2.

Review answers



1. Select the correct answers. What two options are used by an application to code an MQGET with a wait?
 - a. MQGMO_WAIT_FOR_ALL_MESSAGES
 - b. MQGMO_WAIT
 - c. MQGMO_ALL_MSGS_AVAILABLE
 - d. MQGMO_WAIT_INTERVAL

The answer is: b and d. a is not a valid option. d is used for message groups.

2. What statements are true of cooperative browse?
 - a. Cooperative applications optimize the workload by deleting browsed messages.
 - b. They use MQOO_COOP as the MQOPEN option.
 - c. They can use the MQGMO_MARK_BROWSE_CO_OP option to mark a message as already browsed.
 - d. They improve efficiency by alerting another cooperating application that a specific message has already been browsed.

The answer is: b, c, and d. Browsed messages are not deleted unless the expiry is set.

Exercise: Correlating requests to replies

Getting messages and retrieval considerations

© Copyright IBM Corporation 2017

Figure 4-33. Exercise: Correlating requests to replies

Exercise objectives

- Code or modify an application to generate a confirm-on-arrival (COA) Report message that preserves the original message identifier (MsgId)
- Code or modify an application to reply to a request message by setting the correlation identifier of the reply message to the message identifier of the request message
- Use a formatted message descriptor display to check the correct setting of your message and correlation identifier fields
- Alter or code an application to get a reply message from a queue with a correlation identifier that matches the message identifier of its corresponding request message



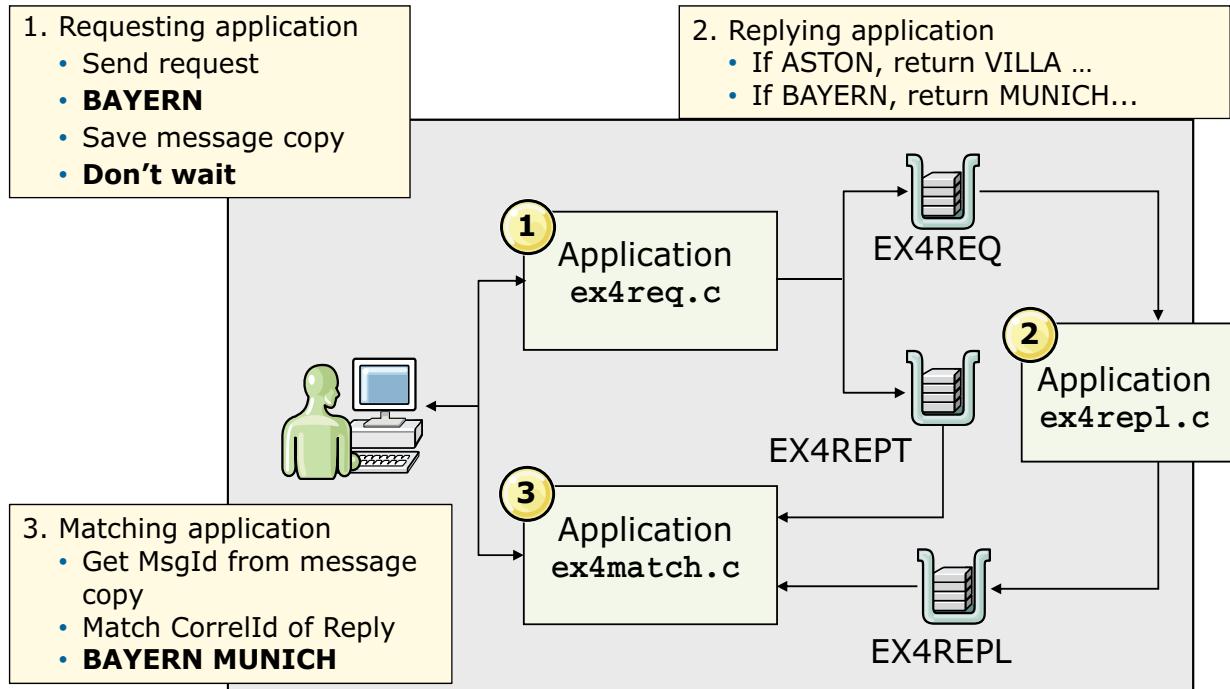
[Getting messages and retrieval considerations](#)

© Copyright IBM Corporation 2017

Figure 4-34. Exercise objectives

Correlating requests to replies

- Ticketing request is missing half the name of the football team
- Customer care representative finds correct name



Getting messages and retrieval considerations

© Copyright IBM Corporation 2017

Figure 4-35. Correlating requests to replies

Unit 5. Data conversion

Estimated time

00:30

Overview

This unit describes considerations to observe when data needs to be converted due to its exchange across different platforms.

How you will check your progress

Accountability:

- Review questions

References

IBM Knowledge Center for IBM MQ V9 documentation

Unit objectives

- Describe the need for data conversion
- Identify key MQMD data conversion fields
- Differentiate how IBM MQ and message data are converted
- Describe the various cases for message data conversion
- Explain how to create a data conversion exit
- Describe considerations to observe in the original MQPUT
- Explain default data conversion
- Identify the case when the sender handles conversion

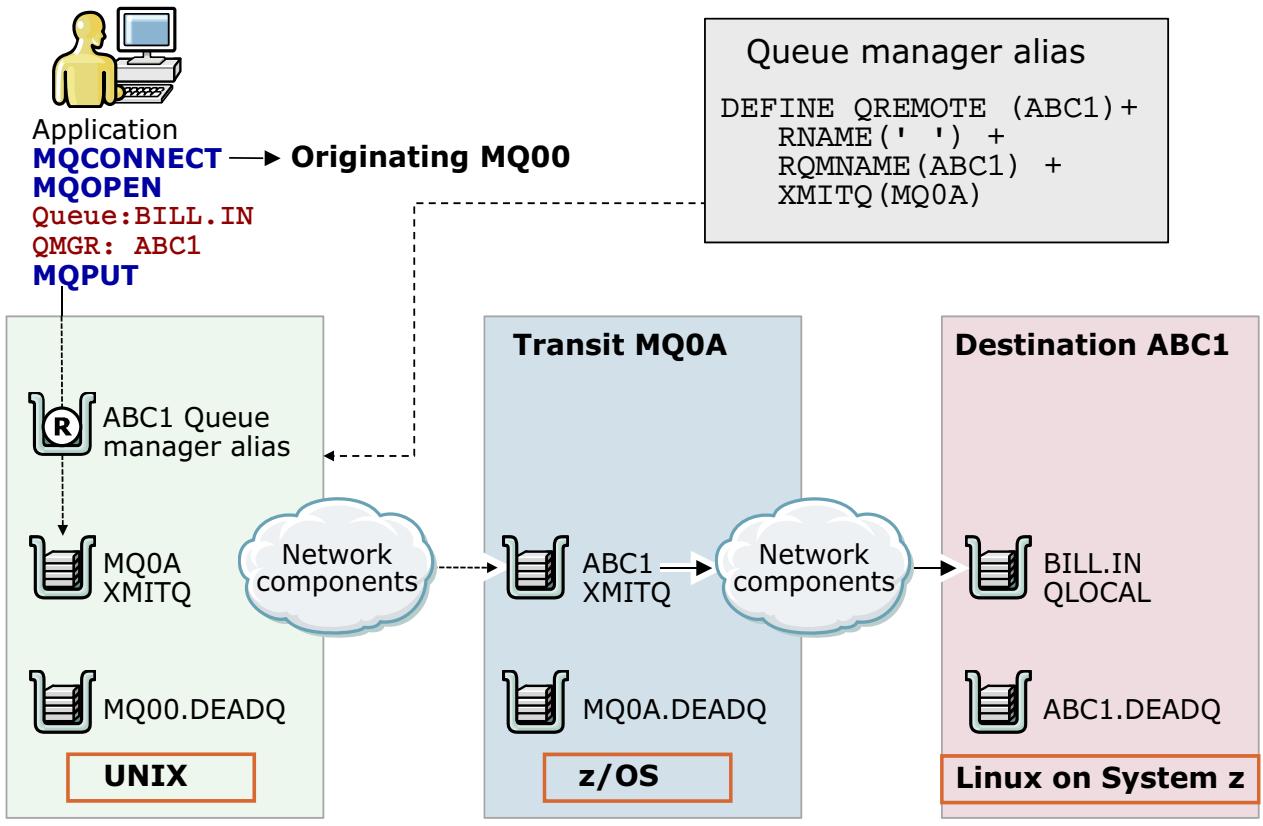
Data conversion

© Copyright IBM Corporation 2017

Figure 5-1. Unit objectives



Data conversion: A case for “receiver makes good”



Data conversion

© Copyright IBM Corporation 2017

Figure 5-2. Data conversion: A case for “receiver makes good”

A key attribute of IBM MQ is the ability to exchange messages across operating systems that use different ways to represent data. Whether you exchange information across “like” systems, or “unlike” systems, data conversion routines are started to determine whether conversion is needed.

This unit covers details about data conversion. It is introduced with the old IBM MQ adage “receiver makes good”, which means that under most circumstances, the application that receives the message should do the conversion. If you review the diagram, the reason for this adage becomes obvious.

It is possible that an IBM MQ message might traverse other queue managers en route to its final destination. Why convert several times from each forwarding queue manager, when only one conversion is needed at the intended receiver?

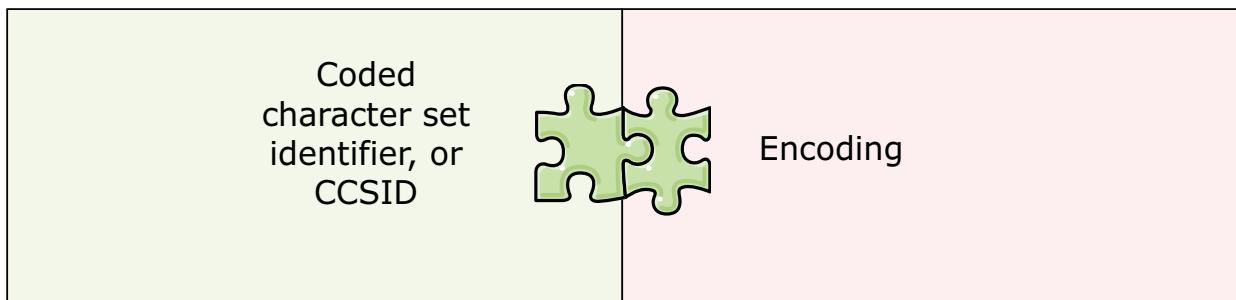
You learn about what gets converted and when in this unit

Two components of data conversion

```

StrucId : 'MD' Version : 2
Report : 0 MsgType : 8
Expiry : -1 Feedback : 0
Encoding: 546 CodedCharSetId : 1208
Format : 'MQSTR'

```



[Data conversion](#)

© Copyright IBM Corporation 2017

Figure 5-3. Two components of data conversion

Earlier in this course, you reviewed the MQMD structure, and maybe you noticed the Encoding and CodedCharSetId fields. Encoding and CodedCharSetId are the key components of data conversion in IBM MQ.

The coded character set identifier (CCSID), which is also referred to as a code page, is a number that is assigned to the way data is represented.

Encoding is generally taken to mean the method that this platform uses to represent numeric data.

Coded character set identifier and terminology

- **Character set:** Defined set of characters that are used to represent text information, such as 0 – 9, A – Z, a – z, punctuation
- **Code page (code set):** A table that maps alphanumeric code and its binary representation
- **CCSID, or Coded character set ID:** Unique, 0-5535 number that IBM assigns to identify a coded character set and a code page
- **Code point:** The location of a character within the code page
- IBM Coded Character Set Reference →

1205	04B5	UTF-16
1208	04B8	UTF-8 with IBM PUA
1209	04B9	UTF-8
1210	04BA	UTF-EBCDIC with IBM PUA
1211	04BB	UTF-EBCDIC
1212	04BC	SCSU with IBM PUA
1213	04BD	SCSU

<https://www.ibm.com/software/globalization/g11n-res.html>

Figure 5-4. Coded character set identifier and terminology

Code pages assign data (character or non-character) to hexadecimal values. These hexadecimal values are in the range of:

- 00 to FF
- For alphabets that cannot be represented as 256 positions, 0000 to FFFF in double-byte character set (DBCS) code pages

This slide shows some of the terminology that is associated with a code page, or coded character set identifiers.

IBM assigns a unique CCSID to identify a coded character set and code page. When this course was written, the IBM Coded Character Set reference web page was found at the link provided in the display. However, if this page moves to a different link, you can search for “IBM Coded Character Set Reference” to locate the changed link.

Encoding

BigEndian 0825	LittleEndian
<pre>***** Message ***** length - 181 of 181 bytes 00000000: 444C 4820 0100 0000 0000 2508 0000 4E4F 542E DLH%....NOT. ' 00000010: 5448 4552 4520 2020 2020 2020 2020 THERE '</pre>	

Operating System	Memory Address	Hex Value	Endianness
IBM i	273	X'00000111'	BigEndian
Linux (Intel)	546	X'00000222'	LittleEndian
Linux on SPARC	273	X'00000111'	BigEndian
Linux on x86	546	X'00000222'	LittleEndian
Solaris on SPARC	273	X'00000111'	BigEndian
UNIX systems	273	X'00000111'	BigEndian
Windows	546	X'00000222'	LittleEndian
z/OS	785	X'00000311'	BigEndian

Data conversion

© Copyright IBM Corporation 2017

Figure 5-5. Encoding

Encoding, sometimes referred to as “endianness”, represents the way that a specific operating system stores numeric data.

BigEndian operating systems store the most significant byte in the smallest, or lower address. For example, the hex value x'0825' would be stored as '0825'.

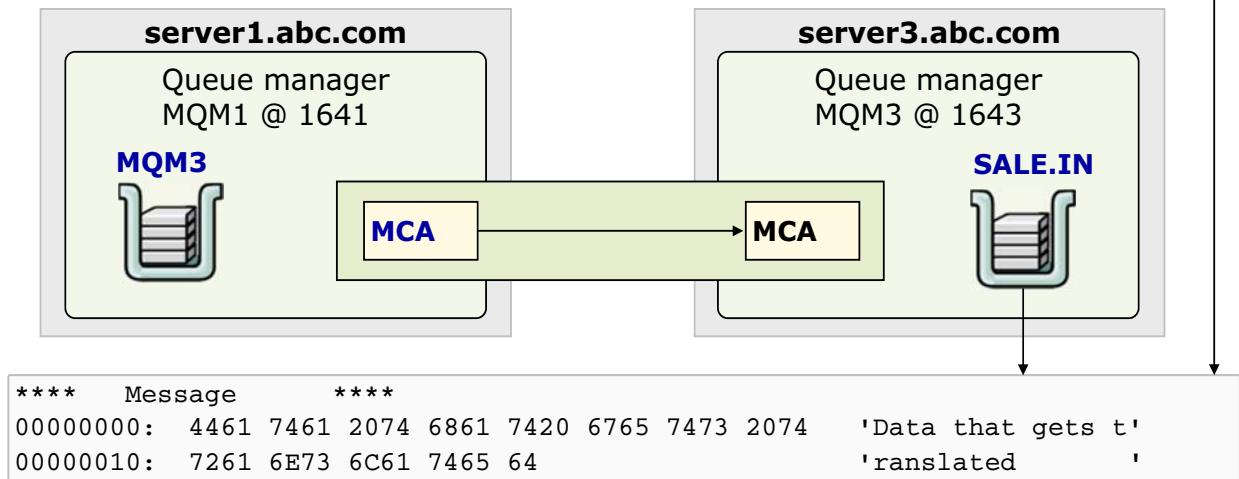
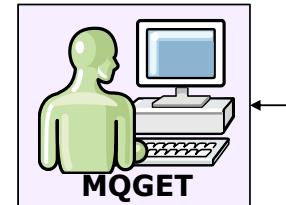
LittleEndian operating systems store the least significant byte in the lower memory location. In this case, x'0825' is stored as '2508'.

The display shows the encoding, or endianness, of some familiar operating systems.

The term “endianness” is said to originate from the book *Gulliver’s Travels*. In the story, people had different ways of eating hardboiled eggs. Some preferred to eat the eggs from the little end first; these were little endians. Others ate the eggs from the big end first; these were big endians. In the story, this “endianness” resulted in several wars.

What gets converted

- **IBM MQ data** is automatically negotiated and queue managers convert it during channel communication:
 - Transmission segment header
 - Message descriptor (MQMD)
- **Message data** might require your action



Data conversion

© Copyright IBM Corporation 2017

Figure 5-6. What gets converted

Data that is converted falls into two categories:

- For IBM MQ use: When channels across two queue managers start, the message channel agents negotiate the CCSID and encoding to be used. The queue managers use the selected CCSID and encoding combination to:
 - Convert the contents of the message descriptor, if necessary
 - Encode a control block exchanged between the connecting queue managers. This block is called a transmission segment header, or TSH
- User data: The user or message data might also need to be converted. Conversion of the user data is addressed later in this unit.

You now look at how the MQMD fields are converted.

IBM MQ data: How MQMD fields are converted (1 of 2)

Field	Description	Offset	Converted as
StrucId	Structure identifier	'00'x	Character
Version	Structure version number	'04'x	Numeric
Report	Options for report messages	'08'x	Numeric
MsgType	Message type	'0C'x	Numeric
Expiry	Message lifetime	'10'x	Numeric
Feedback	Feedback or reason code	'14'x	Numeric
Encoding	Message data numeric encoding	'18'x	Numeric
CodedCharSetId	Message data CCSID	'1C'x	Numeric
Format	Message data format name	'20'x	Character
Priority	Message priority	'28'x	Numeric
Persistence	Message persistence	'2C'x	Numeric
MsgId	Message identifier	'30'x	Byte
CorrelId	Correlation identifier	'48'x	Byte

[Data conversion](#)

© Copyright IBM Corporation 2017

Figure 5-7. IBM MQ data: How MQMD fields are converted (1 of 2)

This slide is the first of two slides that show how MQMD fields are converted. Some fields are converted as character data, some fields are converted as numeric data, and some fields are passed without conversion.

MsgId, CorrelId, Accounting Token, and GroupId are not considered to be character data, and are not converted as character data.

Some applications choose to place character data in these fields. However, depending on the platforms that exchange data, applications might need to take responsibility for the conversion of these fields (or a conversion exit, covered later in this unit). If you allow IBM MQ to generate the content of the MsgId field, it is not entirely character data.

IBM MQ data: How MQMD fields are converted (2 of 2)

Field	Description	Offset	Converted as
BackoutCount	Backout counter	'60'x	Numeric
ReplyToQ	Name of reply queue	'64'x	Character
ReplyToQMGr	Name of reply queue manager	'94'x	Character
UserIdentifier	User identifier	'C4'x	Character
AccountingToken	Accounting token	'D0'x	Byte
ApplIdentityData	Identity-related application data	'F0'x	Character
PutApplType	What appl. type put the message	'110'x	Numeric
PutApplName	Name of appl. putting message	'114'x	Character
PutDate	Date when message was put	'130'x	Character
PutTime	Time when message was put	'138'x	Character
ApplOriginData	Application data related to origin	'140'x	Character
GroupID		'144'x	Byte
Last four fields: MsgSeqNumber, Offset, MsgFlags, OriginalLength, 4 bytes each, all converted as numeric		'15C'x	Numeric

[Data conversion](#)

© Copyright IBM Corporation 2017

Figure 5-8. IBM MQ data: How MQMD fields are converted (2 of 2)

This image is continued from the previous slide.

Message data conversion

- Message contains all character data
 - Set `MQMD.Format` to `MQFMT_STRING`
 - IBM MQ does all the work
- Message contains all numeric data
 - If the platforms where the messages are exchanged are the same, no action is necessary
 - If you are unsure of the platforms where the messages are exchanged now, or for future expansion, use the guidance for mixed character and numeric data
- If the message contains mixed character and numeric data
 - Create a conversion exit
 - Set the `MQMD.Format` in the original `MQPUT` to the format name as designated in the exit

[Data conversion](#)

© Copyright IBM Corporation 2017

Figure 5-9. Message data conversion

When you look at how the user part of the message gets converted, the importance of the design of the user data becomes apparent. The message data gets converted according to its contents.

When your message content is strictly made up of character data, IBM MQ does the conversion. In this case, you set the MQMD format field to `MQFMT_STRING`.

When the message is all numeric:

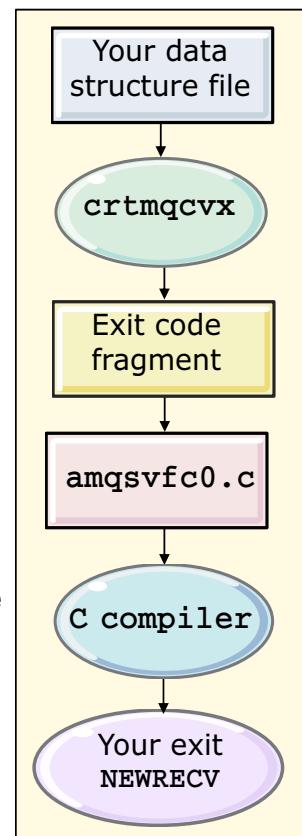
- If both platforms that exchange the messages are the same, no action is necessary.
- If the platforms that exchange the messages are not the same, or if you are unsure of possible future expansion to different platforms, see the convention for mixed data.
- If the message contains mixed numeric and character data, a conversion exit is needed to convert the user part of the message.

A conversion exit should not be intimidating. If you have a header that contains the format of your message data, the exit is simple to generate. When you use the exit, you set the MQMD format of the original `MQPUT` to the given name of the conversion exit.

However, when you do need an exit, you need clearly defined roles as to who maintains the exits in your organization. These exits need to be part of your IBM MQ administrative activities, with designated support resources.

Creating a data conversion exit

- Decide on a name for the exit and format, up to 8 characters, such as `NEWRECV`
- Create a structure to represent your message format
- Run the `crtmqcvx` utility on the C structure to create a **code fragment** for the conversion exit
- Back up and then use the sample conversion exit `amqsvfc0.c` as a base for your exit
 - Include the code fragment that you created from your structure in the `amqsvfc0.c` source by inserting as directed in the exit comments
 - Decide on an **entry point** name for the code; this code is the `MQMD.Format` name that you use in your code
 - Replace the `MQENTRY` and `MQDATACONVEXIT` dummy names with the name selected as your **entry point**
 - Compile your exit (`amqsvfc0.c`) naming the output to match your selected name, for example: `NEWRECV`



Data conversion

© Copyright IBM Corporation 2017

Figure 5-10. Creating a data conversion exit

This slide explains how to create a data conversion exit.

When you use a data conversion exit, remember to use the designated exit name in the `MQMD` format field of the application that does the original `MQPUT` of the message.

Considerations for applications that do original MQPUT

- Set `MQMD.CodedCharSetId` to `MQCCSI_Q_MGR` before each `MQPUT`
- Set `MQMD.Encoding` to `MQENC_NATIVE`

```

MQCONN
MQOPEN
sameMQMD.CodedCharSetID = MQCCSI_Q_MGR
Do until
{
    MQPUT - originating queue manager
    sameMQMD.CodedCharSetID
    MQGET now sets sameMQMD.CodedCharSetID
        to remote queue managers CCSID)
}
MQCLOSE
MQDISC

```



[Data conversion](#)

© Copyright IBM Corporation 2017

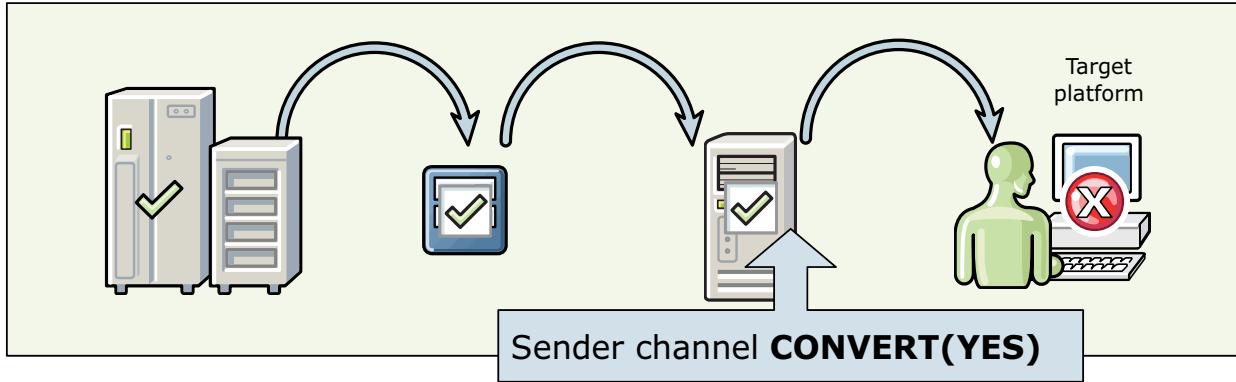
Figure 5-11. Considerations for applications that do original MQPUT

Although the suggested practice is “receiver makes good”, that is, the application that does the `MQGET` does the conversion, an exception exists. This exception is when the target application does not support data conversion.

When the target application does not support data conversion, the conversion should be done in the `SENDER` channel of the queue manager that precedes the target application. If a conversion exit is needed, it should be provided to the queue manager that needs to do the conversion.

Exception case when you convert on the sender side

- Convert on the sending side when receiving (target) platform does not support data conversion



- Conversion is done in the sender channel immediately preceding the target platform
- If a conversion exit is required, it must be provided to the queue manager with the CONVERT(YES) in the sender channel

Data conversion

© Copyright IBM Corporation 2017

Figure 5-12. Exception case when you convert on the sender side

Unit summary

- Describe the need for data conversion
- Identify key MQMD data conversion fields
- Differentiate how IBM MQ and message data are converted
- Describe the various cases for message data conversion
- Explain how to create a data conversion exit
- Describe considerations to observe in the original MQPUT
- Explain default data conversion
- Identify the case when the sender handles conversion

Data conversion

© Copyright IBM Corporation 2017

Figure 5-13. Unit summary

Review questions (1 of 2)



1. What is meant by “receiver makes good” in IBM MQ?
 - a. The receiving application should correct any application errors
 - b. The IBM MQ administrator in the target queue manager should resolve any channel-related conversion problems
 - c. The receiving application should handle data conversion in the MQGET
 - d. The receiving application must set MQMD.Format to MQMFT_NONE
2. True or False: A CCSID is a unique number that IBM assigns to identify a coded character set and a code page.
3. Select all correct answers. What parts of a message does IBM MQ automatically convert without added work by a developer?
 - a. Transmission segment header
 - b. Message descriptor (MQMD)
 - c. Mixed character and numeric data
 - d. Character data

Data conversion

© Copyright IBM Corporation 2017

Figure 5-14. Review questions (1 of 2)

Write your answers here:

- 1.
- 2.
- 3.

Review questions (2 of 2)

4. You need to code a conversion exit. What is the name of the utility that generates code fragments from the data structure you create?
 - a. amqsvfc0.c
 - b. crtmqcvx
 - c. runmqsc
 - d. MQMD.Format
5. True or False: The commonly accepted case for data conversion on the sender side is when the target platform does not support data conversion.



Data conversion

© Copyright IBM Corporation 2017

Figure 5-15. Review questions (2 of 2)

Write your answers here:

4.

5.

Review answers (1 of 2)

1. What is meant by “receiver makes good” in IBM MQ?
 - a. The receiving application should correct any application errors
 - b. The IBM MQ administrator in the target queue manager should resolve any channel-related conversion problems
 - c. The receiving application should handle data conversion in the MQGET
 - d. The receiving application must set MQMD.Format to MQMFT_NONE

The answer is: c. The developer should use the MQGMO_CONVERT option in the MQGET.
2. True or False: A CCSID is a unique number that IBM assigns to identify a coded character set and a code page.
 The answer is: True.
3. Select all correct answers. What parts of a message does IBM MQ automatically convert without added work by a developer?
 - a. Transmission segment header
 - b. Message descriptor (MQMD)
 - c. Mixed character and numeric data
 - d. Message data character data

The answer is: a, b, and d (d: If format is set to MQFMT_STRING)

Data conversion

© Copyright IBM Corporation 2017

Figure 5-16. Review answers (1 of 2)

Review answers (2 of 2)



4. You need to code a conversion exit. What is the name of the utility that generates code fragments from the data structure you create?
- a. `amqsvfc0.c`
 - b. `crtmqcvx`
 - c. `runmqsc`
 - d. `MQMD.Format`

The answer is: b. The utility for UNIX and Windows is crtmqcvx. On iSeries, it is CVTMQMDTA.

5. True or False: The commonly accepted case for data conversion on the sender side is when the target platform does not support data conversion.

The answer is: True. If data conversion is not supported on the target platform, the sender channel needs to handle the conversion task.

Unit 6. Bind and Message groups

Estimated time

00:30

Overview

IBM MQ architects strive to design applications that are conducive to a highly available infrastructure. A key consideration in application design is avoidance of queue manager affinities. However, when queue manager affinities cannot be avoided, applications might need to use bind options and message groupings to accomplish the task. This unit shows you how to use bind options in clustered environments, and how to develop applications that need to produce or consume a group of messages in a specific order.

How you will check your progress

Accountability:

- Review questions

References

IBM Knowledge Center for IBM MQ V9 documentation

Unit objectives

- Explain the importance of limiting applications that introduce queue manager affinities in the IBM MQ architecture
- Describe the use of bind-related attributes in queue definitions and MQOPEN options
- Describe how to write IBM MQ applications that require a distinct sequence of messages to complete processing

Bind and Message groups

© Copyright IBM Corporation 2017

Figure 6-1. Unit objectives

IBM MQ objectives for highly available architectures

Objectives	IBM MQ cluster	Failover solution
Immediate Availability for new requests		
Access to marooned messages		

[Bind and Message groups](#)

© Copyright IBM Corporation 2017

Figure 6-2. IBM MQ objectives for highly available architectures

When an IBM MQ infrastructure is planned, it includes considerations for reliability and scalability. Scalability involves the growth and volume of messages that are expected for an application. Reliability involves the resilience of the queue manager; that is, if the queue manager fails, is there a failover?

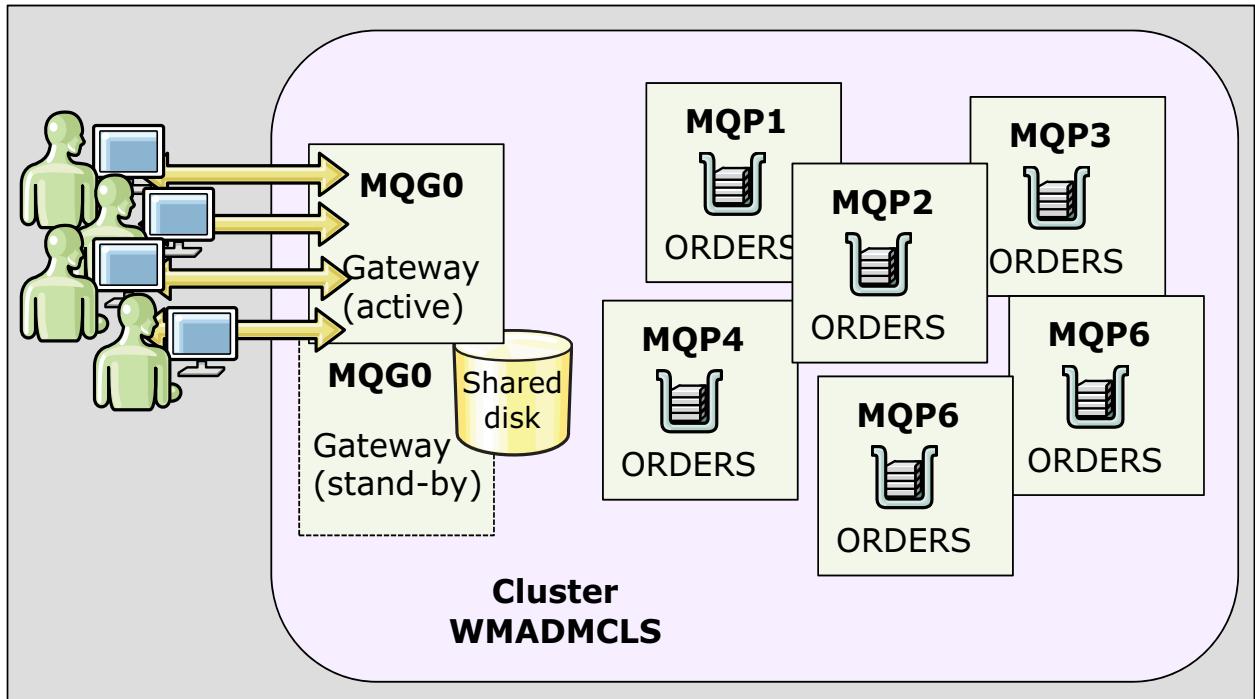
Applications need to plan to have immediate availability for new requests. IBM MQ clusters contribute in that area, by hosting the same queue name in more than one queue manager. However, a distinction is that if a queue manager fails, messages in the failed queue manager might be marooned until the failover solution completes its process, or the queue manager that failed is restored.

One exception is the use of z/OS shared queues. With shared queues, all queue managers' members of the queue-sharing group have access to the same message storage, the coupling facility. The only drawback is that a coupling facility failure would make all queue managers miss their messages, but coupling facilities are resilient, and they can also be set up with a backup coupling facility structure.

How does this fit in with the group and bind capabilities? When applications are bound to a specific queue manager, due to use of message groups or incorrect BIND options, they are considered to have queue manager affinities. Therefore, they are not good candidate applications for an IBM MQ cluster.

Application design revisited

Bind open options and use of segmented messages create queue manager affinities



Bind and Message groups

© Copyright IBM Corporation 2017

Figure 6-3. Application design revisited

When applications are bound to a specific queue manager, due to use of message groups or incorrect BIND options, they are considered to have queue manager affinities. Therefore, they are not good candidate applications for an IBM MQ cluster.

An application might have an affinity to queue manager MQP3 for various possible reasons, for example: because it is sending a group of messages that need to be in the same queue manager, or because the messages are being put in a programming loop with the “bind on open” option. This application is not able to start by using any of the other queue managers in the cluster should the MQP3 fail.

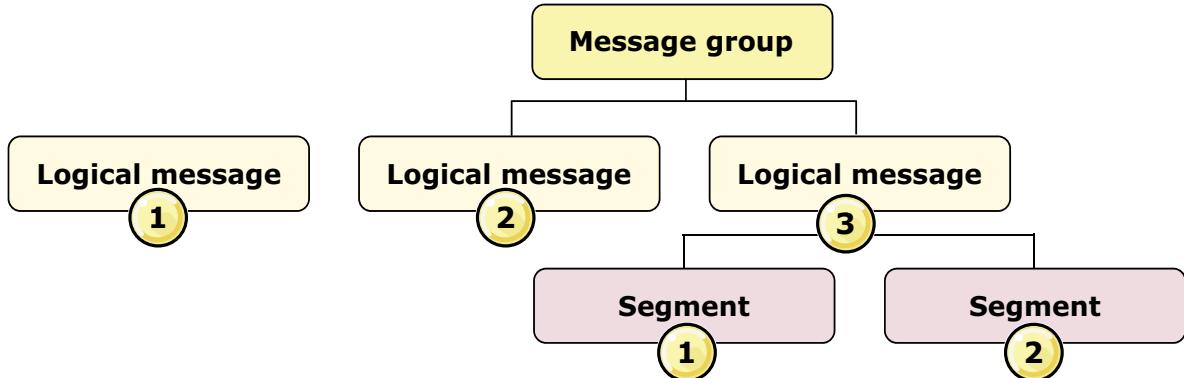
Bind attribute

- **MQOPEN options:**
 - MQOO_BIND_AS_Q_DEF recognizes the DefBind attribute of the queue definition, OPEN, NOTFIXED, and GROUP
 - MQOO_BIND_AS_Q_DEF is the initialized value
 - MQOO_BIND_ON_OPEN
 - MQOO_BIND_NOT_FIXED
 - Preferred for maximum cluster effectiveness
 - MQOO_BIND_ON_GROUP

- **MQOO_BIND_NOT_FIXED considerations**
 - If the MQOPEN MQOO_INPUT* or MQOO_BROWSE options are used, the queue manager forces selection of the local instance of the cluster queue
 - Although all instances of a cluster queue should have equal attributes, if MQOO_INQUIRE is used with MQOO_BIND_NOT_FIXED, successive MQINQ calls that use the same handle might require different instances of the cluster queue
 - Applications that require a series of messages to arrive in the same queue to complete a transaction should not use MQOO_BIND_NOT_FIXED

The MQPUT1 call behaves as if it has BIND_NOT_FIXED.

Physical and logical messages, segments, and groups



- Message group
 - Set of one or more logical messages, which are composed of one or more physical messages
- Logical message
 - Single unit of application information
 - One or more physical messages called *segments*
- Segments are anywhere in the queue
- Physical message
 - Smallest unit of information that can be placed on a queue

Bind and Message groups

© Copyright IBM Corporation 2017

Figure 6-5. Physical and logical messages, segments, and groups

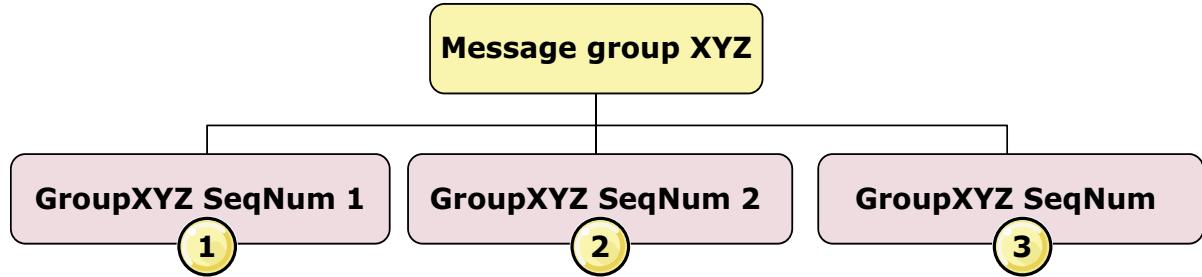
Messages can occur in groups. Grouping enables ordering of messages and segmentation of large messages within the same group.

Message segments are described later in this unit. For now, it is enough to know that a segment is one physical message that, when taken together with other related segments, makes up a logical message. Segments are useful when it is necessary to handle messages that are too large for the putting or getting application or for the queue manager.

Logical messages can be a physical message as well. If not made up of segments, then a logical message is the smallest unit of information, by definition, a physical message.

A message group is made up of one or more logical messages that consist of one or more physical messages that have some relationship.

Message groups



- Logical messages in a group have:
 - The same UNIQUE **GroupId** in the MQMD
 - A different **MsgSeqNumber** in the MQMD, starting at 1
 - Other fields in the MQMD that are independent
- Logical messages are ordered in a group
- Physical messages not in a group have:
 - MQGI_NONE in **GroupId**
 - Value of 1 in the **MsgSeqNumber**

[Bind and Message groups](#)

© Copyright IBM Corporation 2017

Figure 6-6. Message groups

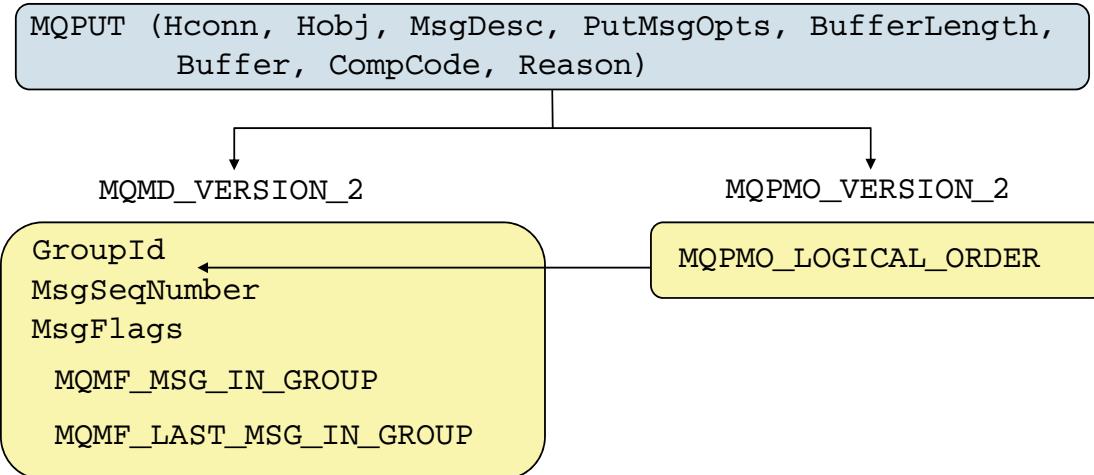
The message descriptor structure (or the MQMDE as described in *the Application Programming Reference*) in version 2 has some fields that are used when working with message groups and segments. The first of these fields is the **MQMD_GroupId**. A 24-byte field, it can contain a unique value to identify a message group. Each logical message that is a member of a particular group has the same **GroupId**.

The next new field is the **MQMD_MsgSeqNumber**; although the **GroupId** is the same, each logical message has a different **MsgSeqNumber** (the numbering starts at 1).

If a message is not part of a group, the **GroupId** is all nulls while **MsgSeqNumber** is set to 1.

The example in the slide shows three logical messages in a single group called XYZ. Each has a different **MsgSeqNumber** (1, 2, and 3).

Grouping logical messages



- The queue manager returns the **GroupId** and the **MsgSeqNumber**
- If not specified:
 - **GroupId** processing is similar to **MsgId** in the previous version
 - The application assigns **MsgSeqNumber**
- No mix for persistence

Figure 6-7. Grouping logical messages

The suggested approach to creating a message group is to allow the queue manager to create a unique **GroupId** and use it for each of the messages within a particular group.

Version 2 of the put message options structure includes a put message options (MQPMO) option MQPMO_LOGICAL_ORDER. Version 2 of the message descriptor contains another field that is called **MsgFlags**. By specifying MQPMO_LOGICAL_ORDER and setting **MsgFlags** to either MQMF_MSG_IN_GROUP or MQMF_LAST_MSG_IN_GROUP, the queue manager generates a new **GroupId** for the starting message of each new group. It keeps that **GroupId** and assigns a new **MsgSeqNumber** for each new logical message within the group.

If the last **GroupId** contained the **MsgFlag** of MQMF_LAST_MSG_IN_GROUP, the queue manager knows to create a **GroupId**. Since message groups imply that more than one message is placed on an output queue, this automatic handling by the queue manager is supported only for MQPUT, not MQPUT1.

If the required application uses message groups with MQPUT1 calls or controls the setting up of the message group, it would not use the MQPMO_LOGICAL_ORDER option.

If the application sets the **GroupId** to MQGI_NONE (nulls), a unique **GroupId** is generated. The application must ensure that unique **MsgSeqNumbers** are assigned for each logical message in the group and to ensure that the **GroupId** continues to be the same until a new **GroupId** is requested.

When MQPMO_LOGICAL_ORDER is specified, the queue manager requires that all messages in a group must be either persistent or non-persistent. If not, the call fails with MQRC_INCONSISTENT_PERSISTENCE. If MQPMO_LOGICAL_ORDER is not specified, it is the responsibility of the application to determine that persistence is kept consistent; the queue manager does not handle any checks. If a channel that allows fast messaging fails, some messages in a group can be lost. If a queue manager stops for any reason and the messages within the group are a mixture of persistent and non-persistent, some messages in a group can also be lost.

Retrieving logical messages (1 of 2)

MQGMO_VERSION_2

- MQGMO_ALL_MSGS_AVAILABLE
 - Messages available for retrieval only when all messages in a group arrive
- MQGMO_LOGICAL_ORDER specified
 - Messages in a group are returned in **MsgSeqNumber** order
 - Only one message group at a time for a Hobj
- MQGMO_LOGICAL_ORDER not specified
 - Application must select the correct **GroupId** and **MsgSeqNumber**
 - Can be used to restart in the middle of a group after a failure
 - Can switch to MQGMO_LOGICAL_ORDER
 - Used for applications that forward messages
- z/OS requires INDXTYPE of GROUPID

[Bind and Message groups](#)

© Copyright IBM Corporation 2017

Figure 6-8. Retrieving logical messages (1 of 2)

The get message options structure has a version 2, which must be used to allow the correct retrieval of logical messages within a group.

By specifying MQGMO_ALL_MSGS_AVAILABLE options, the program can prevent retrieval of messages that belong to an incomplete group. By specifying MQGMO_LOGICAL_ORDER, messages in groups (and segments in logical messages) are returned in order. This order can be different from the physical order of messages on a queue.

If both MQGMO_ALL_MSGS_AVAILABLE and MQGMO_LOGICAL_ORDER are specified, MQGMO_LOGICAL_ORDER is effective only when a current group or logical message exists.

As with the MQPUT, the application can control the MQGET of messages in groups. However, this technique is not the suggested approach except to restart a group after a system or queue manager failure. The queue manager then retains the group and segment information; and subsequent calls, by using the same queue handle, can revert to using MQGMO_LOGICAL_ORDER.

Similarly, messages that are forwarding physical messages should *not* use the MQGMO_LOGICAL_ORDER option or the **GroupId** of the original message can be corrupted.

IBM MQ for z/OS supports message grouping for shared and non-shared queues. The INDXTYPE of GROUPID for local queues enables this function. It uses message priority internally to provide an

efficient method of finding candidate groups and checking them for completeness.
INDXTYPE(GROUPID) can be used on the following commands:

- ALTER QLOCAL
- ALTER QMODEL
- DEFINE QLOCAL
- DEFINE QMODEL

Retrieving logical messages (2 of 2)

- **MatchOptions** field in MQGMO
 - For selective MQGETs
 - Only if MQGMO_LOGICAL_ORDER is not specified

MQMO_MATCH_MSG_ID
 MQMO_MATCH_CORREL_ID
 MQMO_MATCH_GROUP_ID
 MQMO_MATCH_MSG_SEQ_NUMBER

- **GroupStatus** field in MQGMO
 - Returned on the MQGET

MQGS_NOT_IN_GROUP
 MQGS_MSG_IN_GROUP
 MQGS_LAST_MSG_IN_GROUP

Figure 6-9. Retrieving logical messages (2 of 2)

The **MatchOptions** field of the version 2 MQGMO structure allows a program to control which group it retrieves.

If an application wants to retrieve a particular group, it can use the MQGMO_LOGICAL_ORDER option (if no current logical message is being processed) with the **MatchOption** MQMO_MATCH_GROUP_ID. It is also possible to retrieve any message with a specific **GroupId** by not specifying the MQGMO_LOGICAL_ORDER and specifying the **MatchOption** MQMO_MATCH_GROUP_ID.

The MQMO_MATCH_MSG_SEQ_NUM allows the retrieval of a specific message with a matching message sequence number. This setting is in addition to any other matches that might apply. If the MQMO_MATCH_MSG_SEQ_NUM is combined with the MQGMO_LOGICAL_ORDER, it is not valid.

Finally, an application can determine whether it processed the final message in a group by checking the MQGMO **GroupStatus** field after a message is retrieved. The **GroupStatus** field would contain a value that the symbolic MQGS_LAST_MSG_IN_GROUP represents. If not, and a group is processed, the value would be MQGS_MSG_IN_GROUP.

Spanning units of work

- Putting application
 - Some of the group might be committed before a failure
 - Status information must be saved
 - Simplest way is to use a status queue
 - Status information consists of **GroupId** and **MsgSeqNumber**
 - Status queue is empty if a complete group is put successfully
 - MQPMO_LOGICAL_ORDER cannot be used on the first put of the restart application
- Getting application
 - Status queue again
 - The first GET of the restart application must match on **GroupId** and **MsgSeqNumber**

Figure 6-10. Spanning units of work

The MQPMO_LOGICAL_ORDER option affects units of work as follows:

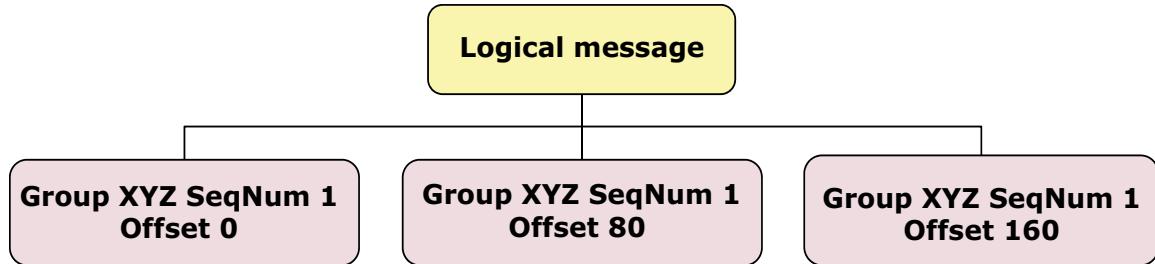
- If the first physical message in the unit of work specifies MQPMO_SYNCPOINT, then all subsequent physical messages in the group or logical message must use the same option. However, it is not necessary to put them within the same unit of work. This technique allows spreading a message group or logical message that has many physical messages into smaller units of work.
- Conversely, if the first physical message specifies MQPMO_NO_SYNCPOINT, then none of the subsequent physical messages within the group or logical message can do otherwise.

If these conditions are not met, the MQPUT fails with MQRC_INCONSISTENT_UOW. The conditions that are described for the MQPUT are the same when using the MQGMO_SYNCPOINT or MQGMO_NO_SYNCPOINT options.

Since it is possible to split a group or logical message over multiple units of work, it is possible to have some of the physical messages but not all committed when a failure occurs. It is the responsibility of the application to track the status information that is associated with a group or logical messages that span units of work. By tracking the **GroupId** and the **MsgSeqNumber** on a special status queue (by using syncpoint options for the MQPUT and MQGET), an accurate picture of what was completed can be kept.

If a restart is done, this information can be used without specifying the get or put message option for logical order to essentially restart correct processing of a group or a logical message.

Message segmentation



- A segment is:
 - A physical message
 - Identified by the **GroupId**, **MsgSeqNumber**, and **Offset** fields in the message descriptor
- Every segment starts with an MQMD
- A message can be segmented and reassembled
 - By the queue manager
 - By an application

[Bind and Message groups](#)

© Copyright IBM Corporation 2017

Figure 6-11. Message segmentation

The previous topic defined message segments at a high level. You now must explore how to use them in a program. These discussions make an assumption that all MQPUT and MQGET work operates within a unit of work. Use this configuration to avoid any possibility of incomplete groups in a network.

Segmented messages use another field available in the version 2 message descriptor (or MQMDE as described in the *Application Programming Guide*). The **Offset** field depicts where a particular segment of data would begin in a complete logical message.

Putting and getting segmented messages can be done under control of the queue manager or the application.

z/OS, and applications that use IBM MQ classes for JMS, do not support message segmentation.

Segmentation by the queue manager

- MQMF_SEGMENTATION_ALLOWED in **MsgFlag** of MQMD version 2
- Queue manager splits user-defined formats on 16-byte boundaries
 - Split never occurs in the middle of an IBM MQ header
 - Do not make any assumptions on how the queue manager splits a message
- Queue manager modifies MQRO_..._WITH_DATA to MQRO_... in MQMD if the segment does not contain any of the first 100 bytes of the user message
- Persistent message segmentation can be done only within a unit of work
 - If no user-defined unit of work exists, the queue manager creates a unit of work
 - MQRC_UOW_NOT_AVAILABLE if a user-defined unit of work exists and MQPMO_NO_SYNCPOINT is specified

Figure 6-12. Segmentation by the queue manager

If the **MsgFlag** field in the version 2 message descriptor includes the value that is represented as MQMF_SEGMENTATION_ALLOWED, the queue manager handles building a segmented message. If the queue manager recognizes that the message is too large for its **MaxMsgLength** (a queue manager attribute) or for the maximum message length of a queue, segmentation is done.

If the message descriptor included the MQRO_..._WITH_DATA option, it is modified to eliminate the request for data on any segments that do not include the first 100 bytes.

Finally, since the queue manager handles the splitting into segments and the rebuilding of the message, no assumptions can be made about how the data is split.

Unit-of-work processing has restrictions; for persistent messages, unit-of-work processing is required. If the queue manager determines that no application-defined unit of work is active, the queue manager creates a unit of work that is active only during the call.

Reassembly by the queue manager or application

- MQGMO assumed to be at version 2 for all named options
- MQGMO_ALL_SEGMENTS_AVAILABLE
 - Messages available for retrieval only when all segments arrive
 - MQGMO_ALL_MSGS_AVAILABLE forces it
- MQGMO_LOGICAL_ORDER specified
 - Messages in a group are returned in **Offset** order
 - Only one logical message at a time for a **Hobj**
- MQGMO_LOGICAL_ORDER not specified
 - Application must supply the **GroupId**, **MsgSeqNumber**, and **Offset**
 - Can be used to restart in the middle of a group after a failure; can then switch to MQGMO_LOGICAL_ORDER
- MQGMO_COMPLETE_MESSAGE
 - With this option, the queue manager reassembles the complete message
 - Same persistence and unit-of-work considerations as described for logical message reassembly

Figure 6-13. Reassembly by the queue manager or application

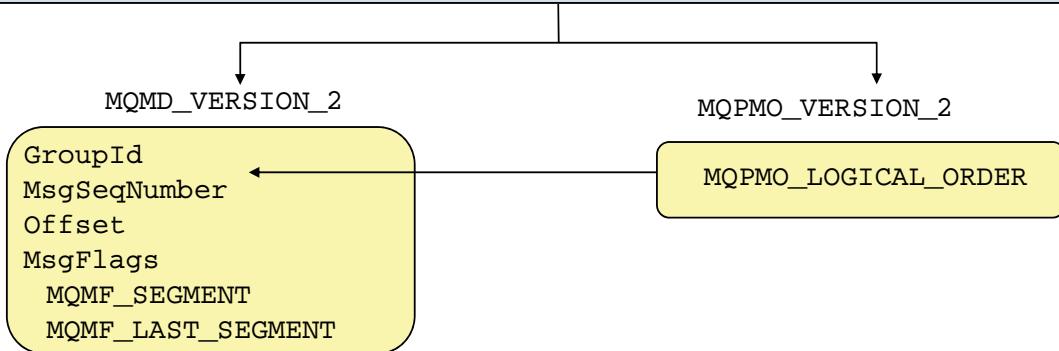
The getting application must include a GET message option, such as MQGMO_COMPLETE_MESSAGE. If the queue manager determines that the message is segmented, it retrieves all the segments and returns the complete message in the program buffer.

Data conversion is done after the message is placed in the buffer of the program; it should not change with message segments. However, data conversion that a sender channel requests fails for a message segment because the exit does not have all of the data from the message at one time.

If the queue manager determines that the message is a persistent message and no user-defined unit of work exists, the queue manager creates a unit of work during the call. It does *not* automatically create a unit of work for non-persistent messages. All messages have the potential for segmentation; use unit-of-work processing.

Segmentation by the application

```
MQPUT (Hconn, Hobj, MsgDesc, PutMsgOpts, BufferLength,
       Buffer, CompCode, Reason)
```



- Queue manager returns the **GroupId**, **MsgSeqNumber**, and **Offset**
- Only one logical message at a time
- Segments are put in order of increasing segment offset with no gaps
- If not specified:
 - **GroupId** processing similar to **MsgId** in previous version
 - Application assigns **MsgSeqNumber**
- **No mix for persistency**

Bind and Message groups

© Copyright IBM Corporation 2017

Figure 6-14. Segmentation by the application

Application segmentation is used for two reasons:

1. Queue manager segmentation is not sufficient because the application buffer is not large enough to handle the full message.
2. The sender channels must convert the data, and the putting program must split the message on specific boundaries to enable successful conversion of the segments.

The application can include the put message options of `MQPMO_LOGICAL_ORDER`. The same caution applies to using unit-of-work processing. As each segment is built and the `MQPUT` is run, the application must ensure that the **MsgFlags** field in the message descriptor includes either `MQMF_SEGMENT` or `MQMF_SEGMENT_LAST`. The queue manager assigns and maintains the **GroupId**, **MsgSeqNumber**, and **Offset**.

If the application does not specify the `MQPMO_LOGICAL_ORDER`, then the program is responsible for ensuring that a new **GroupId** is assigned and assigning an appropriate **MsgSeqNumber** and **Offset**.

Putting and getting messages that span units of work are allowed.

Other considerations for application reassembly

- **MatchOptions** field in MQGMO for selective MQGETs
 - MQMO_MATCH_MSG_ID
 - MQMO_MATCH_CORREL_ID
 - MQMO_MATCH_GROUP_ID
 - MQMO_MATCH_OFFSET and MQGMO_MATCH_MSG_SEQ_NUMBER if MQGMO_LOGICAL_ORDER is not specified
- **SegmentStatus** field in MQGMO returned on the MQGET
 - MQSS_NOT_A_SEGMENT
 - MQSS_SEGMENT
 - MQSS_LAST_SEGMENT

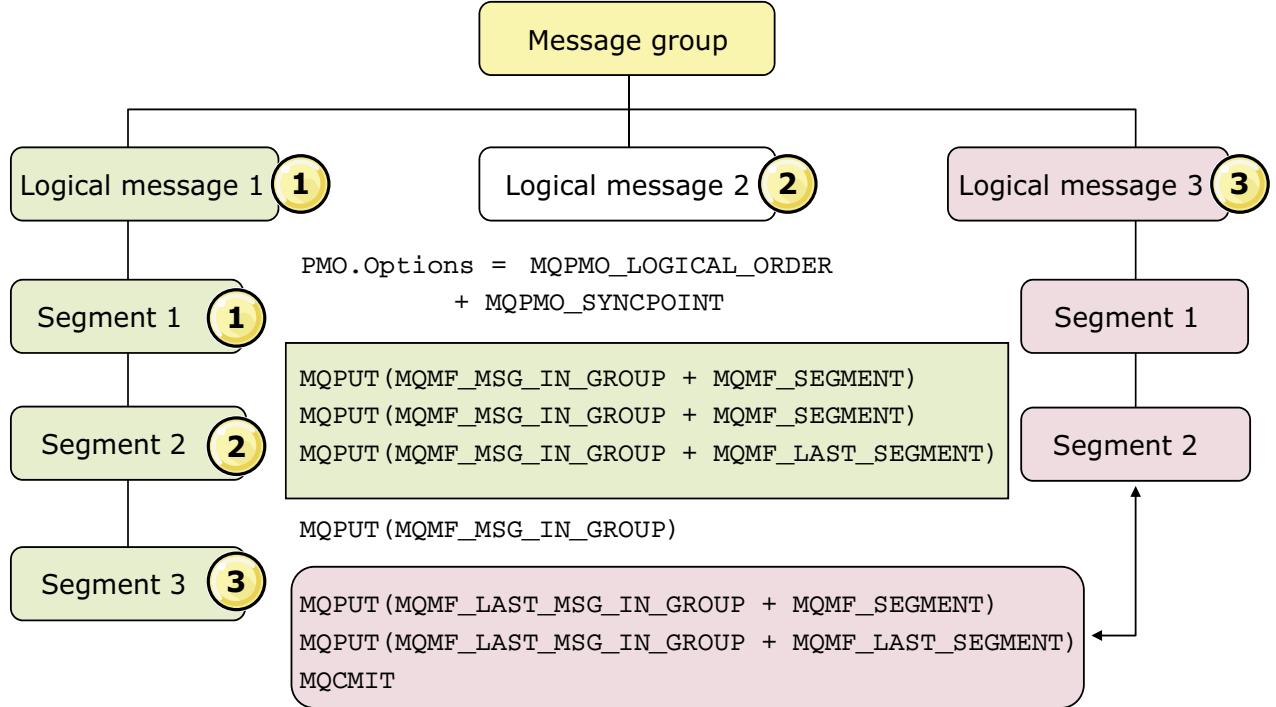
Figure 6-15. Other considerations for application reassembly

It is possible to control which segment a program retrieves and have total control over message retrieval by a program. The **MatchOptions** field of the version 2 get message options structure allows for this capability. The following scenario is similar to the process of retrieving a complete message group. You learn how to retrieve a complete logical message.

If an application wants to retrieve a particular logical message, it can begin retrieval of messages from the **GroupId** and logical message by using **MatchOption** MQMO_MATCH_GROUP_ID. It can get the logical message by using MQMO_MATCH_MSG_SEQ_NUM starting with **Offset** set to zero and including **MatchOption** MQMO_MATCH_OFFSET. If the MQMO_MATCH_MSG_SEQ_NUM is combined with the MQGMO_LOGICAL_ORDER, it is not valid.

Finally, an application can determine whether it processed the final segment of a logical message by checking another field in the get message options structure after a message is retrieved. The **SegmentStatus** field would contain a value that the symbolic MQSS_LAST_SEGMENT represents. If not, and a group was being processed, the value would be MQSS_SEGMENT. The **GroupStatus** field can be checked to see whether all logical messages within the group were processed.

Segmentation and message group: Sample MQPUT



Bind and Message groups

© Copyright IBM Corporation 2017

Figure 6-16. Segmentation and message group: Sample MQPUT

The sample message group has three logical messages. Logical message 1 has three segments. Logical message 2 has only one segment. Logical message 3 has two segments.

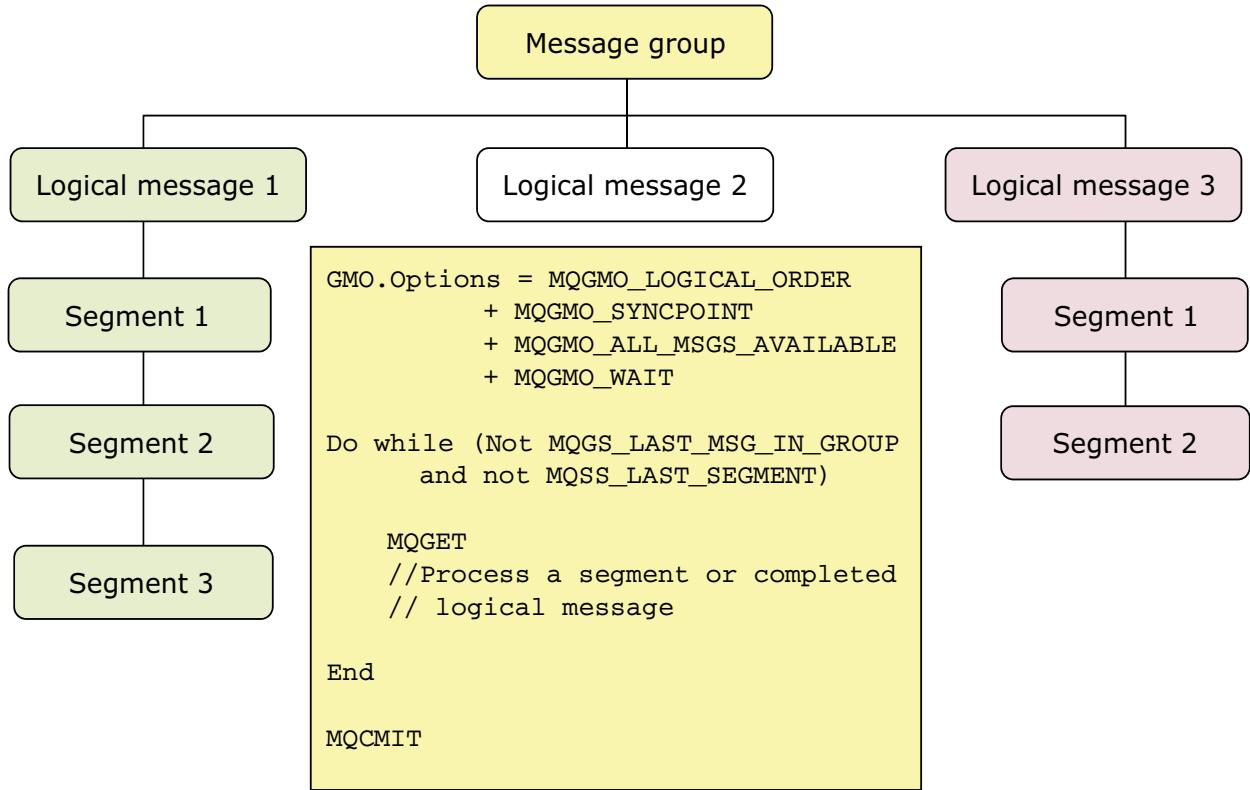
The value in the put message options structure options field is calculated by adding MQPMO_LOGICAL_ORDER and MQPMO_SYNCPOINT. Then, the actual MQPUTs are shown.

Because the application used MQPMO_LOGICAL_ORDER, the queue manager handles the update **MsgSeqNumber** and **Offset** in the message descriptor. The application must ensure that the **MsgFlags** field is properly set.

After all segments of the first logical message are complete, the second message, consisting of one segment, is prepared and the MQPUT is issued. Because **MsgFlags** has no information about the segment, it is not necessary to concern the queue manager or application with segmentation. The physical message is the logical message.

Finally, the third logical message again requires segmenting. The application builds each segment and allows the queue manager to set the values in **MsgSeqNumber** and **Offset** while it makes sure that the **MsgFlags** field is properly set up.

Segmentation and message group: Sample MQGET



Bind and Message groups

© Copyright IBM Corporation 2017

Figure 6-17. Segmentation and message group: Sample MQGET

The application is going to reassemble the messages that it retrieves. Therefore, the options for the get message options structure include syncpoint, logical order, and the combination of wait and all messages available, which means that no segments are retrieved until they all arrive.

The loop runs until after the last message is retrieved for the group because of the check for the GroupStatus (MQGS_LAST_MSG_IN_GROUP) and SegmentStatus (MQSS_LAST_SEGMENT).

Again, it has a user-defined unit of work, so the queue manager does not do any unit of work of its own.

Segmented messages and reports

- Report message can be generated for each of the segments
- MQMD fields are copied to queue manager generated reports
 - **GroupId**, **MsgSeqNumber**, **Offset**
 - **MsgFlags**
 - **OriginalLength** of the segment or message for which the report is generated
- Use MQGMO_COMPLETE_MESSAGE to reassemble report messages that have the same feedback code (that is, COA, COD)
 - Application should be ready to deal with orphan report segments
- Reports that are generated from a queue manager or MCA that does not support segmentation do not contain the segment information
 - Use MQRO_..._WITH_(FULL_)DATA

[Bind and Message groups](#)

© Copyright IBM Corporation 2017

Figure 6-18. Segmented messages and reports

Report messages require special processing by the queue manager.

If the MQGMO_ALL_SEGMENTS_AVAILABLE option is specified, the queue manager checks to see whether at least one report message is available for each segment that makes up the logical message. If so, the requested condition is satisfied. However, the queue manager does not check the types of report messages present so it might have a mixture of report types.

By specifying MQGMO_COMPLETE_MSG, the queue manager checks to see whether all the report messages of the report type that relates to all the different segments are present. If so, it then retrieves the message. For this action to be possible, a queue manager or MCA that supports segmentation must generate the report messages, or the originating application must request at least 100 bytes of message data. The application can use the MQRO_..._WITH_DATA or MQRO_..._WITH_FULL_DATA option to request the message data. If less than the full amount of application data is present for a segment, nulls replace the missing bytes in the report message that is sent back.

Unit summary

- Explain the importance of limiting applications that introduce queue manager affinities in the IBM MQ architecture
- Describe the use of bind-related attributes in queue definitions and MQOPEN options
- Describe how to write IBM MQ applications that require a distinct sequence of messages to complete processing

Bind and Message groups

© Copyright IBM Corporation 2017

Figure 6-19. Unit summary

Review questions

1. True or False: The MQOPEN option `MQOO_BIND_NOT_FIXED` helps distribute work among different instances of a clustered queue.
2. Where is the group ID specified?
3. True or False: A message group provides a way to group related messages and ensure ordering on message retrieval.
4. If you wanted to ensure that your application got messages only from complete groups, which option would you use?
 - a. `MQGMO_COMPLETE_MESSAGE`
 - b. `MQGMO_ALL_MSGS_AVAILABLE`
 - c. `MQMO_MATCH_GROUP_ID`
 - d. `MQGMO_ALL_SEGMENTS_AVAILABLE`



Bind and Message groups

© Copyright IBM Corporation 2017

Figure 6-20. Review questions

Write your answers here:

- 1.
- 2.
- 3.
- 4.

Review answers (1 of 2)

1. True or False: The MQOPEN option

`MQOO_BIND_NOT_FIXED` helps distribute work among different instances of a clustered queue



The answer is: True. It keeps all messages from being bound to a specific queue manager.

2. Where is the group ID specified?

The answer is: In the MQMD GroupId field of the message descriptor structure.

3. True or False: A message group provides a way to group related messages and ensure ordering on message retrieval.

The answer is: True. If the group identifier is uniqueMessage, groups and segments can be processed correctly.

Review answers (2 of 2)

4. If you wanted to ensure that your application got messages only from complete groups, which option would you use?

- a. MQGMO_COMPLETE_MESSAGE
- b. MQGMO_ALL_MSGS_AVAILABLE
- c. MQMO_MATCH_GROUP_ID
- d. MQGMO_ALL_SEGMENTS_AVAILABLE

The answer is: b. MQGMO_ALL_MSGS_AVAILABLE prevents retrieval of messages for incomplete groups.



Unit 7. Committing and backing out units of work

Estimated time

00:30

Overview

In this unit, you learn how to coordinate actions that must occur together to complete a valid process. The unit describes some of the terminology that is used, local and global units of work, and the IBM MQ function calls that are used for completion or back out of work. Finally, the unit describes details and considerations for using triggering and syncpoint.

How you will check your progress

Accountability:

- Review questions
- Lab Exercises

References

IBM Knowledge Center for IBM MQ V9 documentation

Unit objectives

- Describe the terminology that is associated with committing and backing out units of work
- Differentiate between local and global units of work
- Describe how syncpoint control is implemented in IBM MQ
- Describe the syntax and use of the MQBEGIN, MQCMIT, and MQBACK function calls
- Explain how to use triggering to start an application
- Describe considerations to observe when using triggering and syncpoint in the same application

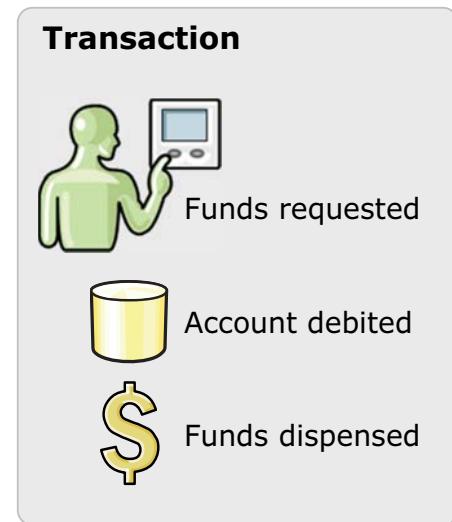
[Committing and backing out units of work](#)

© Copyright IBM Corporation 2017

Figure 7-1. Unit objectives

Transactions: Terminology

- A **resource manager** is a system that owns and controls its components such as:
 - A database manager owns its tables
 - A queue manager owns its queues
- A **transaction** or **unit of work** is a set of changes that must be completed in their entirety (**committed**) or restored to a previous consistent state (**backed out**)
- A **transaction manager** is a subsystem that coordinates units of work
- A IBM MQ queue manager can:
 - Act as a transaction manager over its own resources
 - Manage updates to DB2 tables
 - Run under a compatible external transaction manager
- An ATM withdrawal is a common scenario of a process that needs to be coordinated with a transaction manager



[Committing and backing out units of work](#)

© Copyright IBM Corporation 2017

Figure 7-2. Transactions: Terminology

A familiar example of a transaction is the withdrawal of funds from an automated teller machine. The process must ensure that after the funds are deducted from your account, they are dispensed to you. If a problem arises with dispensing the funds, then the amount that is deducted from your account needs to be reversed, or backed out. This concept is sometimes known as a transaction. Controlling the transaction handling by IBM MQ resources is ensured by getting and putting within syncpoint.

When a program gets a message from a queue in syncpoint, that message remains on the queue until the program commits the unit of work. However, the message is not available to other programs for retrieval.

Similarly, when a program puts a message on a queue within a unit of work, the message is made visible to other programs only when the program commits the unit of work. This behavior ensures that the entire unit of work (perhaps consisting of the MQGET, some validation of the input data, and some processing and the MQPUT of an output message) is successfully completed. During the processing, if a problem is encountered, all the activities that are associated with the unit of work can be backed out. Essentially, everything is put back as it was before any processing of that unit of work started. This state is known as the last point of consistency.

In addition to messages on queues, other resource managers, such as database managers, can use unit-of-work processing to ensure that their resources are coordinated.

In some cases, these different resource managers and their activities can be combined into a single unit of work. A transaction manager generally coordinates the unit of work. IBM WebSphere Application Servers, CICS, Encina, and BEA Tuxedo are examples of transaction managers.

A normal end of process is also an implied commit for unit-of-work coordinators; an abnormal end of process is a rollback or backout.

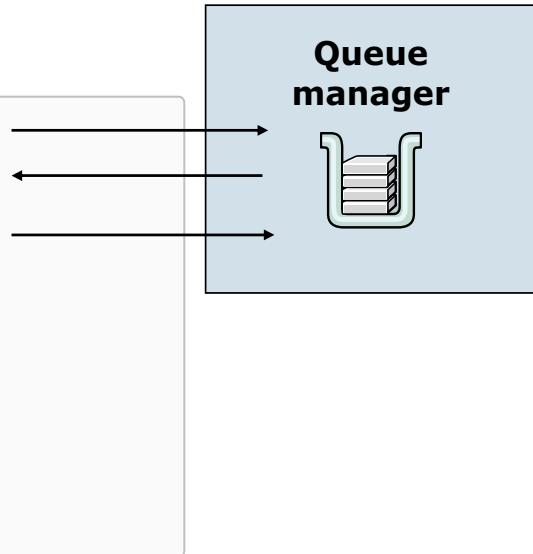
Local units of work

Application

```

MQPUT # w/ syncpoint opt
MQGET # w/ syncpoint opt
MQPUT # w/ syncpoint opt
|
If RC = 0      # no errors
    MQCMIT      #commit
else          # errors
    MQBACK      #rollback

```



[Committing and backing out units of work](#)

© Copyright IBM Corporation 2017

Figure 7-3. Local units of work

A local unit of work involves only updates to IBM MQ resources (queues). The queue manager uses a single-phase commit to provide syncpoint coordination.

Start a unit of work by setting the syncpoint option for each message operation you want in your unit of work:

```

MQGET = MQGMO_SYNCPOINT
MQPUT and MQPUT1 = MQPMO_SYNCPOINT

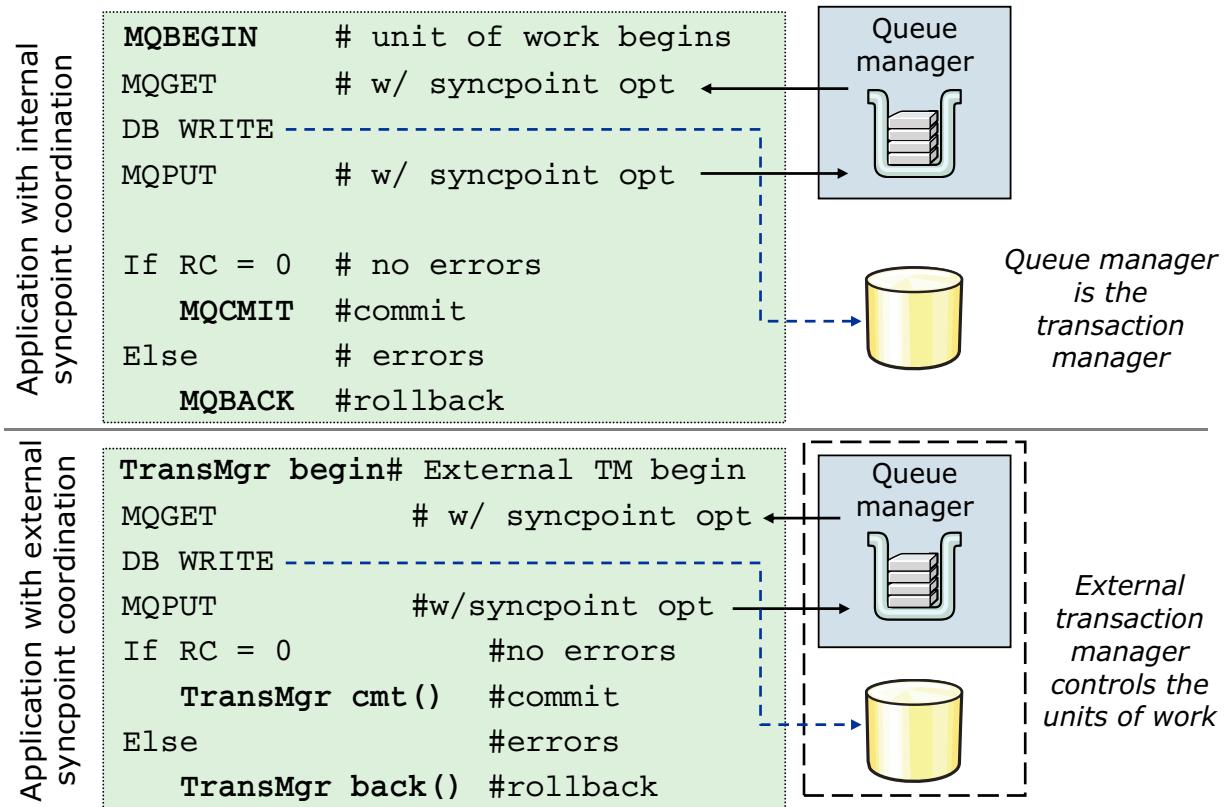
```

An MQCMIT issued after the unit of work commits the operations. An MQBACK rolls back the operations. A broken connection also ends the unit of work.

If an application issues an MQDISC, a *commit* attempt is made. But if the application terminates without disconnecting, a rollback happens.



Global units of work: Syncpoint coordination



Committing and backing out units of work

© Copyright IBM Corporation 2017

Figure 7-4. Global units of work: Syncpoint coordination

A global unit of work includes the updates to resources that belong to other resource managers. The IBM MQ distributed queue managers (AIX, HP-UX, Solaris, Linux, and Windows) support internal syncpoint coordination of global units of work in some cases.

To start the internal syncpoint coordination of a global unit of work, an MQBEGIN call is used. The only parameters that are supplied are the queue manager handle as input, and the completion and reason codes as output. The updates to resources are handled as they would normally be when within a unit of work. When ready to commit, the MQCMT call is used. MQBACK is used to back out all updates. These calls are reviewed later in more detail. Check to see which database resource managers can use IBM MQ to coordinate their updates.

External syncpoint coordination occurs when a syncpoint coordinator other than IBM MQ is selected, for example, WebSphere Application Server, CICS, or Tuxedo. In this situation, IBM MQ registers its interest in the outcome of the unit of work with the syncpoint coordinator so that it can commit or roll back any uncommitted get or put operations as required. When an external coordinator is used, an MQCMT, MQBACK, and MQBEGIN cannot be issued. Calls to these functions fail with the reason code MQRC_ENVIRONMENT_ERROR. For any MQPUT or MQGET operation, you should specify the MQPMO_SYNCPOINT or MQGMO_SYNCPOINT option.

Syncpoint

```

MQGET      #(customer order)
Write DB
MQPUT      #(dispatch request)
MQPUT      #(delivery confirmation)
Commit

```

- Option on each MQGET, MQPUT, or MQPUT1
 - NO_SYNCPOINT: Message is added or removed immediately
 - SYNCPOINT: MQGET, MQPUT, or MQPUT1 becomes visible or removed when transaction is committed
- Default is platform-dependent

[Committing and backing out units of work](#)

© Copyright IBM Corporation 2017

Figure 7-5. Syncpoint

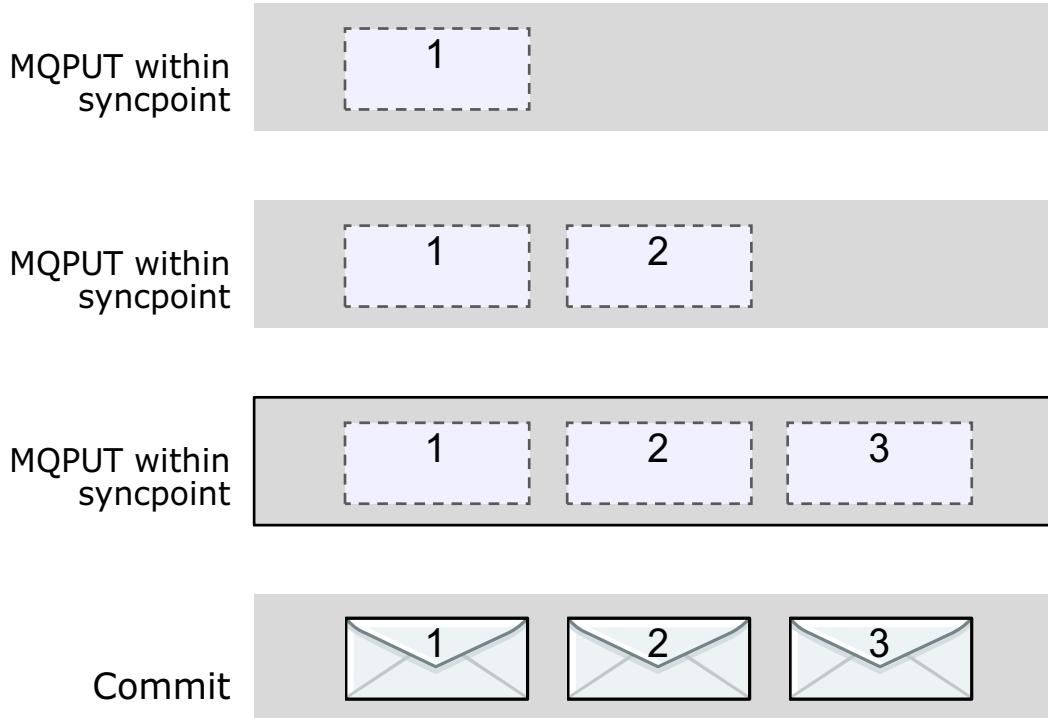
Not all MQGETs and MQPUTs within a program must be part of a unit of work. It is possible to change the MQGMO_SYNCPOINT or MQPMO_SYNCPOINT option to MQGMO_NO_SYNCPOINT or MQPMO_NO_SYNCPOINT.

With no syncpoint, the messages that are retrieved from a queue are removed from the queue immediately (a destructive read); messages that are put on an output queue are immediately available for retrieval by other programs.

Syncpoint control delays the deletion of messages that are retrieved from input queues and delays delivery of messages that are placed on output queues until the unit of work is committed.

Finally, it is best to explicitly specify the intended syncpoint option since the default varies depending on the platform where the program runs.

MQPUT within syncpoint control



[Committing and backing out units of work](#)

© Copyright IBM Corporation 2017

Figure 7-6. *MQPUT within syncpoint control*

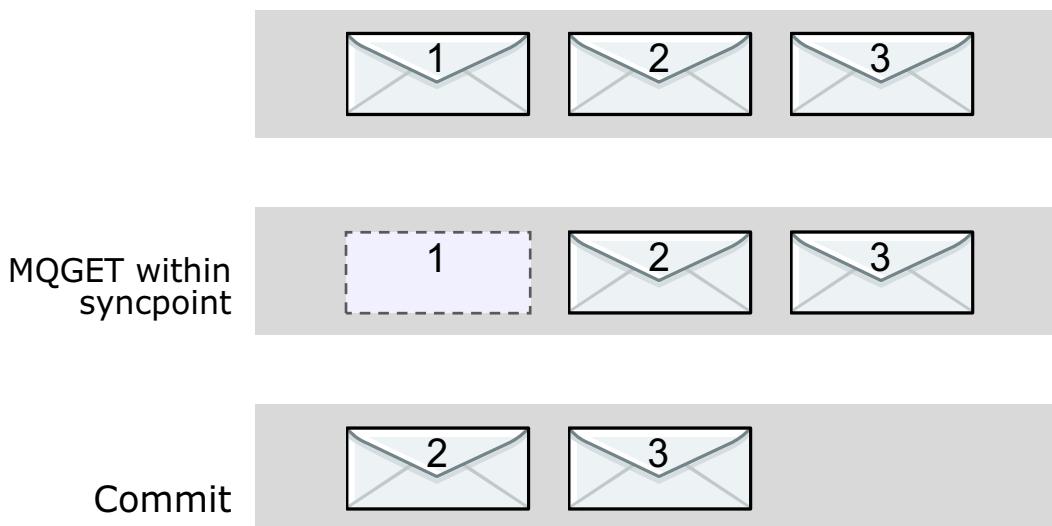
The put message options MQPMO_SYNCPOINT must be used with the arrival of a message on a queue. The opposite option is MQPMO_NO_SYNCPOINT. Explicitly set the intended option because the default differs by platform.

When MQPMO_SYNCPOINT is specified, the message is physically placed on the queue immediately, and the completion code indicates that result. However, the message, although physically on the queue, is not visible or available to other applications.

As the example shows, when syncpoint is in use, messages can continue to be placed on the queue. They count toward the number of messages on the queue and they are part of the physical count. The current depth reflects these messages, but any attempt to use MQGET to retrieve them fails with MQRC_NO_MSG_AVAILABLE.

The messages are available to other applications only after the commit is issued.

MQGET within syncpoint control



Committing and backing out units of work

© Copyright IBM Corporation 2017

Figure 7-7. *MQGET within syncpoint control*

The MQGET with syncpoint means that the application is given a copy of the message, and the actual message is left on the input queue but marked for deletion and not available to other MQGET calls. The physical number of messages on a queue includes those messages that are retrieved in syncpoint. The messages are not physically removed until the commit is completed (delayed deletion).

If a rollback occurs, the message is made available again and the delete flag is turned off.

MQBEGIN

- Begins a unit of work that the queue manager coordinates, and that can involve external resource managers
- BeginOptions structure (MQBO) allows the application to specify options that relate to the creation of a unit of work

```

MQHCONN Hconn;           /* Connection handle */
MQBO BeginOptions;       /* MQBEGIN control options */
MQLONG CompCode;         /* Completion code */
MQLONG Reason;           /* Reason code */

. . . . . . . . .

MQBEGIN Hconn,           /* Connection handle */
&BeginOptions,           /* MQBEGIN options */
&CompCode,               /* Completion code */
&Reason);                /* Reason code */

```

[Committing and backing out units of work](#)

© Copyright IBM Corporation 2017

Figure 7-8. MQBEGIN

The MQBEGIN call begins a unit of work that the queue manager coordinates, and that might involve external resource managers.

The parameters are simple. The connection handle is passed as input and completion and reason codes are returned as output. Be sure to check the completion and reason codes after each call.

The MQBEGIN call is supported in the following environments:

- AIX
- HP-UX
- IBM i
- Solaris
- Windows

MQBEGIN error codes

Reason codes when return code is MQCC_WARNING

MQRC_NO_EXTERNAL_PARTICIPANTS

(2121, X'849') No participating resource managers register

MQRC_PARTICIPANT_NOT_AVAILABLE

(2122, X'84A') Participating resource manager not available

Reason codes when return code is MQCC_FAILED (partial)

MQRC_BO_ERROR

(2134, X'856') Begin-options structure not valid

MQRC_CALL_IN_PROGRESS

(2219, X'8AB') MQI call entered before previous call complete

MQRC_ENVIRONMENT_ERROR

(2012, X'7DC') Call not valid in environment

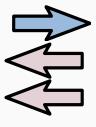
MQRC_UOW_IN_PROGRESS

(2128, X'850') Unit of work already started

MQCMIT

- Make permanent all the message gets and puts that occurred since the last synchronization point
 - Messages that are put as part of a unit of work are made available to other applications
 - Messages that are retrieved as part of a unit of work are deleted
- On z/OS, only batch programs use the call
- On IBM i, this call is not supported for applications that run in compatibility mode

```

MQHCONN Hconn;           /* Connection handle */ */
MQLONG CompCode;         /* Completion code */ */
MQLONG Reason;           /* Reason code */ */
...
MQCMIT Hconn,           /* Connection handle */ */
&CompCode,               /* Completion code */ */
&Reason);               /* Reason code */ */


```

[Committing and backing out units of work](#)

© Copyright IBM Corporation 2017

Figure 7-10. MQCMIT

After a program is satisfied with all processing within a unit of work, the MQCMIT call can be used to process the completion of the unit of work. It physically deletes any messages that are retrieved from input queues and makes messages that are put to output queues available to other applications.

If the queue manager is running in an environment where an external coordinator such as CICS is used, the MQCMIT call cannot be used. Any attempt to use MQCMIT, MQBACK, or MQBEGIN results in a failure with MQRC_ENVIRONMENT_ERROR.

All MQPUT and MQGET calls that are done in syncpoint (and any participating updates for other resource managers that occurred after the MQBEGIN call) are considered part of the unit of work and are committed. If a global unit of work is being committed (including resources that are not IBM MQ resources), then a two-phase commit process is used. Therefore, all the participating resource managers are first asked to prepare for commit. Only if all the resource managers acknowledge that they are prepared to commit does the second phase (the actual commit) occur. Otherwise, all the involved resources are rolled back.

MQCMIT error codes

Reason codes when return code is MQCC_WARNING

MQRC_BACKED_OUT (2003, X'7D3')

Unit of work backed out

MQRC_OUTCOME_PENDING (2124, X'84C')

Result of commit operation is pending

Reason codes when return code is MQCC_FAILED (partial)

MQRC_CALL_INTERRUPTED

(2549, X'9F5') MQPUT or MQCMIT was interrupted and reconnection processing cannot reestablish a definite outcome

MQRC_ENVIRONMENT_ERROR

(2012, X'7DC') Call not valid in environment

MQRC_OUTCOME_MIXED

(2123, X'84B') Result of commit or back-out operation is mixed

MQRC_RECONNECT_FAILED

(2548, X'9F4') After reconnecting, an error occurred reinstating the handles for a reconnectable connection

MQBACK

- Back out all the message gets and puts that occurred since the last syncpoint
 - Messages that are put as part of a unit of work are deleted
 - Messages that are retrieved as part of a unit of work are reinstated on the queue
- On z/OS, only batch programs use this call
- On IBM i, this call is not supported for applications that are running in compatibility mode

```

MQHCONN Hconn;           /* Connection handle */ */
MQLONG CompCode;          /* Completion code */ */
MQLONG Reason;            /* Reason code */ */
...
MQBACK Hconn,             /* Connection handle */ */
&CompCode,                /* Completion code */ */
&Reason);                /* Reason code */ */


```

[Committing and backing out units of work](#)

© Copyright IBM Corporation 2017

Figure 7-12. MQBACK

The MQBACK call is the reverse of the MQCMT call. All resources that are involved in the unit of work are returned to the state that they were in before the unit of work began. Retrieved messages are no longer marked for deletion and are again available to get. Messages that are put are removed from the output queues. In a global unit of work (identified by the MQBEGIN), the MQBACK also backs out all updates by the other resource managers.

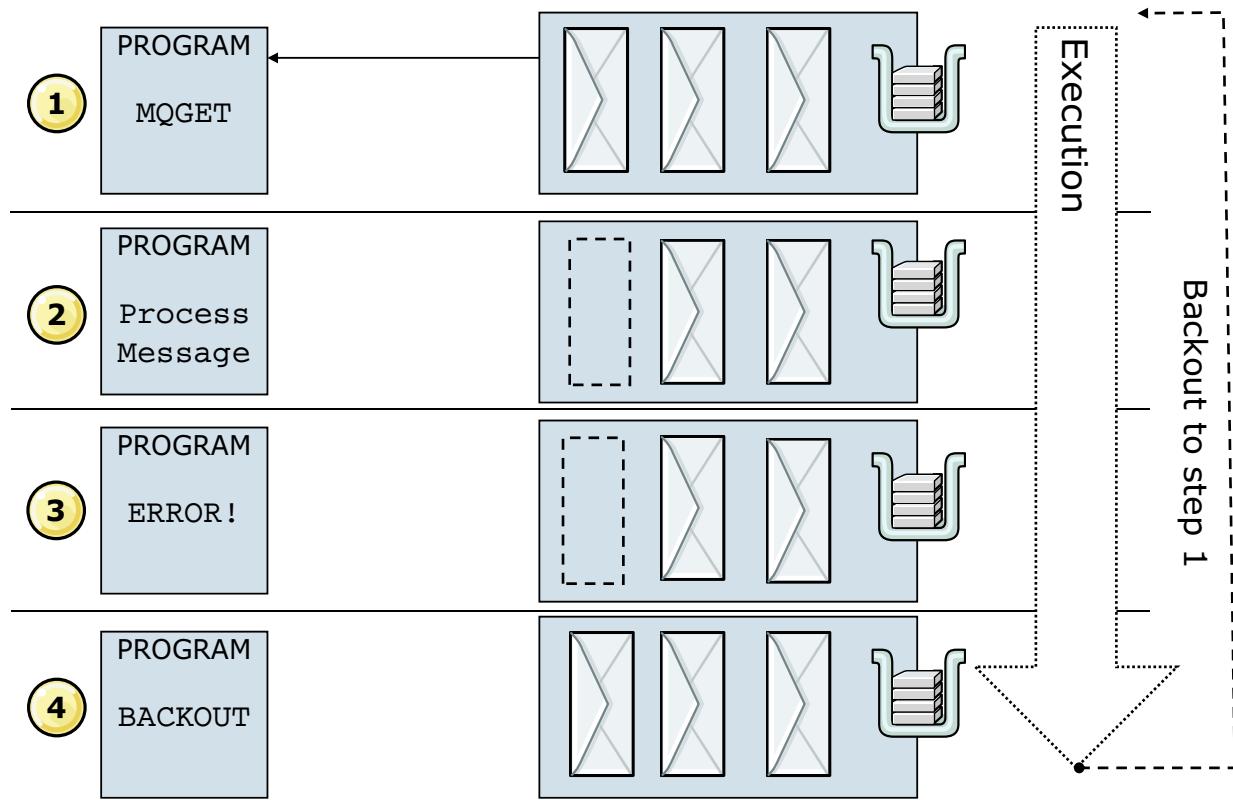
The parameters are simple. The connection handle is passed as input, and completion and reason codes are returned as output. Be sure to check the completion and reason codes after each call.

After rollback, for any messages that are retrieved by using an MQGET in syncpoint, the BackoutCount is incremented by one.

IBM Training



Poisoned messages



Committing and backing out units of work

© Copyright IBM Corporation 2017

Figure 7-13. Poisoned messages

If a message is retrieved from a queue in syncpoint and some data in that message contains invalid data that causes a backout, the message that contains the bad data is available on the input queue. This process can result in a loop where the message is continually rolled back and retrieved.

The message descriptor has a field that might be useful, the BackoutCount. When a message participates in a rollback, the counter that is maintained in the field increments by one. It is now possible to check that BackoutCount when the next MQGET is issued. If it is greater than zero, some type of error handling routine can be done.

Some attributes are also associated with the queue definition. On queue managers other than IBM i, the BOTHRESH (BackoutThreshold) can be set to some value. It also has a BOQNAME (BackoutRequeueQName) attribute. The application can use MQINQ to inquire about these attributes and then compare the value in the BackoutCount of the message descriptor. If greater than the threshold, the application can reroute the message to the queue named in the BackoutRequeueQName. The queue manager takes no action on its own when the threshold is met during program processing.

Mark skip backout (z/OS only)

```

MQGET using MQGMO_MARK_SKIP_BACKOUT
DB update 1
DB update 2

If error...
back out unit of work, including DB updates

```

Original message available in new transaction

```

MQPUT reply message
SYNCPOINT

```

[Committing and backing out units of work](#)

© Copyright IBM Corporation 2017

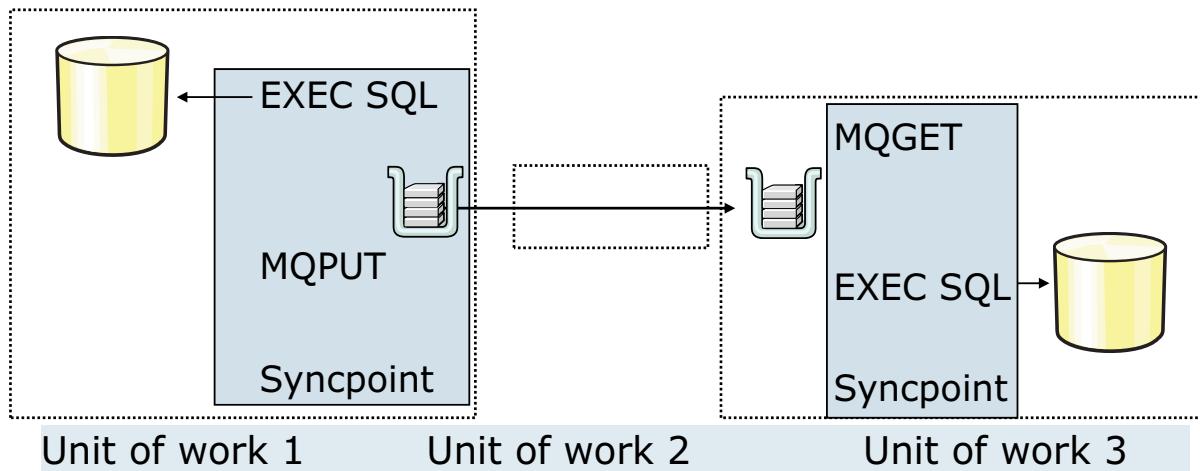
Figure 7-14. Mark skip backout (z/OS only)

On z/OS, it is possible to avoid rolling back a message that contains poisoned data. If an input queue holds the potential for containing messages with bad data, use the get message option of MQGMO_MARK_SKIP_BACKOUT. Only one message per logical unit of work can specify this option.

If the program determines a rollback is required, the message that the MQGMO_MARK_SKIP_BACKOUT retrieves is not made available on the input queue. Also, the message does not have a delete flag that is turned off. Rather, all other resources that are involved in the unit of work are rolled back, leaving the poisoned message as the sole participant in its own unit of work.

It would now be possible to use an MQPUT in syncpoint to place that message on some type of error queue and then to issue a commit to delete it from the input queue. This action makes it available for exception processing on the error queue.

Remote updates



Local syncpoint participation = committed changes

[Committing and backing out units of work](#)

© Copyright IBM Corporation 2017

Figure 7-15. Remote updates

When IBM MQ applications are distributed across more than one platform, time independence means that a traditional distributed unit of work does not exist.

In addition, the message channel agents use unit-of-work processing for assured delivery. If that message is placed on a transmission queue and no commit occurs to release it, the MCA cannot get the message off the queue to send it across the channel.

The message is successfully delivered and the expected response is returned. In cases when this action does not occur, the application must include some exception handling. This type of exception handling is called *compensating transactions*. For example, in an airline reservation system, consider the case when the last seat on a plane is offered to a customer. The confirmation is sent over to the airline system, but the application is informed that the seat is already gone. You must offer an alternative to the customer to compensate for the one you thought you might get.

If *all* work uses unit-of-work processing (assuming the configuration that is shown in the figure), what is the *minimum* number of units of work needed to send a message and receive a response?

Coordination choices

- AIX, HP-UX, Solaris
 - IBM TXSeries
 - Oracle Tuxedo
 - WebSphere Application Server
- Windows
 - WebSphere Application Server
 - IBM TXSeries
 - Oracle Tuxedo
 - Windows COM+/MTS
- Linux
 - Oracle Tuxedo
 - WebSphere Application Server
- IBM i
 - IBM CICS Transaction Server
 - WebSphere Application Server
 - Windows COM+/MTS

[Committing and backing out units of work](#)

© Copyright IBM Corporation 2017

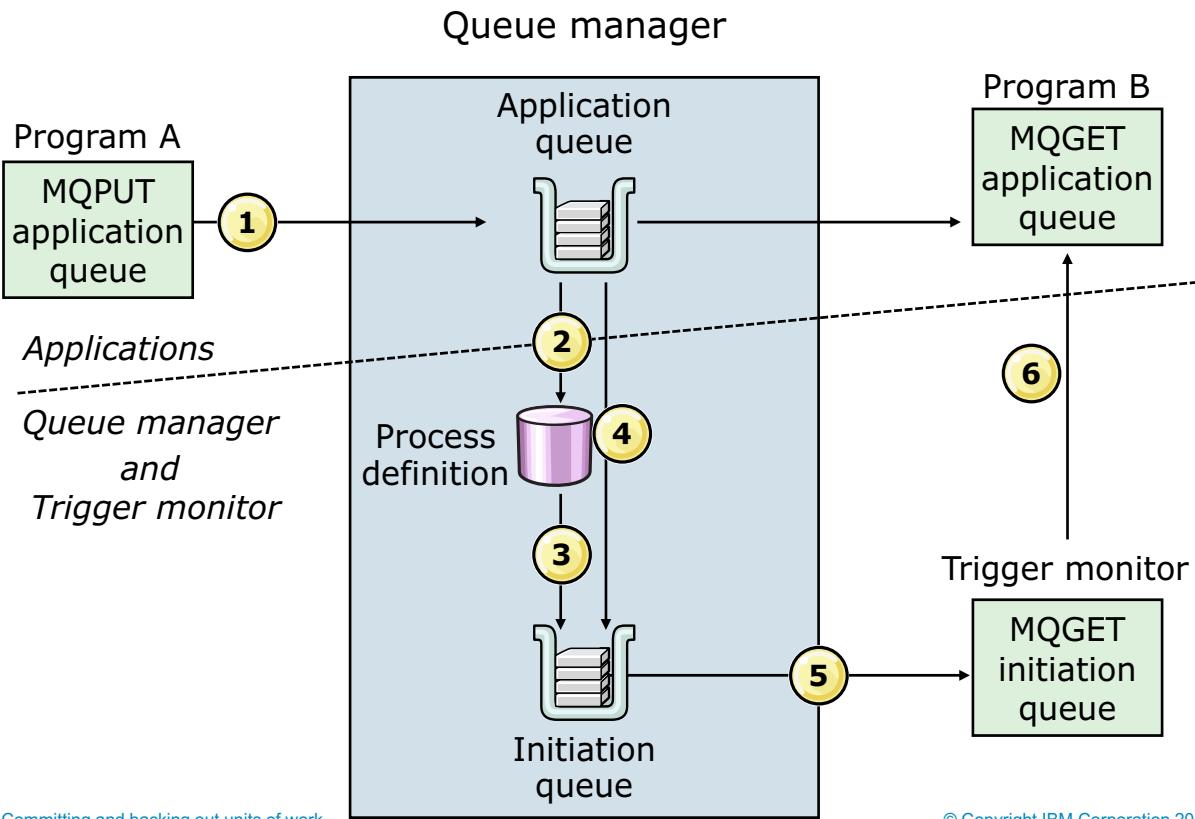
Figure 7-16. Coordination choices

As of the time that this course was written, the cooperating XA-compliant transaction coordination products that IBM MQ supports are listed in the figure.

For the newest information and versions, you should check the IBM Knowledge Center for IBM MQ. As of the time that this course was written, the link for the IBM Knowledge Center for IBM MQ is: https://www.ibm.com/support/knowledgecenter/SSFKSJ_9.0.0.

If the link becomes outdated, use your favorite engine, and search for “IBM Knowledge Center IBM MQ”.

Triggering



Committing and backing out units of work

© Copyright IBM Corporation 2017

Figure 7-17. Triggering

In the figure, what happens above the dotted line is application code. The actions below the dotted line are the responsibility of the queue manager and a supplied utility that is called a trigger monitor.

Program A MQPUTs a message to the application queue.

The queue manager recognizes that the application queue is defined as a triggered queue and references the process that is specified in the queue definition.

Using information from the queue definition (name of queue, name of process definition, name of initiation queue) and from the process definition (name of application to start), the queue manager builds a special trigger message.

The trigger message is placed on the initiation queue.

A constantly running application that is called a trigger monitor retrieves the trigger message and obtains the name of the application to start from within that message.

The trigger monitor starts the program (Program B in this example), passing it the name of the application queue.

Trigger types

- First
 - TriggerMsgPriority
- Every
 - TriggerMsgPriority
- Depth
 - TriggerDepth
 - TriggerMsgPriority

[Committing and backing out units of work](#)

© Copyright IBM Corporation 2017

Figure 7-18. Trigger types

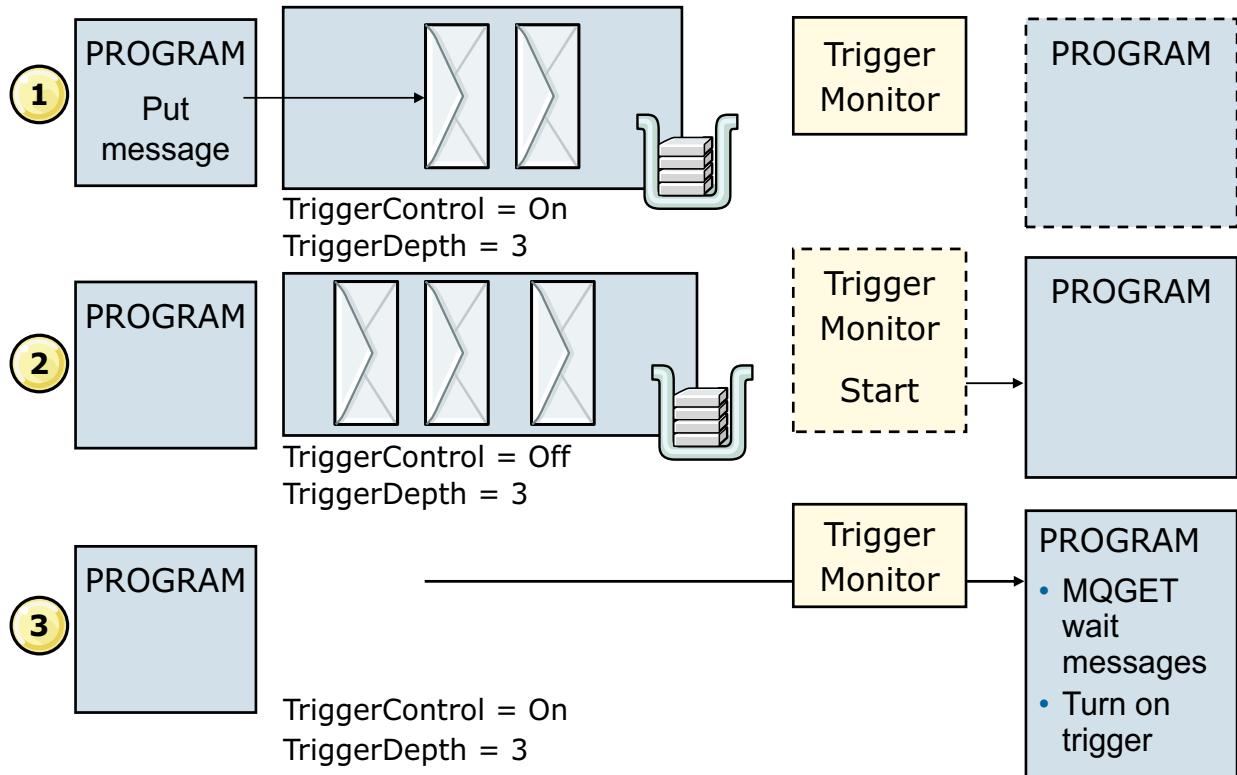
The TRIGTYPE parameter of the triggered queue defines the circumstances in which the triggered program is started.

It is possible to set up a queue for triggering when:

- The first message arrives (TRIGTYPE=FIRST)
- Some fixed number of messages arrives (TRIGTYPE=DEPTH, TRIGDEPTH=3)
- Whenever a new message arrives (TRIGTYPE=EVERY)

The conditions for triggering are set up as part of the queue definition, so the IBM MQ administrator must be involved in any design that requires triggering.

Trigger type depth



Committing and backing out units of work

© Copyright IBM Corporation 2017

Figure 7-19. Trigger type depth

As shown in the figure, when the trigger conditions are satisfied on the triggered queue, the queue manager creates a trigger message and disables triggering on that queue by using the **TriggerControl** attribute. The triggered application must issue the MQSET call to re-enable triggering.

To avoid an up and down effect, set the WAITINTERVAL to see whether more triggering conditions arrive before turning on the trigger and exiting.

MQTMC2 (trigger message)

```

MQCHAR4      StrucId;      /* Structure identifier      */
MQCHAR4      Version;      /* Structure version number*/
MQCHAR48     QName;        /* Name of triggered queue */
MQCHAR48     ProcessName; /* Name of process object */
MQCHAR64     TriggerData; /* Trigger data             */
MQCHAR4      ApplType;    /* Application type         */
MQCHAR256    ApplId;      /* Application identifier   */
MQCHAR128    EnvData;     /* Environment data         */
MQCHAR128    UserData;    /* User data                */
/* Ver:1 */
MQCHAR48     QMgrName;    /* Queue manager name       */
/* Ver:2 */

```

[Committing and backing out units of work](#)

© Copyright IBM Corporation 2017

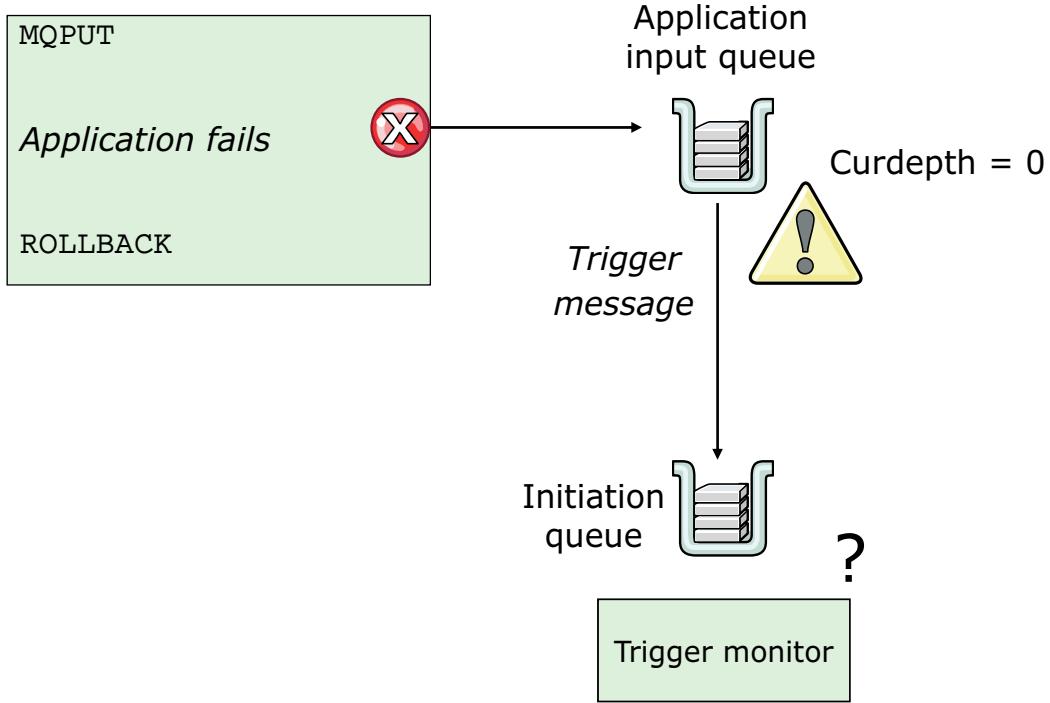
Figure 7-20. MQTMC2 (trigger message)

When a trigger monitor retrieves a trigger message (MQTM) from an initiation queue, the trigger monitor might need to pass information in the trigger message to the application that the trigger monitor starts. Information that the started application might need includes **QName**, **TriggerData**, and **UserData**. The trigger monitor application can pass the MQTM structure directly to the started application.

By passing the **QName** to the triggered program, you can design a generic program that different queues can trigger. The triggered program retrieves the name of the queue from the trigger message and uses that in the **ObjectName** field in the object descriptor for the MQOPEN call. The **TriggerData** and **UserData** fields give you different ways of passing information from the queue to the triggered program.

The three different trigger message formats are MQTM, MQTMC, and MQTMC2. All contain the fields that are listed on the figure. MQTM defines the version and **ApplType** fields as MQLONG, while the MQTMC defines those fields as characters. MQTMC2 is the same definition as MQTMC, though the version value is 2 and it has one more field, **QMgrName**. Only IBM i uses MQTMC. Any platform can use MQTM and MQTMC2.

Triggering and syncpoint



Committing and backing out units of work

© Copyright IBM Corporation 2017

Figure 7-21. Triggering and syncpoint

If a message is physically placed on a queue, it has the potential to satisfy trigger conditions that are associated with the queue. For example, assume that a message is put in syncpoint and is the first message that is placed on the queue. If TRIGTYPE=FIRST and triggering is enabled for the queue, the queue manager builds a trigger message and places it on the appropriate initiation queue. However, the trigger message is also put in syncpoint.

This action can delay the start of an application to process the messages, which might be what is intended. But consider a queue that has a mix of messages, some of which are put in syncpoint and some without syncpoint. The subsequent messages that are put without syncpoint are not processed until the trigger message is made available for the trigger monitor to receive.

When any syncpoint occurs, whether a commit or rollback, the message in the initiation queue is committed and made available to the trigger monitor. This action might cause an application to start and find no messages available (if they were all rolled back).

Unit summary

- Describe the terminology that is associated with committing and backing out units of work
- Differentiate between local and global units of work
- Describe how syncpoint control is implemented in IBM MQ
- Describe the syntax and use of the MQBEGIN, MQCMIT, and MQBACK function calls
- Explain how to use triggering to start an application
- Describe considerations to observe when using triggering and syncpoint in the same application

[Committing and backing out units of work](#)

© Copyright IBM Corporation 2017

Figure 7-22. Unit summary

Review questions

1. What are the two types of units of work in IBM MQ?
 - a. Logical and physical
 - b. Local and global
 - c. Persistent and non-persistent
2. How can you tell how many times a message was backed out?
 - a. *BackoutCount* attribute on the message descriptor
 - b. *BackoutCount* attribute on the queue
 - c. *BackoutCount* attribute on the queue manager
3. What are the three types of triggering?
 - a. Manual, automatic, and coordinated
 - b. File, database, and queue
 - c. Every, first, and depth



Committing and backing out units of work

© Copyright IBM Corporation 2017

Figure 7-23. Review questions

Write your answers here:

- 1.
- 2.
- 3.

Review answers

1. What are the two types of units of work in IBM MQ?

- a. Logical and physical
- b. Local and global
- c. Persistent and non-persistent

The answer is: b, Local and global.



2. How can you tell how many times a message was backed out?

- a. *BackoutCount* attribute on the message descriptor
- b. *BackoutCount* attribute on the queue
- c. *BackoutCount* attribute on the queue manager

The answer is: a, BackoutCount on the message descriptor.

3. What are the three types of triggering?

- a. Manual, automatic, and coordinated
- b. File, database, and queue
- c. Every, first, and depth

The answer is: c, Every, first, and depth.

Committing and backing out units of work

© Copyright IBM Corporation 2017

Figure 7-24. Review answers

Commit and back out review

Committing and backing out units of work

© Copyright IBM Corporation 2017

Figure 7-25. Commit and back out review

Exercise objectives

- Locate an example of a commit and back out application in the IBM MQ installation
- Include the back out structure declaration and initialization in your application
- Code the MQBEGIN function call
- Design and code the flow of tasks that are required for a commit or back out application
- Code the MQCMIT and MQBACK function calls



Committing and backing out units of work

© Copyright IBM Corporation 2017

Figure 7-26. Exercise objectives

Unit 8. Asynchronous messaging

Estimated time

00:30

Overview

In this unit, you learn about IBM MQ callback and control functions that enable delivery of messages to a “unit of code” or module for consumption. This unit uses asynchronous messaging, which is accomplished with the IBM MQ MQCB and MQCTL function calls.

How you will check your progress

Accountability:

- Review questions
- Lab Exercises

References

IBM Knowledge Center for IBM MQ V9 documentation

Unit objectives

- Describe the concept and use cases for asynchronous messaging
- Explain the parameters and operation options of the IBM MQ callback (MQCB) function call
- Identify the fields in the callback data descriptor (MQCBD) structure
- Describe the parameters and use of the IBM MQ control (MQCTL) function call
- Identify the fields in the callback context (MQCBC) structure
- Differentiate between the context of the MQCB_FUNCTION and the MQCB function call

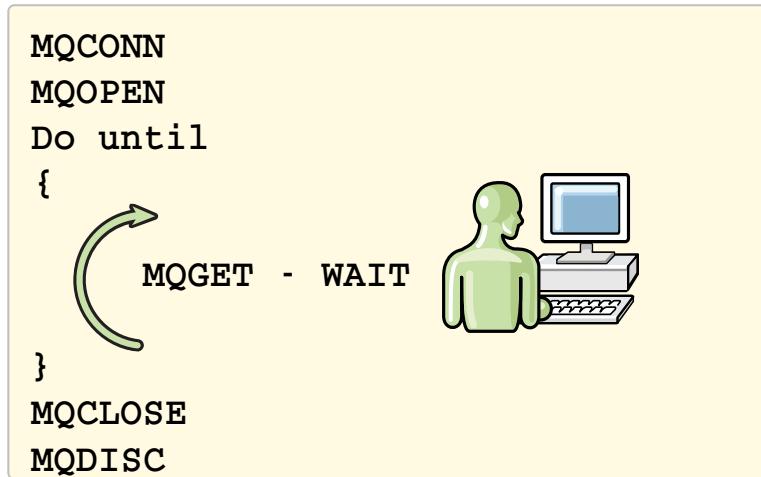
Asynchronous messaging

© Copyright IBM Corporation 2017

Figure 8-1. Unit objectives

Using MQGET for traditional message retrieval

- Waiting for message arrival
- Application thread is blocked



Asynchronous messaging

© Copyright IBM Corporation 2017

Figure 8-2. Using MQGET for traditional message retrieval

The figure shows the traditional way of retrieving messages where an application thread waits for the arrival of the message in the MQGET call. The application designer must fine-tune the time interval for which the MQGET waits for the arrival of a message.

Asynchronous message consumer

- Function or routine that starts with the arrival of a message
 - Message listener is started whenever a new message is available on the queue
 - Can provide asynchronous notification of events such as connection termination or queue manager quiescing
- No application threads are busy waiting for a message
- Message is delivered in a buffer so the application does not need to allocate a buffer on its own for holding the data that is passed in the message

[Asynchronous messaging](#)

© Copyright IBM Corporation 2017

Figure 8-3. Asynchronous message consumer

Asynchronous message consumption provides a way to process messages when they arrive on the queue. *Asynchronous message consumption* should not be confused with *triggering*, where a process is started upon the arrival of messages on a *triggered* queue. *Asynchronous message consumption* uses specific IBM MQ call functions to accomplish the task of starting the application or segment of code that obtains and processes the messages.

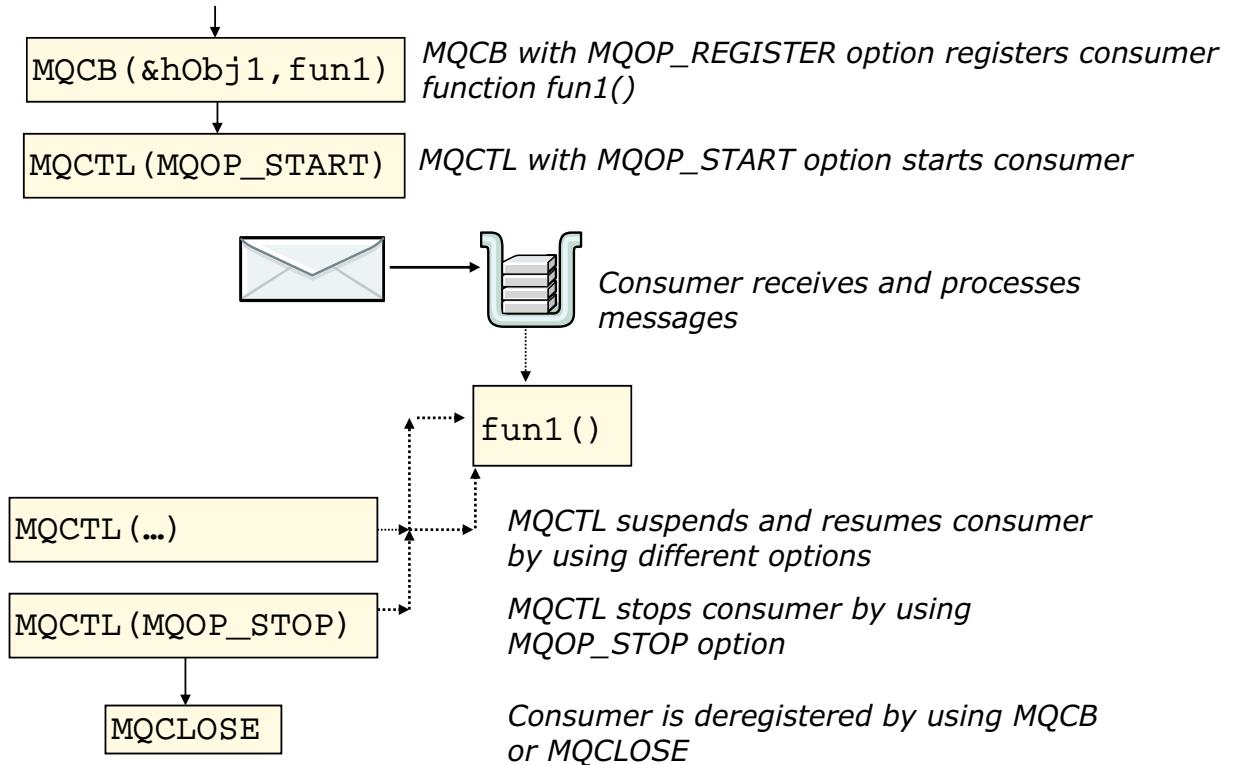
An *asynchronous consumer* is the function or routine that is called when a message arrives, or an event takes place. The application data that is embedded in the message is passed in a buffer as an input parameter to the asynchronous message consumer. The length of this buffer is also passed as an input parameter so that it is not necessary for the application to allocate or reserve a buffer on its own to receive the message. The process eliminates the need to guess the message size while allocating the buffer and simplifies the application design.

Asynchronous message consumption allows applications to continue processing instead of waiting for messages to arrive. The arrival of a message calls a specified user-defined function to process this message. Since the arrival of the message drives the call to this message processing function and is asynchronous in nature, this method of getting messages is called *asynchronous message consumption*.

Another feature of asynchronous message consumption is that applications can also be notified of the occurrence of certain events like connection termination or queue manager quiescing when

these events occur. Therefore, event handlers that process user-defined operations whenever an event occurs can also be called by using asynchronous message consumption.

Lifecycle of a message consumer



Asynchronous messaging

© Copyright IBM Corporation 2017

Figure 8-4. Lifecycle of a message consumer

A message consumer function type is called when a message is available on a queue or a topic and that message meets the selection criteria that are specified for the message consumer.

The message consumer uses the MQI call MQCB to register. Several message consumers can be registered for different queues in the same application thread by making multiple calls to MQCB by using different object handles. The MQCTL call with the option MQOP_START starts the message consumer. The message consumer is started when a new message that meets the specified selection criteria arrives at the queue. The message consumer runs either in the same application thread or a newly allocated thread that depends on the threading mode used. The MQCTL or MQCB calls with the options MQOP_SUSPEND and MQOP_RESUME to control the state of the message consumer.

The MQCTL call with the MQOP_STOP option stops the message consumer. Finally, the message consumer explicitly removes the registration with the MQCTL call and the MQOP_DEREGISTER option, or implicitly with the MQCLOSE call.

Message consumer types

- A function or an entry point in the program
- An external program
 - On Windows, a function in a dynamic linked library
 - On UNIX systems, a function in a dynamically loadable module or library
 - On z/OS, name of a load module (valid for LINK or LOAD macro)

Asynchronous messaging

© Copyright IBM Corporation 2017

Figure 8-5. Message consumer types

The message consumer can be a module of instructions of different types. This application depends on the programming environment. The slide lists different types of message consumers.

MQCB: Manage callback

- Reregisters a callback for the specified object handle, controls activation and changes to the callback
- Message consumer callback function is called when a message that meets the selection criteria that are specified is available on an object handle
- Only one callback function can be registered against each object handle
 - If a single queue is read with multiple selection criteria, then the queue must be opened multiple times and a consumer function must be registered on each handle

[Asynchronous messaging](#)

© Copyright IBM Corporation 2017

Figure 8-6. MQCB: Manage callback

The MQCB call registers a callback for the specified object handle and controls activation and changes to the callback.

A callback is a piece of code (specified as either the name of a function that can be dynamically linked or as function pointer) that IBM MQ calls when certain events occur.

You can configure a consumer to start callback at key points during the lifecycle of the consumer, for example:

- When the consumer is first registered
- When the connection is started
- When the connection is stopped
- When the consumer is deregistered, either explicitly or implicitly, by an MQCLOSE

To use MQCB on a IBM MQ V7 client, you must be connected to a V7 or later server, and the SHARECNV parameter of the channel must have a nonzero value.

MQCB function

```

MQCBD cbd = {MQCBD_DEFAULT}; /*Callback Descriptor*/
MQCTLO ctlo = {MQCTLO_DEFAULT};/* Control Options */
cbd.CallbackFunction = MessageConsumer;
gmo.Options = MQGMO_NO_SYNCPOINT;
MQCB (Hcon,      ➔
      Operation, ➔ field replaced by value such as MQOP_REGISTER
      &cbd,        ➔
      Hobj,        ➔
      &md,         ➔
      &gmo,         ➔
      &CompCode,    ➔
      &Reason);    ➔

```

Asynchronous messaging

© Copyright IBM Corporation 2017

Figure 8-7. MQCB function

The MQCB function call registers a callback function for a specified object handle (queue or topic). The callback function can be dynamically linked or specified as a function pointer.

The MQCB function uses the parameters that are shown:

- Operation specifies the operation to start and is sent as input.
- CallbackDesc describes the callback function that is registered and is passed as input.
- MsgDesc defines the attributes of the messages and is sent as input. MsgDesc is not needed for event handlers.
- GetMsgOpts is sent as input to control how messages are obtained and is not required for event handlers.
- The last two items are output fields completion code and associated reason.

MQCB operations

- **MQOP_REGISTER**

Define the callback function for the specified object handle

- **MQOP_DEREGISTER**

Stop consuming messages for the object handle and remove the handle from the ones that are eligible for a callback

- **MQOP_SUSPEND**

Suspend consuming messages for the object handle

- **MQOP_RESUME**

Resume consuming messages for the object handle

The figure lists the valid message callback function operations. You specify at least one of the operations listed. If more than one option is required, the values can be added or combined by using the bitwise OR operation, if the programming language supports it.

Do not add the same constant more than one time.

MQCBD: Callback data descriptor

MQCHAR4	StrucId;	/* Structure identifier */
MQLONG	Version;	/* Structure version nbr */
MQLONG	CallbackType;	/* Callback function type*/
MQLONG	Options;	/* Callback options */
MQPTR	CallbackArea;	/* User data passed */
MQPTR	CallbackFunction; /* CB function pointer */	
MQCHAR128	CallbackName;	/* Callback name */
MQLONG	MaxMsgLength;	/* Maximum mesg length */

```
*****  
/* FUNCTION: MessageConsumer <===(MQCB_FUNCTION) */  
/* Callback function called when messages arrive */  
*****
```

Asynchronous messaging

© Copyright IBM Corporation 2017

Figure 8-9. MQCBD: Callback data descriptor

The application uses the `MQCBD` structure to specify the callback function, its nature, its state, and other details. This structure is used for registering and deregistering the callback function and is also used to change its state. This structure is the principal way of controlling the callback function from the application.

Before the `MQCB` function is called for the callback function in the display, which is called `MessageConsumer`, the callback descriptor structure, or `MQCBD`, represented by `cbd.CallbackFunction`, is set as shown:

```
cbd.CallbackFunction = MessageConsumer;
```

The fields in the `MQCBD` structure are:

- `StrucID` is the structure identifier.
- `Version` is the version. The current version is 1.
- `CallbackType` identifies the type of callback function. Two callback types are available:
 - `MQCBCT_MESSAGE_CONSUMER`
 - `MQCBCT_EVENT_HANDLER`
- `Options` specify the state of the consumer. The allowable states are:

- MQCBDO_REGISTER_CALL
 - MQCBDO_START_CALL
 - MQCBDO_START_CALL
 - MQCBDO_DEREGISTER_CALL
 - MQCBDO_EVENT_CALL
 - MQCBCT_MC_EVENT_CALL (multicasting only)
- MaxMsgLength specifies the length in bytes of the longest message that can be read from the object handle and given to the callback routine. The value MQCBD_FULL_MSG_LENGTH can be used to specify that messages should be delivered without truncation.
 - CallbackFunction is a pointer to the callback function.
 - CallbackName is the name of the function that is started as the dynamically linked program.
 - CallbackArea is an optional field, which can be a pointer to a memory area that the function can use where MQCB is called to communicate with the callback function.

You must specify either CallbackFunction or CallbackName. If you specify both, the reason code MQRC_CALLBACK_ROUTINE_ERROR is returned.

If CallbackName or CallbackFunction is not set, the call fails with the reason code MQRC_CALLBACK_ROUTINE_ERROR.

MQCTL control callbacks

- Controls actions on callbacks and object handles that are opened for a connection

```
MQCBD cbd = {MQCBD_DEFAULT}; /*Callback Descriptor*/
MQCTLO ctlo= {MQCTLO_DEFAULT};/* Control Options */
```

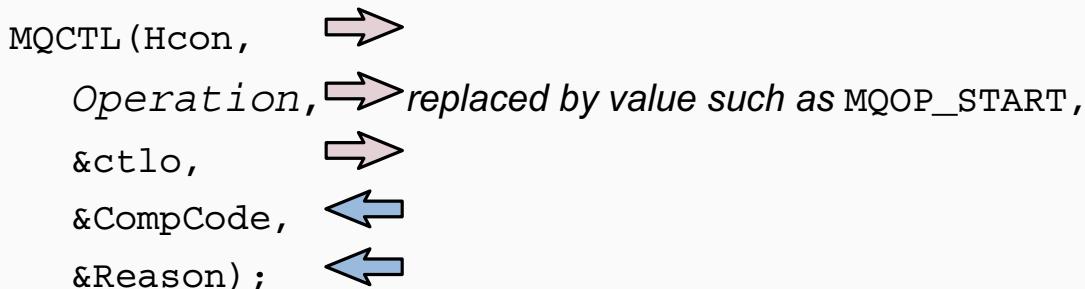


Figure 8-10. MQCTL control callbacks

`MQCTL` is used to control the functioning of the callback functions. `MQCTL` determines when the callback functions are ready to start. It is also used to change the state of the callback functions. `MQCTL` starts asynchronous message consumption or event handling by changing the callback function to the “start” state. It is generally called to stop the asynchronous processing at the end of the application before closing the connection and object handles. This function is used for controlling the state of the consumer by using operations like START, STOP, SUSPEND, and RESUME.

`MQCTL` takes these parameters. The correspondence of the notes and the slide is positional:

- `Hcon` is the connection handle to the queue manager that is sent as input.
- `Operation` is an input field that describes the action that `MQCTL` is processing. Values are:
 - `MQOP_START`
 - `MQOP_START_WAIT`
 - `MQOP_STOP`
 - `MQOP_SUSPEND`
 - `MQOP_RESUME`
- `ControlOpts` is used to control the operations that the `MQCTL` call specifies. Supported values are:

- `MQCTLQ_FAIL_IF QUIESCING`: This option forces the `MQCTL` call to fail if the queue manager or connection is in the quiescing state. With this option, you can specify `MQGMO_FAIL_IF QUIESCING` in the `MQGMO` options that are passed on the `MQCB` call to cause notification to message consumers when they are quiescing.
- `MQCTLQ_THREAD_AFFINITY`: This option informs the system that the application requires that all message consumers, for the same connection, are called on the same thread. This thread is used for all invocations of the consumers until the connection is stopped.
 - If you do not require the previous options, use `MQCTLQ_NONE`.
- The last two parameters are the output completion code and associated reason code.

MQCTL operations

- **MQOP_START**
Start consuming messages for all defined message consumer functions for the specified connection handle
- **MQOP_START_WAIT**
Start consuming messages for all defined message consumer functions for the specified connection handle
- **MQOP_STOP**
Stop consuming messages, and wait for all consumers to complete their operations before this option completes
- **MQOP_SUSPEND**
Pause consuming messages
- **MQOP_RESUME**
Resume consuming messages

[Asynchronous messaging](#)

© Copyright IBM Corporation 2017

Figure 8-11. MQCTL operations

The `MQCTL Operations` attribute identifies the action that is processed on the callback that is defined for the specified object handle. Specify one option.

The choice of start options results in one of two threading modes.

- When `MQOP_START_WAIT` is used, the queue manager takes the application thread, and the application program has to wait until all the callback functions are deregistered and ended.
- When `MQOP_START` is used, the application can continue doing other non-MQ function, or an IBM MQ function that uses a different connection handle while the callback is running.

MQCB_FUNCTION: Callback function

- Callback function for asynchronous message consumption
- Specification of actual function is an input to the MQCB call and is passed in through the MQCBD structure
- *Call direction is from queue manager to callback function application*

```
/*********************************************
/*FUNCTION:MessageConsumer (generic MQCB_FUNCTION) */
/* Callback function called when messages arrives */
/*********************************************
void MessageConsumer(MQHCONN hConn,
                      MQMD * pMsgDesc,
                      MQGMO * pGetMsgOpts,
                      MQBYTE * Buffer,
                      MQCBC * pContext)
```

Asynchronous messaging

© Copyright IBM Corporation 2017

Figure 8-12. MQCB_FUNCTION: Callback function

The application uses the callback function to process the message that is retrieved from the queue. The callback function can run any MQI call except for `MQCTL` with operation `MQCTL_START_WAIT` and `MQDISC`. The application data that is embedded in the message is passed as an input parameter to the callback function by the queue manager. The application does not have to reserve a buffer for storing the retrieved message since the queue manager is handling the process.

Before the `MQCB` function is called for the callback function in the display, which is called `MessageConsumer`, the callback descriptor structure, or `MQCBD`, represented as `cbd.CallbackFunction` is set as shown:

```
cbd.CallbackFunction = MessageConsumer;
```

The callback function, `MessageConsumer`, is the function that is registered and called when messages are available on the associated object handle or events are detected.

Only one callback function is allowed to be registered per object handle by using the same queue manager connection handle. If a second callback function is registered for the same object and connection handle, it replaces the previous registration. Messages are processed one at a time to preserve the specified order (priority or first-in first-out) in the `MQGMO` options in the `MQCB` function call.

It is possible to register multiple callback functions for the same object handle by using different queue manager connection handles. Messages are processed in parallel in this case, and sequence is not assured.

In this slide, the direction of the interface is from queue manager to the callback function. The parameters that are provided by using pointers are:

- `MsgDesc` defines the attributes of the message whose arrival started the invocation of the callback function.
- `GetMsgOpts` controls how the messages are retrieved from the queue.
- `Buffer` is the area where the application data in the message is stored.
- `Context` is the data structure that stores context information for the callback functions.

MQCBC: Callback context

```

MQCHAR4  StrucId;      /* Structure identifier      */
MQLONG    Version;       /* Structure version number */
MQLONG    CallType;     /* Why Function was called */
MQHOBJ   Hobj;         /* Object Handle           */
MQPTR    CallbackArea; /* CB data passed to function*/
MQPTR    ConnectionArea; /*MQCTL data to function */
MQLONG   CompCode;      /* Completion Code          */
MQLONG   Reason;        /* Reason Code              */
MQLONG   State;         /* Consumer State           */
MQLONG   DataLength;    /* Message Data Length      */
MQLONG   BufferLength;  /* Buffer Length             */
MQLONG   Flags;         /* Consumer info flags      */
/* Ver:1 */
MQLONG   ReconnectDelay; /* Millisecs before reconn */
/* Ver:2 */

```

Asynchronous messaging

© Copyright IBM Corporation 2017

Figure 8-13. MQCBC: Callback context

MQCBC is passed to the callback functions as context information. The display provides a summary of the fields of the MQCBC structure:

- StructID and Version as standard in most structures.
- CallType provides the information about why this function was called. The values are:
 - MQCBCT_REGISTER_CALL
 - MQCBCT_START_CALL
 - MQCBCT_STOP_CALL
 - MQCBCT_DEREGISTER_CALL
 - MQCBCT_MSG_REMOVED
 - MQCBCT_MSG_NOT_REMOVED
 - MQCBCT_EVENT_CALL
- State provides information about the state of the current consumer. It is normally used when a nonzero reason code is passed.
- The DataLength field is used by the queue manager to return the length of the message at run time, which eliminates the need for the application to guess this value.

- The `CallbackArea` is saved and returned unchanged by the queue manager. The callback function can use it to store state information that is preserved across invocations of the callback function.
- The `ConnectionArea` field is also saved and returned unchanged by the queue manager. The callback function can also use the `ConnectionArea` to store state information across function calls.

MQCBC call types

- **MQCBCT_MSG_REMOVED**

Message consumer function that is called with a message destructively removed from the object

- **MQCBCT_MSG_NOT_REMOVED**

Message consumer function that is called with a message that is not destructively removed from the object handle

- **MQCBCT_REGISTER_CALL**

Allow the callback function to do some initial setup

- **MQCBCT_START_CALL**

Allow the callback function to do some setup when it is started

- **MQCBCT_STOP_CALL**

Allow the callback function to do some cleanup when it is stopped for a while

- **MQCBCT_DEREGISTER_CALL**

Allow the callback function to do final cleanup at the end of the consume process

- **MQCBCT_EVENT_CALL**

Message consumer function that is started without a message when an error is detected that is specific to the object handle

Figure 8-14. MQCBC call types

The `CallType` field contains information about why this function is called. The display lists the valid callback context `CallType` values within categories.

- Message delivery call types contain information about a message. The `DataLength` and `BufferLength` parameters are valid for these call types. Message delivery call types are:

- `MQCBCT_MSG_REMOVED`
- `MQCBCT_MSG_NOT_REMOVED`

- Callback control call types contain information about the control of the callback and do not contain details about a message. These call types are requested by using `Options` in the `MQCBDO` structure. The `DataLength` and `BufferLength` parameters are not valid for these call types. Callback control call types are:

- `MQCBCT_REGISTER_CALL`
- `MQCBCT_START_CALL`
- `MQCBCT_STOP_CALL`
- `MQCBCT_DEREGISTER_CALL`
- `MQCBCT_EVENT_CALL`

An extra call type exists, which is `MQCBCT_MC_EVENT_CALL`. This call type is used for multicast programming.

Unit summary

- Describe the concept and use cases for asynchronous messaging
- Explain the parameters and operation options of the IBM MQ callback (MQCB) function call
- Identify the fields in the callback data descriptor (MQCBD) structure
- Describe the parameters and use of the IBM MQ control (MQCTL) function call
- Identify the fields in the callback context (MQCBC) structure
- Differentiate between the context of the MQCB_FUNCTION and the MQCB function call

Asynchronous messaging

© Copyright IBM Corporation 2017

Figure 8-15. Unit summary

Review questions

1. Which two MQI calls are used for asynchronous message consumption?
 - a. MQCB and MQCBC
 - b. MQCBC and MQCBD
 - c. MQCB and MQCTL
2. True or False: Multiple callback functions can be registered by using the same object handle.
3. Which option is used with MQCTL for using the application thread to do the callback dispatch?
4. Which MQI calls cannot be initiated inside the callback function?
5. Where is the message length of the retrieved message stored?



Asynchronous messaging

© Copyright IBM Corporation 2017

Figure 8-16. Review questions

Write your answers here:

- 1.
- 2.
- 3.
- 4.
- 5.

Review answers (1 of 2)

1. Which two MQI calls are used for asynchronous message consumption?

- a. MQCB and MQCBC
- b. MQCBC and MQCBD
- c. MQCB and MQCTL

The answer is: c. MQCB and MQCTL.

2. True or False: Multiple callback functions be registered by using the same object handle.

The answer is: False. However, multiple callback functions can be registered by using the same connection handle.

3. Which option is used with MQCTL for using the application thread to do the callback dispatch?

The answer is: MQOP_START_WAIT.

Review answers (2 of 2)

4. Which MQI calls cannot be initiated inside the callback function?

The answer is: MQCTL with option MQOP_START_WAIT and MQDISC.



5. Where is the message length of the retrieved message stored?

The answer is: In the DataLength field of the MQCBC structure that is passed to the callback as an argument.

Asynchronous messaging review

Asynchronous messaging

© Copyright IBM Corporation 2017

Figure 8-19. Asynchronous messaging review

Exercise objectives

- Describe the mechanics and component exchanges of asynchronous messaging applications
- Explain the role of the MQCB function call and its required parameters
- Differentiate between the IBM MQ MQCB callback function and the application callback module that the MQCB_FUNCTION parameter definition represents
- Describe how to initiate and end consumption of messages with the MQCTL function call
- Explain how to code and exchange information in an application callback function



Asynchronous messaging

© Copyright IBM Corporation 2017

Figure 8-20. Exercise objectives

Unit 9. IBM MQ clients

Estimated time

00:30

Overview

In your development work, you might need to code a program for an IBM MQ client environment. This unit explains the differences between IBM MQ servers and IBM MQ clients, and various ways to connect an IBM MQ client to an IBM MQ server. You also learn the difference to observe when compiling client code.

How you will check your progress

Accountability:

- Review questions
- Lab Exercises

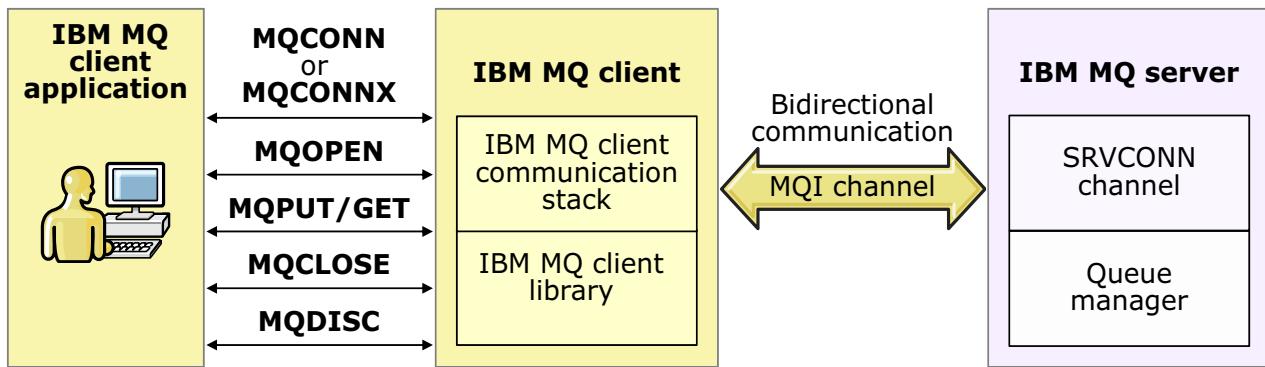
References

IBM Knowledge Center for IBM MQ V9 documentation

Unit objectives

- Describe the differences between an IBM MQ client and an IBM MQ server
- List the supported languages per platform that can be used to code an IBM MQ client application
- Explain how to compile an IBM MQ client application
- Describe the considerations to observe when working with IBM MQ clients
- Describe the use of environment variables or a client channel definition table (CCDT) to connect an IBM MQ client to a queue manager in an IBM MQ server
- Explain how to use the MQCONN call to connect an IBM MQ client application to a queue manager directly from your code
- Differentiate the capabilities that are available with the various client connectivity options
- Describe a redistributable client (V8.0.0.4 and later)
- Summarize the use and precedence order of a CCDT URL

IBM MQ clients



- An IBM MQ client has IBM MQ client libraries that are installed
- No queue managers or queues are defined
- Uses the same MQI calls as the IBM MQ server
- Flow of messages in an MQI channel is bidirectional
- Transactional capabilities

IBM MQ clients

© Copyright IBM Corporation 2017

Figure 9-2. IBM MQ clients

An IBM MQ client is an option that allows installation of a different set of IBM MQ libraries that run in client mode. The client does not have queues or queue managers. An IBM MQ client application and a server queue manager communicate with each other by using an MQI channel. An MQI channel does not use a channel pair, but rather an MQI connection.

An MQI channel starts when the client application issues an MQCONN or MQCONNX call to connect to the queue manager, and ends when the client application issues an MQDISC call to disconnect from the queue manager.



Note

Do not confuse the term client as used in the general architectural mode, with an IBM MQ client. An application that is compiled with IBM MQ server libraries might serve as a client application, but uses distributed channel pairs. An IBM MQ client that is compiled with IBM MQ client libraries uses MQI channels, and must connect to a queue manager SVRCONN channel.

Programming languages supported for clients

Client platform	C	C++	COBOL	pTAL	RPG	Visual Basic
AIX	Yes	Yes	Yes			
HP Integrity NonStop Server	Yes		Yes	Yes		
HP-UX	Yes	Yes	Yes			
IBM	Yes		Yes		Yes	
Linux	Yes	Yes	Yes			
Solaris	Yes	Yes	Yes			
Windows	Yes	Yes	Yes			Yes

IBM MQ clients

© Copyright IBM Corporation 2017

Figure 9-3. Programming languages supported for clients

The IBM MQ client languages available are listed by operating system.

Examples of preparing C client programs

Linux

(IBM MQ server application, simple format)

```
gcc -I/opt/mqm/inc -lmqic -o mqput mqput.c
```

(IBM MQ server application, 64-bit, non-threaded)

```
gcc -m64 -o amqsputc_64 amqsputc0.c -I MQ_INSTALLATION_PATH/inc  
-L MQ_INSTALLATION_PATH/lib64 -Wl,-rpath=MQ_INSTALLATION_PATH/lib64  
-Wl,-rpath=/usr/lib64 -lmqic
```

Windows

- Open a Visual studio command prompt
- Include the C libraries
`SET INCLUDE=C:\Program
Files\IBM\MQ\tools\c\include;%INCLUDE%`
- Change to your source directory

```
cl mqput.c "c:\Program Files\IBM\MQ\tools\Lib\mqic.lib"
```

(With 64-bit libraries)

```
cl -MD amqsget0.c -Feamqsget.exe MQ_INSTALLATION_PATH\Tools\Lib64\mqic.lib
```

Figure 9-4. Examples of preparing C client programs

The main difference when you compile an IBM MQ client application is the link libraries that are used to link; instead of `mqm`, the client uses the `mqic` libraries.

Considerations when building client applications

- The maximum message size attribute of the SVRCONN channel might limit the message length
- Coded character set identifier (CCSID)
 - MQMD `CodedCharSetID` and multiple MQPUT calls
 - Client code assumes CCSID for data from client matches CCSID of the workstation
- Read-ahead
- Syncpoint coordination

Figure 9-5. Considerations when building client applications

Keep in mind the following considerations when building client applications:

- CCSID: When you code a client application, you should use the CCSID of the operating system where your client is hosted and defer any conversion to the queue manager. As in regular server applications, after you do the first MQPUT, the CCSID might become changed.
- Read-ahead: Read-ahead is a client-only capability that allows the client to receive non-persistent messages from the queue manager without having to send a request for the message. Read-ahead is used to improve performance. Use of read-ahead:
 - Has several requirements, which must be met
 - Is available only if both sides of the connection are at IBM MQ V7 or later
 - Might not suit all applications due to option consistency between MQGET calls
- Syncpoint coordination is detailed in a separate slide.

IBM MQ base and extended transactional client

- A IBM MQ transactional client:
 - Is a IBM MQ base client with extra transactional capabilities
 - Is supplied with IBM MQ
 - Is a separate installation for WebSphere MQ V7.5 and earlier
 - The IBM MQ transactional client is available for Linux, UNIX, and Windows platforms
- The differences in transactional capabilities between IBM MQ base and IBM MQ extended transactional client are:

IBM MQ base client	IBM MQ extended transactional client
<ul style="list-style-type: none"> • Can participate in a unit of work that is managed by the queue manager to which it is connected • Able to use MQCMIT or MQBACK to complete the unit of work • Unable to update resources that another resource manager owns, such as a database 	<ul style="list-style-type: none"> • Is able to put to and get messages from the queue manager to which it is connected • Is able to update resources that another resource manager owns in the same unit of work • Unit of work must be managed by an external transaction manager that is collocated with the client application • Uses external transaction manager API, no MQCMIT, or MQBACK

IBM MQ clients

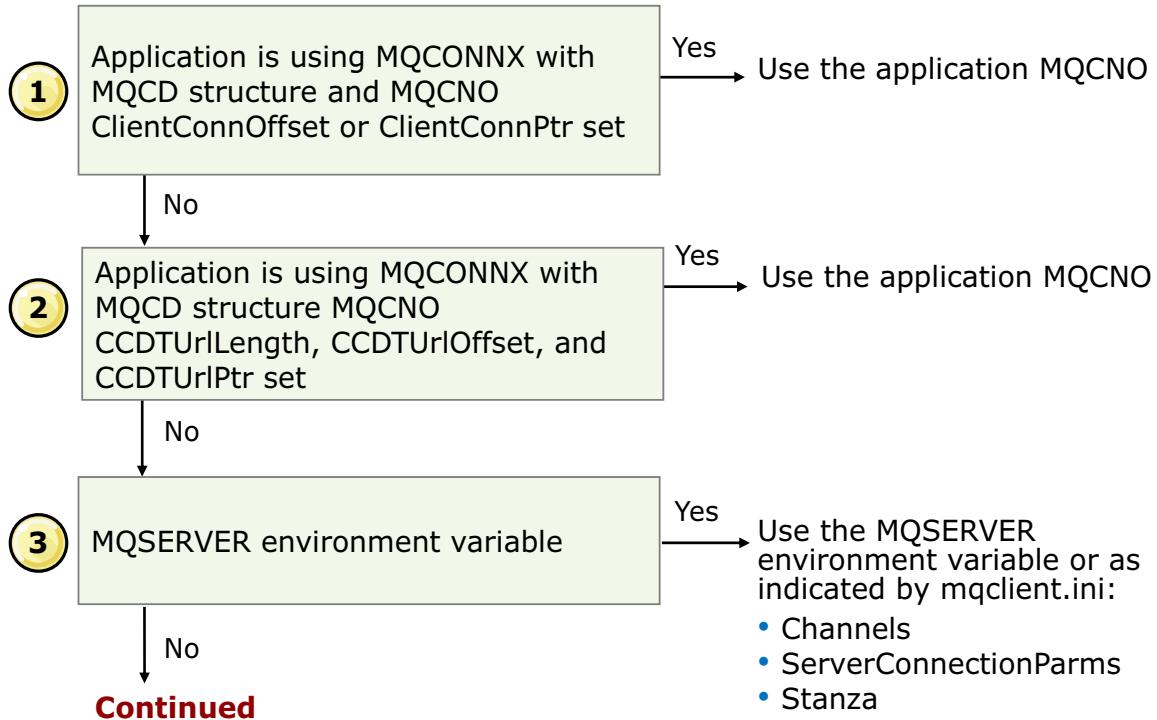
© Copyright IBM Corporation 2017

Figure 9-6. IBM MQ base and extended transactional client

All IBM MQ clients were not created equal, but starting with IBM MQ V8, it is no longer necessary to install a separate client to get the capabilities provided by the transactional client.

The two tables in this slide summarize the differences between the basic IBM MQ client and the IBM MQ Extended Transactional client.

IBM MQ client connectivity options (1 of 2)



IBM MQ clients

© Copyright IBM Corporation 2017

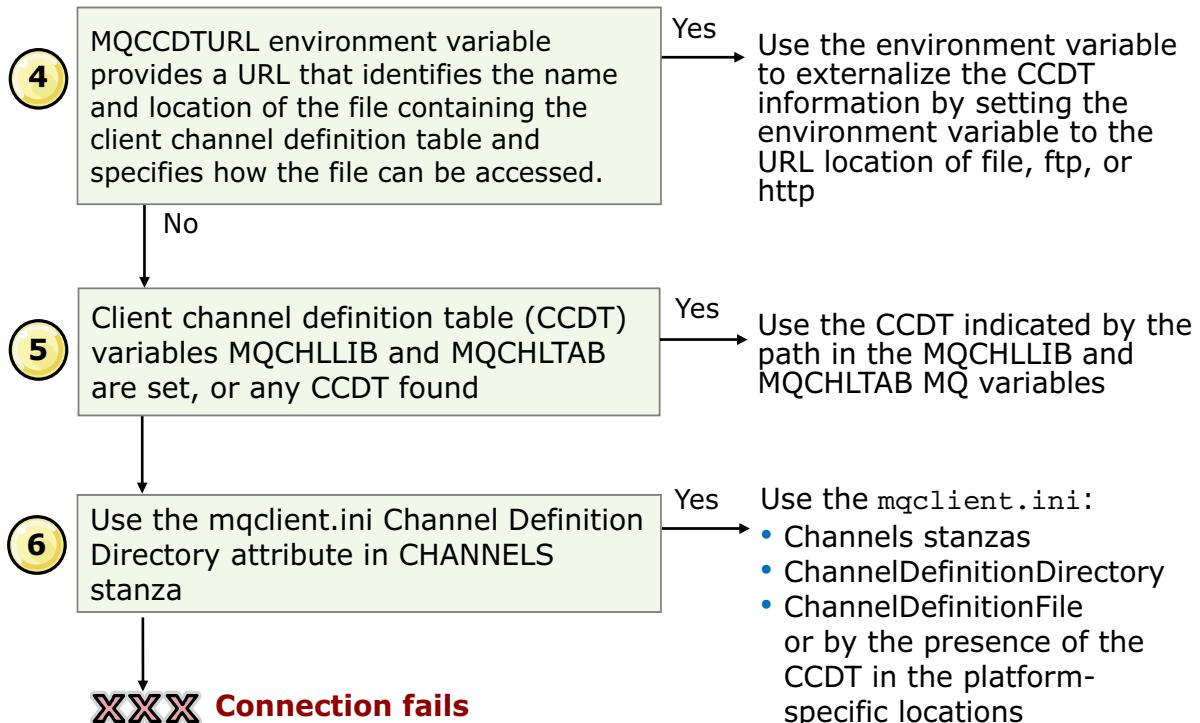
Figure 9-7. IBM MQ client connectivity options (1 of 2)

Multiple methods exist to configure a client connection to obtain a channel definition:

- MQCD provided by using the MQCONN with a MQCD structure and MQCNO connection options:
 - ClientConnOffset or ClientConnPtr or CCDTUILengh, CCDTUrlOffset, and CCDTUrlPtr
- MQSERVER environment variable
- Client channel definition table (CCDT)

If an application provides an MQCD in the MQCNO, the environment variables are ignored. The MQCD structure is detailed later in this unit.

IBM MQ client connectivity options (2 of 2)



IBM MQ clients

© Copyright IBM Corporation 2017

Figure 9-8. IBM MQ client connectivity options (2 of 2)

Shown is a continuation of the multiple methods available to configure a client connection to obtain a channel definition.

You can also use the MQCHLLIB environment variable or the ChannelDefinitionDirectory variable under the CHANNELS stanza of the client configuration file to locate a CCDT file. The MQCHLLIB value is a directory stem and works in combination with MQCHLTAB to derive the fully qualified URL.

URL support for CCDT files

- Supports “`ftp://`”, “`file://`” and “`http://`” protocols
 - `MQCCDTURL` environment variable specifies full URL:
 - `MQCCDTURL=file:///var/mqm/qmgrs/QMGR/@ipcc/AMQCLCHL.TAB`
 - `MQCCDTURL=file:///C:/student/urlccdt/fileurl/AMQCLCHL.TAB`
 - `MQCCDTURL=ftp://example.com//files/MYCHLTAB.TAB`
 - `MQCCDTURL=http://www.example.com/MYFILE.TAB`
 - Basic authentication specifies user password:
 - `MQCCDTURL=ftp://user:password@example.com//files/MYCHLTAB.TAB`
 - `MQCCDTURL=http://user:password@www.example.com/MYFILE.TAB`
 - Alternatively use `MQCHLLIB` to provide the stem of the URL examples:
 - `MQCHLLIB=ftp://user:password@example.com//files`
 - `MQCHLTAB=MYCHLTAB.TAB`

Using an HTTP-style URL



```
SET MQCCDTURL=
  http://195.16.80.3:8080/QASTAGE.TAB
```

IBM MQ clients

© Copyright IBM Corporation 2017

Figure 9-9. URL support for CCDT files

Web addressable access to the client channel definition table (CCDT) improves the ability for clients to remain connected to IBM MQ queue managers by hosting the CCDT in a central location. The CCDT location is accessible through a URI, removing the need to individually update the CCDT for each deployed client.

Shown are examples of using environment variables to specify the path to locate the CCDT.

You can locate a CCDT through a URL, either by programming, by using the `MQCCDTURL` and `MQCHLLIB` environment variables, or by using the `mqclient.ini` file stanzas.

You can use the environment variable option only for C, COBOL, or C++ applications. The environment variables have no effect for Java, JMS, or managed .NET applications.

The `MQCCDTURL` environment variable specifies a file, FTP, or HTTP URL as a single value from which a CCDT can be obtained.

You can also use the `MQCHLLIB` environment variable or the `ChannelDefinitionDirectory` variable under the `CHANNELS` stanza of the client configuration file to locate a CCDT file. The `MQCHLLIB` value is a directory stem and works in combination with `MQCHLTAB` to derive the fully qualified URL.

Precedence order for finding the CCDT

1. Application specifies connection details in MQCONN call
 2. Application specifies CCDT URL location in MQCONN call
 3. MQSERVER environment variable
 4. MQCCDTURL environment variable
 5. MQCHLLIB and MQCHLTAB environment variables
 6. ChannelDefinitionDirectory attribute in CHANNELS stanza of `mqclient.ini` file
- Errors that retrieve CCDT are reported in error logs

Example:

```
AMQ9795: The client channel definition could not be retrieved from its URL,
error code (16).
```

EXPLANATION:

The client channel definition location was specified as URL
`'http://www.example.com/files/AMQCLCHL.TAB'`, however the file could not be
retrieved from this location.

The error returned was (16) 'HTTP response code said error'. The protocol
specific response code was (404).

ACTION:

Ensure that the URL is reachable and if necessary correct the details provided.

Figure 9-10. Precedence order for finding the CCDT

This figure lists the order of precedence for a client application to find a CCDT.

If the client application cannot find a CCDT, an error is reported in the error logs.

MQI channel objects

- SVRCONN
 - Channel type used by all clients
 - MCAUSER security implications, SVRCONN attributes, and performance considerations
 - Use standard facilities to define and display status on IBM MQ server
- CLNTCONN with `runmqsc -n` option
 - Correct method to use to create CCDT for IBM MQ for z/OS V8 and later
 - CCDT created at the client side, no CCDT download necessary
- z/OS: CLNTCONN with CSQUTIL MAKECLNT option
 - MAKECLNT last update before IBM MQ V8
 - CCDT had to be downloaded to client
 - Do not use MAKECLNT for IBM MQ V8 for z/OS and later

Figure 9-11. MQI channel objects

On the queue manager side, the SVRCONN channel is defined for use by clients. It is not necessary to have a CLNTCONN definition to connect to an SVRCONN channel. The advantage of a CLNTCONN and a CCDT are the extra capabilities, such as use of SSL, or use of an alternative queue manager to connect to, perhaps for a failover situation.

The setting of the SVRCONN SHARECNV attribute can be tuned for performance. Having a single socket host multiple connections is called multiplexing, and might enhance the performance of an application. The following sections of the IBM Knowledge Center explain the settings and application considerations for:

- Tuning client and server connection channels
- MQI client: Default behavior of client-connection and server-connection channels

When displaying channel status, SVRCONN-specific attribute CURSHCNV can be used to determine whether applications are using multiplexed connections.

When defining CLNTCONN channels *on the queue manager side*, all channels must be defined in the same queue manager because CLNTCONN creates the file that needs to be downloaded to the client side. This file cannot be concatenated or patched up from several queue managers; it must be one file from the same queue manager.

A new option with IBM MQ V8 is the ability to generate CLNTCONN channels in the client side.

**Note**

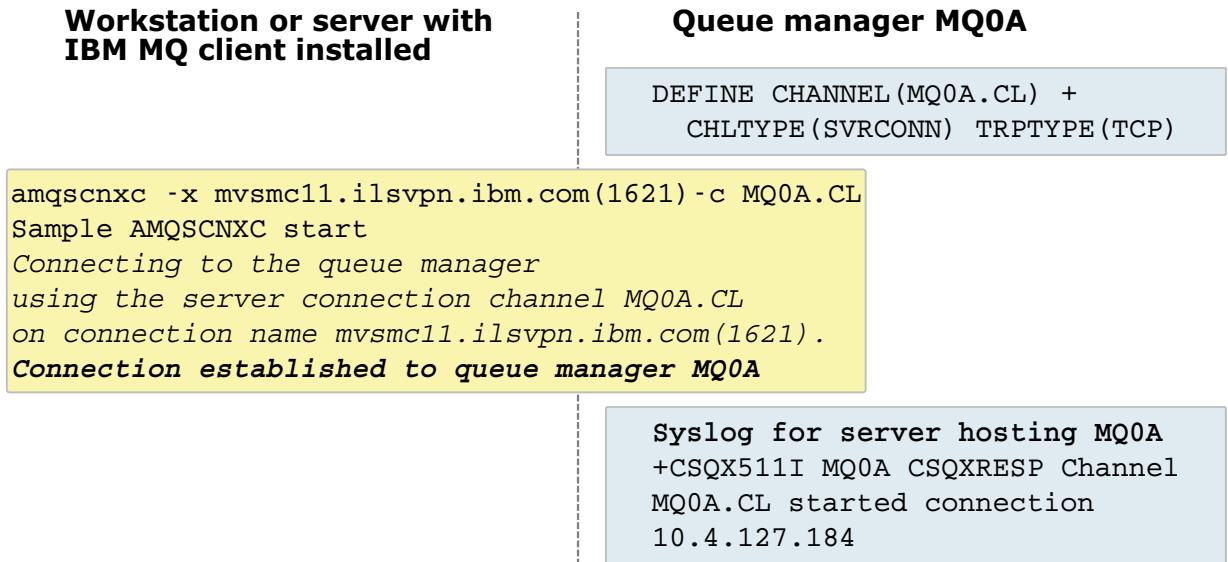
To define a CLNTCONN on the client side, use the `-n` option of `runmqsc` to create a CCDT `AMQCLCHL.TAB` file.

The SVRCONN on the IBM MQ server queue manager must match the name of the CLNTCONN, in this case `MQ00.CL`.

```
C:\>runmqsc -n
5724-H72 (C) Copyright IBM Corp. 1994, 2014.
Starting local MQSC for 'AMQCLCHL.TAB'.
DEF CHL(MQ00.CL) CHLTYP(CLNTCONN) TRPTYPE(TCP)
CONNNAME('mvsmc11.ilsvpn.ibm.com(1600)')
    1 : DEF CHL(MQ00.CL) CHLTYP(CLNTCONN) TRPTYPE(TCP) CONNNAME('mvsmc11.ilsvpn
    .ibm.com(1600)')
AMQ8014: WebSphere MQ channel created.
end
2 : end
No commands have a syntax error.
```

Testing client to server connectivity with MQCONN

- Components required:
 - SVRCONN channel on the IBM MQ server side
 - IBM MQ client installed on client side
 - IBM MQ sample client program to test connectivity: amqscnxc



IBM MQ clients

© Copyright IBM Corporation 2017

Figure 9-12. Testing client to server connectivity with MQCONN

This example uses a z/OS queue manager. The displays show a connection that uses an MQCONN call from IBM MQ sample application `amqscnxc` in a Windows platform to the SVRCONN channel `MQ0A.CL` defined in the IBM MQ for z/OS queue manager `MQ0A`.

On the client side, no MQSERVER variable or client connection definition table (CCDT) is used. The MQCONN call in the application contains all the information necessary to connect to the `MQ0A.CL` SVRCONN channel by using the MQCD structure.

Revisiting MQCONN

```

MQCNO  cno = {MQCNO_DEFAULT} /* connection options*/
MQCSP  csp = {MQCSP_DEFAULT};/* security parms      */
MQHCONN Hcon;           /* connection handle      */
MQLONG  CompCode;        /* completion code       */
MQLONG  CReason;         /* reason code          */
char    QMName[50];      /* queue manager name   */
char    *UserId;          /* Id for authentication */

MQCONN (QMName,      ➔ /* queue manager      */
         &cno,          ➔ /* connection options */
         &Hcon,          ➙ /* connection handle   */
         &CompCode,       ➙ /* completion code    */
         &CReason);     ➙ /* reason code        */

```

Figure 9-13. Revisiting MQCONN

Version-dependent attributes MQCNO (1 of 2)

```

MQCHAR4  StrucId;          /* Structure identifier      */
MQLONG   Version;          /* Structure version number */
MQLONG   Options;          /* MQCONNX actions control opts */

/* Ver:1 */                ← Following fields recognized if MQCNO_VERSION_2 or higher
MQLONG   ClientConnOffset; /* Client conn MQCD struct offset*/
MQPTR    ClientConnPtr;   /* Client conn MQCD struct addr */

/* Ver:2 */                ← Following fields recognized if MQCNO_VERSION_3 or higher
MQBYTE128 ConnTag;        /* Queue-manager connection tag */

/* Ver:3 */                ← Following fields recognized if MQCNO_VERSION_4 or higher
PMQSCO   SSLConfigPtr;    /* Client conn MQSCO struct addr */
MQLONG   SSLConfigOffset; /* Client conn MQSCO str offset */

/* Ver:4 */                ← Following fields recognized if MQCNO_VERSION_5 or higher

```

IBM MQ clients

© Copyright IBM Corporation 2017

Figure 9-14. Version-dependent attributes MQCNO (1 of 2)

The `ClientConnPtr` is used to point to the channel descriptor structure (MQCD). As with other structures, it is important to set the correct structure version number. Without the correct version number, the MQCD pointer is ignored.

IBM MQ V8.0.0.4 and later adds CCDT URL support to the MQCNO structure, by providing a `CCTUrlPtr` or `CCTUrlOffset` in the MQCNO. `CCTUrlLength` specifies the length of the URL string.

Example

```

cno.Version = MQCNO_VERSION_6;
cno.CCDTUrlPtr = "file:///var/mqm/qmgrs/QMGR@ipcc/AMQCLCHL.TAB";
cno.CCDTUrlLength = strlen(cno.CCDTUrlPtr);
MQCONNX(qmname, &cno, &hconn, &CompCode, &Reason);

```

Version-dependent attributes MQCNO (2 of 2)

```

MQBYTE24 ConnectionId;           /* Unique Connection Identifier */
MQLONG    SecurityParmsOffset;   /* Offset of MQCSP structure */
MQCSP     SecurityParmsPtr;     /* Address of MQCSP structure */
/* Ver:5 */
    <----- Following fields recognized if MQCNO_VERSION_6 or higher
PMQCHAR   CCDTUrlPtr;          /* Address of CCDT URL string */
MQLONG    CCDTUrlOffset;        /* Offset of CCDT URL string */
MQLONG    CCDTUrlLength;        /* Length of CCDT URL */
MQBYTE8   Reserved;            /* Reserved */
/* Ver:6 */

```

IBM MQ clients

© Copyright IBM Corporation 2017

Figure 9-15. Version-dependent attributes MQCNO (2 of 2)

Shown is the continuation of the version-dependent attributes for the MQCNO structure. To use the CCDT URL fields, you must set the structure version to MQCNO_VERSION_6 or higher.

MQCD structure (partial)

```

MQCHAR ChannelName[20];      /* Channel definition name */
MQLONG Version;             /* Structure version number */
MQLONG ChannelType;         /* Channel type */
MQLONG TransportType;       /* Transport type */
MQCHAR Desc[64];            /* Channel description */
MQCHAR QMgrName[48];         /* Queue-manager name */
MQCHAR XmitQName[48];        /* Transmission queue name */

. . . . .

/* Ver:1 */

. . . . .

MQCHAR ConnectionName[264];  /* Connection name */
MQCHAR RemoteUserIdentifer[12];/* First 12 bytes part UID */
MQCHAR RemotePassword[12];   /* Password from partner */
/* Ver:2 */

. . . . .

```

IBM MQ clients

© Copyright IBM Corporation 2017

Figure 9-16. MQCD structure (partial)

If you compare the channel definition structure (MQCD) to the attributes of a channel definition, you see many parallels.

The MQCD structure must use a special initialization structure called MQCD_CLIENT_CONN_DEFAULT.

MQCD_CLIENT_CONN_DEFAULT (partial)

- There are other `MQCD_DEFAULT` structures in the `cmqxc.h` header file
- For clients, initialize structure with `MQCD_CLIENT_CONN_DEFAULT`
- `MQCD_CLIENT_CONN_DEFAULT` has the correct initialization values for the Version, ChannelType, and TransportType fields of the `MQCD` structure

```
#define MQCD_CLIENT_CONN_DEFAULT {"", \
                                MQCD_VERSION_6, \
                                MQCHT_CLNTCONN, \
                                MQXPT_TCP, \
                                {"", \
                                {"", \
                                {"", \
                                ...
```

Figure 9-17. `MQCD_CLIENT_CONN_DEFAULT` (partial)

The values that are used by the `MQCD` initialization structure for an IBM MQ client are displayed in the slide. Observe how the version is set to 6. Look at the parallels with a channel definition. When you define a CLNTCONN channel, you type:

```
def chl(CHANNEL.NAME) chltype(CLNTCONN) trptype(TCP) CONNAME(...) MCAUSER(...)
```

The values that are passed to the structure show the channel type and the transport type, similar to the channel definition. You pass the connection information in your code.

Checklist to use MQCONN for the client connection

1. Include the `cmqcx.h` header file in the code
2. Include an MQCD structure declaration initialized with the `MQCD_CLIENT_CONN_DEFAULT` structure
3. At a minimum, determine the values to set the following parameters:
 - QMgrName for the MQCONN call parameter
 - ConnectionName for the MQCD structure
 - ChannelName for the MQCD structure
4. Point the MQCNO ClientConnPtr to the MQCD declaration

Figure 9-18. Checklist to use MQCONN for the client connection

To create the connection from your code, you follow the steps that are outlined in the slide. In the lab for this unit, you code a client connection.

Connect with CCDT and put a message

- Required:

- SVRCONN and local queue object definitions on IBM MQ queue managers
- CCDT with all CLNTCONN definitions created with `runmqsc -n` on client side
- For this example, CCDT information is specified on `mqclient.ini` without any variables set
- Sample IBM MQ client put program on client side: `amqsputc`

Workstation with IBM MQ client installed

```
mqclient.ini
```

```
ChannelDefinitionDirectory=
C:\IBM\WEBSPH~1\Qmgrs\QMGR1\@ipcc
ChannelDefinitionFile=AMQCLCHL.TAB
```

CLNTCONN definition with `runmqsc -n`

```
DEFINE CHANNEL (MQ84.SVRCONN) +
CHLTYPE(CLNTCONN) TRPTYPE(TCP)
CONNNAME('localhost(1690)')
```

```
amqsputc PRICE.CL MQ84
Sample AMQSPUTO start
target queue is PRICE.CL
Hello test CCDT setup
Sample AMQSPUTO end
```

IBM MQ clients

Figure 9-19. Connect with CCDT and put a message

IBM MQ server with Windows queue manager MQ84

SVRCONN definition

```
DEFINE CHANNEL (MQ84.SVRCONN) +
CHLTYPE(SVRCONN) TRPTYPE(TCP)
```

```
amqsget PRICE.CL MQ84
Sample AMQSGETO start
message <Hello test CCDT setup>
Sample AMQSGETO end
```

© Copyright IBM Corporation 2017

This slide shows the configuration of a client channel definition table from the client side and the queue manager side.

It is not imperative to set up the `mqclient.ini`. The IBM MQ client code looks for the table in a default location first. Use of a specific location for the table is helpful to control the version of the CCDT in use.

In the client side, the channel is defined with `runmqsc -n`. Before IBM MQ V8, you were required to define the CLNTCONN in the queue manager, and transfer the CCDT to the client. With IBM MQ V8, the client has its copy of the `runmqsc` utility, which works by using the `-n` parameter.

On the queue manager side, a type SVRCONN channel is defined with the same name as the CLNTCONN channel.

The slide shows use of the `amqsputc` sample IBM MQ client application. *Not shown on the slide:* If the connection is successful, you can check the channel status with `runmqsc` or IBM MQ Explorer and it should be in `RUNNING` status. Unlike a server message channel, a client channel is not manually started; it starts with the application connection, and ends when the application disconnects.

The MQI channel is bidirectional, which means that you can also get messages by using the same channel.

Use MQSERVER environment variable

- Components required:
 - SVRCONN channel and local queue on the IBM MQ server side (not shown)
 - IBM MQ client installed on client side
 - IBM MQ sample client programs: `amqsputc` and `amqsgetc`

Client

```
C:\>set MQSERVER=MQ84.SVRCONN/TCP/localhost(1690)
C:\>amqsputc PRICE.CL
Sample AMQSPUT0 start
target queue is PRICE.CL
abc
Sample AMQSPUT0 end
```

Client

```
Example of running amqsgetc
C:\IBM\MQ\tools\c\Samples\Bin>amqsgetc PRICE.CL
Sample AMQSGET0 start
message <abc>
no more messages
Sample AMQSGET0 end
```

Figure 9-20. Use MQSERVER environment variable

The example in this slide uses the same SVRCONN channel in the queue manager. Instead of the CCDT table, this connection is made by using the MQSERVER variable in the client side.

When you use a CCDT and an MQSERVER, the MQSERVER connection takes precedence over the CCDT connection. If the MQSERVER connection was to a different queue manager, such as MQ01, the connection is made to queue manager MQ01. In this case, it is using the same channel.

This example shows how you can get the message that you put by using the same channel on the client side.

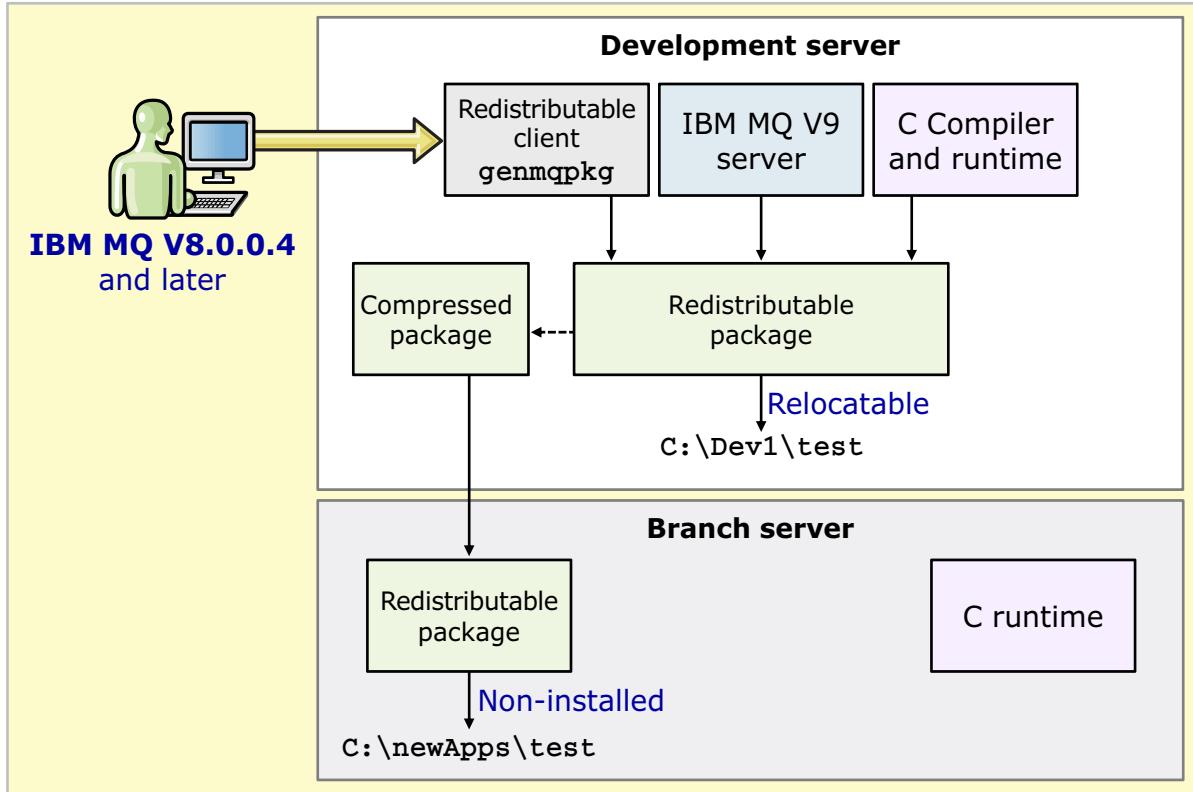
IBM MQ client connection capabilities summary

Connection capability	MQCONNX MQCNO	MQSERVER	CCDT
Connect to alternate queue managers	NO	YES* <i>if ConnectionName is a list</i>	YES
Security exits	YES	NO	YES
Use a subset of channel attributes	YES	Few attributes available	YES
SSL	YES	NO	YES
Server failover detection, load balancing, and weighting	NO	NO* <i>ConnectionName list helps failover</i>	YES

Figure 9-21. IBM MQ client connection capabilities summary

This table summarizes the IBM MQ client capabilities per selected connectivity option.

Redistributable clients



IBM MQ clients

© Copyright IBM Corporation 2017

Figure 9-22. Redistributable clients

Redistributable implies distribution of the needed runtime components with an IBM MQ client application, both inside, and outside the customer's IBM MQ environment. Redistributable clients can be relocatable, or non-installed.

A relocatable client implies putting the files somewhere else other than a fixed default location, for example, instead of installing into `/opt/`, installing into `/usr/local`.

A non-installed client implies that an installer is not required to lay down client files. These client files can be extracted and copied as required. The non-installed redistributable client does not require an IBM MQ installation.

To prepare a redistributable client:

1. Compile your client application as usual in the server that has IBM MQ, the C compiler and runtime, and your redistributable client package installed.
2. Use the `genmqpkg` utility to generate a package appropriate to the requirements of your application.
3. Include your client application in the package that is generated from the `genmqpkg` utility.
4. Compress (.zip) your application as needed. You are now ready to move it to another server, or to a different path in the same server where the package was assembled.

5. Ensure that the correct runtime is installed on the server where the application is moved. No IBM MQ libraries are required outside the libraries that the genmqpkg utility packaged.

IBM MQ client security considerations

Authentication:

- MCAUSER attribute of SVRCONN
 - If left blank, a platform-dependent ID is used 
 - Channel authentication records can be used to map the MCAUSER of the same SVRCONN channel to different user IDs with different authorizations
- mqccred exit
 - Assumption: IBM MQ server has CONNAUTH enabled
 - Use with client applications that do not currently send a user ID and password
 - mqccred.ini file and MQCCRED variable
 - Password obfuscation for mqccred.ini file
 - Update CLNTCONN definition with SCYEXIT("mqccred(ChIExit)") attribute
- Message exits

Authorization:

- Business as usual on the server side
- Consider use of CHLAUTH

IBM MQ clients

© Copyright IBM Corporation 2017

Figure 9-23. IBM MQ client security considerations

Unit summary

- Describe the differences between an IBM MQ client and an IBM MQ server
- List the supported languages per platform that can be used to code an IBM MQ client application
- Explain how to compile an IBM MQ client application
- Describe the considerations to observe when working with IBM MQ clients
- Describe the use of environment variables or a client channel definition table (CCDT) to connect an IBM MQ client to a queue manager in an IBM MQ server
- Explain how to use the MQCONN call to connect an IBM MQ client application to a queue manager directly from your code
- Differentiate the capabilities that are available with the various client connectivity options
- Describe a redistributable client (V8.0.0.4 and later)
- Summarize the use and precedence order of a CCDT URL

Review questions (1 of 2)

- 
1. True or False: An IBM MQ client installation does not have a configured queue manager or queues.
 2. Select the best answer. You write an application that does several MQPUT calls to the queue in the server queue manager. Which of these actions is important?
 - a. Ensure that you create a client connection channel by using `runmqsc -n` on the client side
 - b. Ensure that you can override the maximum message size of the target queue manager channel
 - c. Ensure that the MQSERVER client-side variable is configured correctly
 - d. Ensure that you reset the `CodedCharSetID` field of the MQMD

Figure 9-25. Review questions (1 of 2)

Write your answers here:

- 1.
- 2.

Review questions (2 of 2)

3. You inherit a client application that uses the `MQCONN` with the `MQCD.ChannelName` set to `SALES.MQ03`. The client workstation has a `CLNTCONN` channel defined to `SALES.MQ07`. Your only update to the application was adding a sales tax calculation. What channel is used when you test the application?
- a. The channel in the `MQSERVER` environment variable, which is at `SALES.MQ04`
 - b. The channel used by the `MQCONN` call, `SALES.MQ03`
 - c. The `CCDT` defined channel, `SALES.MQ07`
 - d. The channel in the `CCDT` that the `DefaultPrefix` in the `mqsc.ini` file points to



IBM MQ clients

© Copyright IBM Corporation 2017

Figure 9-26. Review questions (2 of 2)

Write your answer here:

3.

Review answers (1 of 2)

1. True or False: An IBM MQ client installation does not have a configured queue manager or queues.
The answer is: True. An IBM MQ client is a set of libraries with IBM MQ functionality. It does not have a queue manager, so no queues can be defined on the client side.
2. Select the best answer. You write an application that does several MQPUT calls to the queue in the server queue manager. Which of these actions is important?
 - a. Ensure that you create a client connection channel by using `runmqsc -n` on the client side
 - b. Ensure that you can override the maximum message size of the target queue manager channel
 - c. Ensure that the MQSERVER client-side variable is configured correctly
 - d. Ensure that you reset the `CodedCharSetID` field of the MQMD

The answer is: d. Reset the CodedCharSetID field of the MQMD. Depending on the situation, the CodedCharSetID might change when the function call completes. This situation might result in the incorrect CCSID set in the next message put.

Review answers (2 of 2)

3. You inherit a client application that uses the `MQCONN` with the `MQCD.ChannelName` set to `SALES.MQ03`. The client workstation has a `CLNTCONN` channel defined to `SALES.MQ07`. Your only update to the application was adding a sales tax calculation. What channel is used when you test the application?
- The channel in the `MQSERVER` environment variable, which is at `SALES.MQ04`
 - The channel used by the `MQCONN` call, `SALES.MQ03`
 - The `CCDT` defined channel, `SALES.MQ07`
 - The channel in the `CCDT` that the `DefaultPrefix` in the `mqs.ini` file points to

The answer is: b. The channel in the MQCONN call, SALES.MQ03. A channel that is coded in the MQCONN call supersedes the MQSERVER and CCDT connections.



Working with an IBM MQ client

IBM MQ clients

© Copyright IBM Corporation 2017

Figure 9-29. Working with an IBM MQ client

Exercise objectives

- Compile an existing program as an IBM MQ client
- Review a client connection channel definition and test connectivity to a queue manager by using the client channel definition table
- Configure and test connectivity to the queue manager by using first the MQCCDTURL and then the MQSERVER environment variable
- Alter the client program to use the MQCONN call to establish connectivity to the queue manager



IBM MQ clients

© Copyright IBM Corporation 2017

Figure 9-30. Exercise objectives

Unit 10. Introduction to publish/subscribe

Estimated time

01:00

Overview

This unit teaches you how to work with the publish/subscribe messaging style. It explains the history of publish/subscribe, describes its components, and defines key terminology to use when referring to the co-existing publish/subscribe capabilities. The unit also teaches you how to code subscriber and publisher applications.

How you will check your progress

Accountability:

- Review questions
- Lab Exercises

References

IBM Knowledge Center for IBM MQ V9 documentation

Unit objectives

- Differentiate between publish/subscribe and point-to-point messaging
- Describe how the history of publish/subscribe influences its functions and terminology
- Identify the basic components of publish/subscribe
- Describe key properties of topics, subscriptions, and publications
- Explain how to write an application that subscribes to a topic by using the MQSUB function call
- Describe how to code the MQOPEN and MQPUT calls to write an application that publishes to a topic

Introduction to publish/subscribe

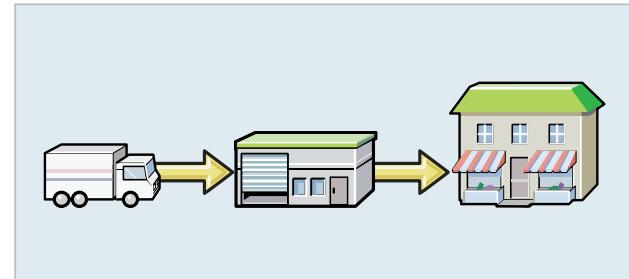
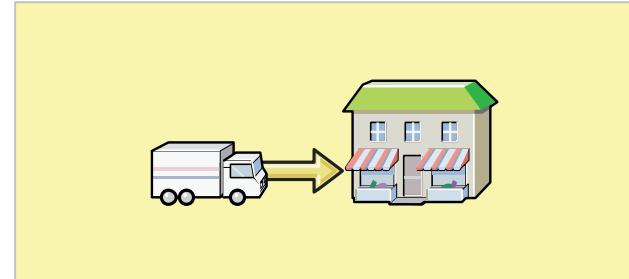
© Copyright IBM Corporation 2017

Figure 10-1. Unit objectives

Point-to-point and publish/subscribe

- Point-to-point application
 - Sends messages to a predefined destination
 - Application does not need to know where the destination is
 - IBM MQ locates the target by using object definitions

- Publish/subscribe application
 - Publishes messages to an interim destination according to a topic
 - Interested recipients subscribe to the topic
 - No explicit connection between publishing and subscribing applications



[Introduction to publish/subscribe](#)

© Copyright IBM Corporation 2017

Figure 10-2. Point-to-point and publish/subscribe

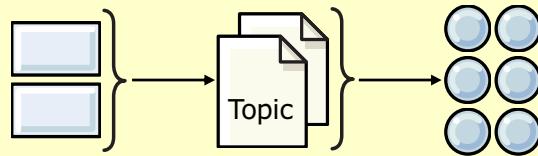
With publish/subscribe, the sending, or publishing of messages and the subscription or getting of messages is done by using the topic tree as an intermediary. It is not apparent where the message is going, or how it shows up, the publish/subscribe process handles those details.

Publish/subscribe uses distributed and cluster channels, but the source and destination of the messages are determined by the intermediary publish/subscribe process. Publish/subscribe adds another level of application decoupling.

Distributed publish/subscribe patterns

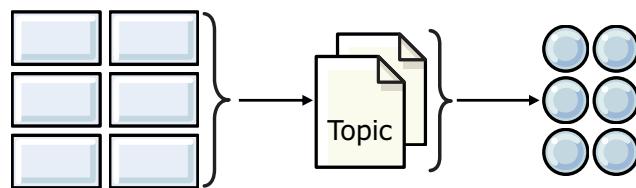
Few-to-many:

- Research
- News tickers



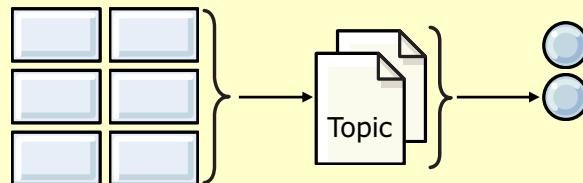
Many-to-many:

- Prices
- Quotations



Many-to-few:

- Orders
- Check inventory



Key: Publisher Subscriber

Introduction to publish/subscribe

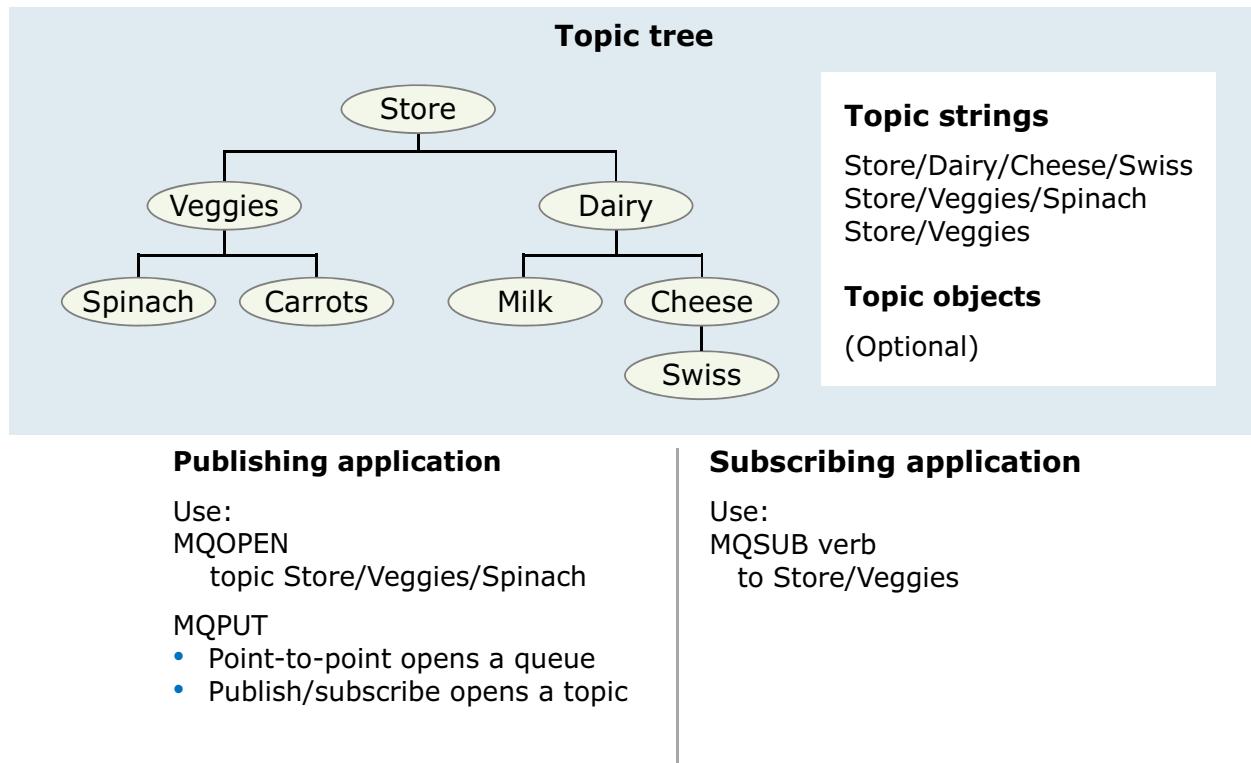
© Copyright IBM Corporation 2017

Figure 10-3. Distributed publish/subscribe patterns

One of the reasons publish/subscribe gained popularity was the flexibility of configurations that are possible to achieve. The publish/subscribe patterns apply to stocks, sports, sales, and other popular applications.

Forms of publish/subscribe are even embedded in software products such as IBM Integration Bus, formerly WebSphere Message Broker, IBM MQ Managed File Transfer, and in telemetry mobile applications.

Publish/subscribe basic components



Introduction to publish/subscribe

© Copyright IBM Corporation 2017

Figure 10-4. Publish/subscribe basic components

At the center of publish/subscribe is the topic tree. A topic tree is a hierarchical structure that contains and organizes topics. A topic tree's organization is influenced by how the topics are created. The topic tree might be compared to file system with directories and sub directories.

Topics can be created by using the create topic commands or by specifying the topic for the first time in a publication or subscription. Topics are structured into topic strings. Topic strings have separate categories that are separated by a slash (/) character.

When messages are sent in a publish/subscribe application, they are published, or put to a topic instead of a queue. A topic can be an explicitly defined object, or a publish or a subscribe operation to the topic string might dynamically create a topic. Published messages are called publications.

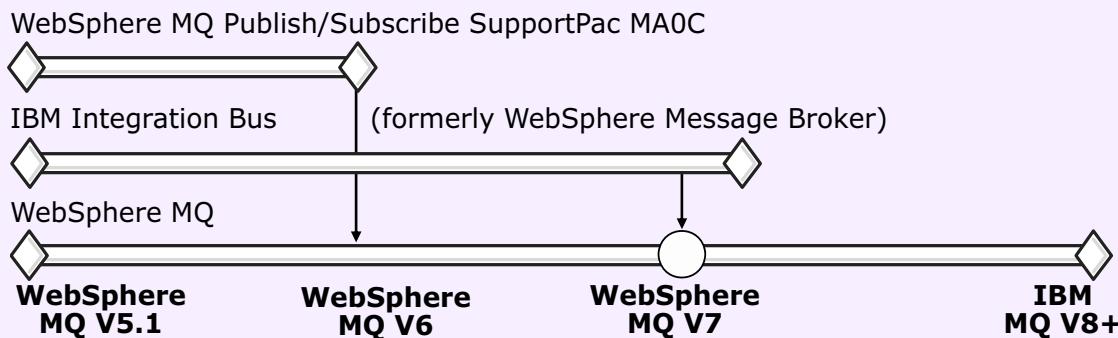
To get the publications, applications subscribe to the topics or topic strings. The subscriptions can be an explicitly defined object, or a dynamic subscription created by an application.

So you have the topic tree, topics, publications and publishers, and subscribers and subscriptions.

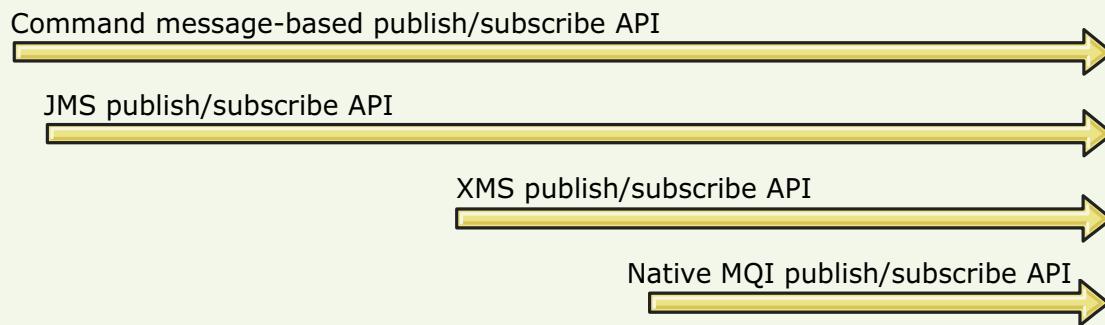
To better understand today's publish/subscribe implementation, you look at publish/subscribe over the years.

Current publish/subscribe functions: A little history

Publish/subscribe brokers



IBM MQ publish/subscribe APIs



Introduction to publish/subscribe

© Copyright IBM Corporation 2017

Figure 10-5. Current publish/subscribe functions: A little history

The implementation and terminology of publish/subscribe can be confusing. Queue managers have a “mode” or PSMODE parameter. The terms “managed queues” and “queued publish/subscribe” are mentioned. Why such terminology?

Publish/subscribe started out as IBM MQ Support pack MA0C. Publishing and subscribing with the original support pack involved the use of the RFH command message-based API to send subscription and publication requests. WebSphere Message Broker also had a separate publish/subscribe mechanism and used WebSphere Message Broker capabilities to do the publish/subscribe actions. Both the support pack and WebSphere Message Broker referred to the publish/subscribe engine as a “broker”.

As publish/subscribe evolved, it was included in the IBM MQ product. The publish/subscribe API was also changed and augmented. This course does not address migration of the “old” publish/subscribe applications. However, if you work with an existing application that uses the “old” API and broker function, it is good to keep in mind that the need to address or handle a migration might surface.

A good example of the publish/subscribe heritage is the subscription object PSPROP attribute, which deals with the ways that related message properties are added to messages sent to the defined. You look at PSPROP later in this unit.

Publish/subscribe terminology baseline

- Terms for publish/subscribe support pack or WebSphere Message Broker engine: Broker, queued publish/subscribe, command-message-based
- The V7+ publish/subscribe is referred to as: Integrated publish/subscribe, publish/subscribe interface, publish/subscribe MQI
- Managed publish/subscribe: Publish/subscribe used dynamically without creating defined queues or topics
 - “Managed” can also apply to a dynamically created queue or a dynamically created topic
- Administered: Creating an explicitly defined object for publish/subscribe such as a topic or a subscription
 - An administered object such as a subscription can have a managed queue
- Streams: Concept that is used in queued publish/subscribe to separate the flows for different topics
 - Default stream available to V7+ queue managers if queued publish/subscribe is enabled
 - Named streams require a queue definition added to
SYSTEM.QPUBSUB.QUEUE.NAMELIST

Introduction to publish/subscribe

© Copyright IBM Corporation 2017

Figure 10-6. Publish/subscribe terminology baseline

This slide covers some of the terminology that you might see in the IBM Knowledge Center and also notice when the channel initiator comes up.

Publish/subscribe functions

- **PSMODE** queue manager attribute controls publish/subscribe versions
 - **ENABLED:** The default, which indicates that both the **publish/subscribe engine** and **queued publish/subscribe** are running
 - **COMPAT:** Publish/subscribe engine active, and queued publish/subscribe stopped
 - **DISABLED:** Both publish/subscribe engine and queued publish/subscribe are stopped

```
.... .... ....
AMQ5052: The queue manager task 'PUBSUB-DAEMON' has started.
.... .... ....
AMQ5975: 'IBM MQ Distributed Pub/Sub Controller' has started.
.... .... ....
AMQ5975: 'IBM MQ Distributed Pub/Sub Fan Out Task' has started.
.... .... ....
AMQ5806: Queued Publish/Subscribe Daemon started for queue manager
MQ01.
.... .... ....
```

[Introduction to publish/subscribe](#)

© Copyright IBM Corporation 2017

Figure 10-7. Publish/subscribe functions

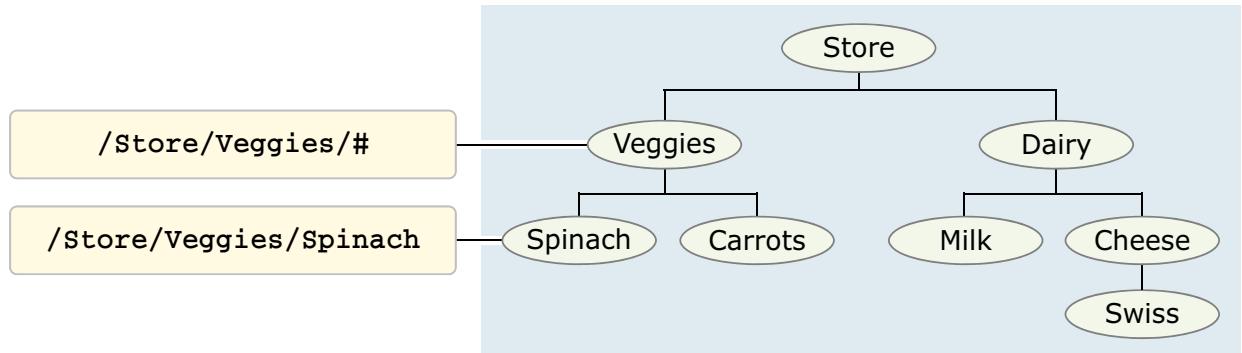
PSMODE is a queue manager attribute across the z/OS and distributed platforms that controls the type of publish/subscribe functionality that is enabled in a queue manager.

In the snapshot of the channel initiator start console messages, you can see how the publish/subscribe engine starts, then later the “old” publish/subscribe, that is the “queued” publish/subscribe process also starts.

The predominant modes are:

- **ENABLED:** Use this mode when you are not sure what the mix of applications is. It supports the old and new publish/subscribe, within any limitations that are documented in the IBM Knowledge Center.
- **COMPAT:** The “old” publish/subscribe is not active. Only the new engine is started. If you are sure that no old publish/subscribe applications exist, use this mode.
- **DISABLED:** However, if publish/subscribe is not being used, startup of the publish/subscribe process can be stopped other software components outside the applications, such as WebSphere Message Broker or IBM MQ Managed File Transfer might require queued publish/subscribe to be active. Always confirm what runs in the environment before setting PSMODE to DISABLED.

Topic tree, topic strings, topic nodes, and topic objects



- Topic strings are the common sets of slash structured text that applications publish and subscribe to
- Topic nodes are a segment of the topics
- Topic objects are explicitly defined topics, and are also called administered topics
- The topic tree is a hierarchical structure that holds topic strings and nodes
 - When a topic is published to, subscribed to, or defined as an administered topic object, the topic tree expands to include the new topic



It is **critical** to implement governance and exercise control over the topic tree

[Introduction to publish/subscribe](#)

© Copyright IBM Corporation 2017

Figure 10-8. Topic tree, topic strings, topic nodes, and topic objects

This slide elaborates on topic-related terminology.

Topic trees can take many “shapes”. While some topic trees are hierarchical, others can be “flat”, for instance can have more horizontal nodes.

Since topics nodes can be added in so many ways, it is easy for a topic tree to grow larger than expected. When an application publishes or subscribes, it must search the topic tree. Searching a large topic tree, or a topic tree with unused nodes, eventually results in performance impacts to an application.

A topic node and a topic object are distinct from each other, as the topic object is defined as an IBM MQ object with the MQSC command `DEFINE TOPIC`.

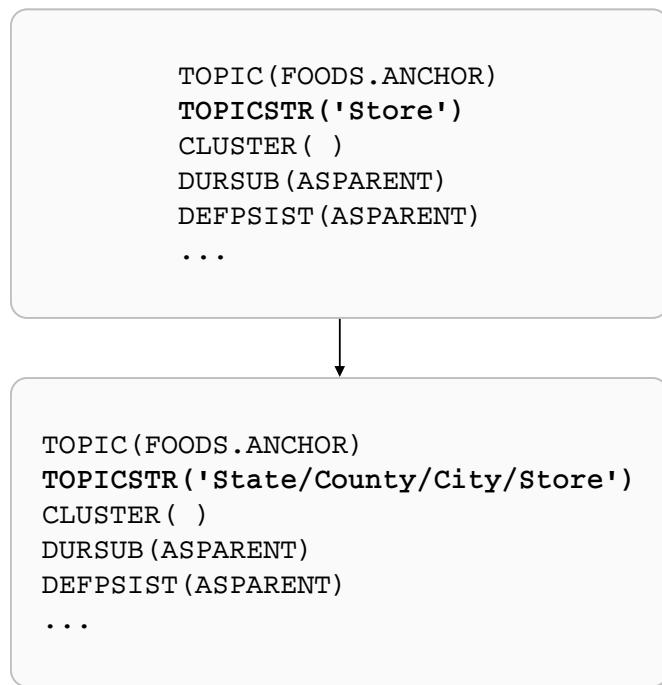
The next slide shows the advantages of a defined topic object.

Topic objects

- Provide an administrative control point for the topic tree
 - Configuration attributes
 - Security profiles
 - Topic tree isolation
- Provide flexibility if the topic tree changes later
- Defined topic objects for publish/subscribe are optional

Use:
MQSUB
to Store/Veggies

Use:
MQSUB
to FOODS.ANCHOR + Veggies



Introduction to publish/subscribe

© Copyright IBM Corporation 2017

Figure 10-9. Topic objects

The ability to dynamically add a topic provides significant flexibility to an organization, but topic objects provide extra flexibility.

Imagine that an organization started locally with one store. This store was successful and grew to a national entity. What happens to all applications that published or subscribed to the original store?

By having an explicitly defined topic object as an anchor topic, an application can use the anchor topic with other nodes in the topic string, and automatically be part of the larger scope. If the applications used the FOODS.ANCHOR topic to start, you have no concerns if extra nodes must be added ahead of the original hierarchy.

The defined topic also helps as a point to establish “parental” attributes and apply security profiles.

Topic object attributes: DISPLAY TOPIC view

- **ASPARENT** values
 - Taken from the first parent administrative node found in the topic tree
 - If no administrative nodes are found, values are taken from SYSTEM.BASE.TOPIC
- **DEFPSIST** is persistence that is used when the application uses the **MQPER_PERSISTENCE_AS_TOPIC_DEF** option

```
DIS TOPIC(FOODS*) ALL
  TOPIC(FOODS.ANCHOR)
  TYPE(LOCAL)
  QSGDISP(QMGR)
  CLUSTER()
  TOPICSTR(Store)
  DEFPRTY(ASPARENT)
  DEFPSIST(ASPARENT)
  DURSUB(ASPARENT)
  NPMSGDLV(ASPARENT)
  PMSGDLV(ASPARENT)
  MDURMDL()
```

```
PUB(ASPARENT)
SUB(ASPARENT)
PUBSCOPE(ALL)
SUBSCOPE(ALL)
PROXYSUB(FIRSTUSE)
WILDCARD(PASSTHRU)
CLROUTE(DIRECT)
DESCR()
DEFPRESP(ASPARENT)
USEDLQ(ASPARENT)
CUSTOM()
ALTDATE(2014-09-15)
... ... ... ... ...
```

Introduction to publish/subscribe

© Copyright IBM Corporation 2017

Figure 10-10. Topic object attributes: DISPLAY TOPIC view

This slide shows a topic object display and introduces the concept of inheritance.

At the base, or starting topic of the topic tree is a topic object called SYSTEM.BASE.TOPIC. If topic objects are defined to use default attributes, and no other topic objects are between the topic string and the SYSTEM.BASE.TOPIC that change the attributes, the object attributes are inherited from the SYSTEM.BASE.TOPIC object.

When a topic is displayed with the “DIS TOPIC” command, you see the value ASPARENT in many of the attributes. This value indicates that the topic object definition takes its attributes from the next higher topic object definition.

The default values for topic attributes are listed in the IBM Knowledge Center.

Topic object attributes: DISPLAY TPSTATUS view

- **TPSTATUS** shows some actual values in place of **ASPARENT**

```

TPSTATUS( Store/Veggies/Spinach )
TYPE(TOPIC)
DEFPRES(SYNC)
DEFPSIST(NO)
DEFPRTY(0)
DURSUB(YES)
PUB(ENABLED)
SUB(ENABLED)
ADMIN( )
MDURMDL(SYSTEM.DURABLE.MODEL.QUEUE)
MNDURMDL(SYSTEM.NDURABLE.MODEL.QUEUE)
...

```

```

...
NPMGDLV(ALLAVAIL)
PMSGDLV(ALLDUR)
RETAINED(YES)
PUBCOUNT(0)
SUBCOUNT(2)
PUBSCOPE(ALL)
SUBSCOPE(ALL)
CLUSTER( )
USEDLQ(YES)
CLROUTE(NONE)

```

[Introduction to publish/subscribe](#)

© Copyright IBM Corporation 2017

Figure 10-11. Topic object attributes: DISPLAY TPSTATUS view

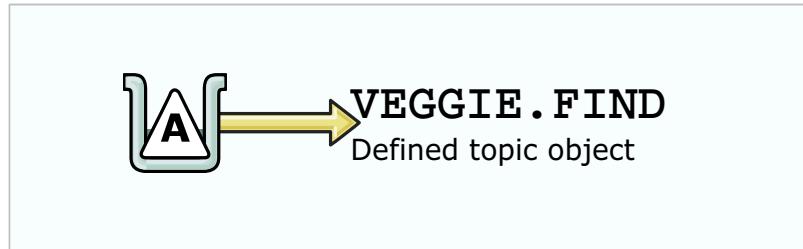
To check the status of a topic, use the DISPLAY TPSTATUS command. When displaying the status, the actual inherited attributes, rather than “ASPARENT”, are shown on the display.

If you need to match all topic status, you use the number sign (#) instead of the asterisk, for example:

```
DIS TPSTATUS( '#' ) ALL
```

Remember that if the topic tree is large the command might cause quite a bit of searching.

Topic alias



- Alias queue that points to an explicitly defined topic object
- Topic object must be defined in same queue manager as the alias queue
- Set TARGTYPE attribute to TOPIC in the QALIAS definition

Introduction to publish/subscribe

© Copyright IBM Corporation 2017

Figure 10-12. Topic alias

If an application currently puts messages onto a queue, it can be made to publish to a topic by making the queue name an alias for the topic.

Wildcard schemes

- Wildcard **character** scheme
 - Avoid using in MQ V7 and later applications
 - Matching is slower – used for MQ V6 and earlier applications
- Wildcard **topic** scheme is default
 - # is the multi-level string
 - + is the single level string
- Assume wildcard topic string Soccer/MLS/Orlando
- Using multi-level string if you subscribe to Soccer/MLS/# you get messages for Soccer/MLS and Soccer/MLS/Orlando
- Using single-level string if you subscribe to Soccer/+ you get messages for Soccer/MLS, but not Soccer/MLS/Orlando

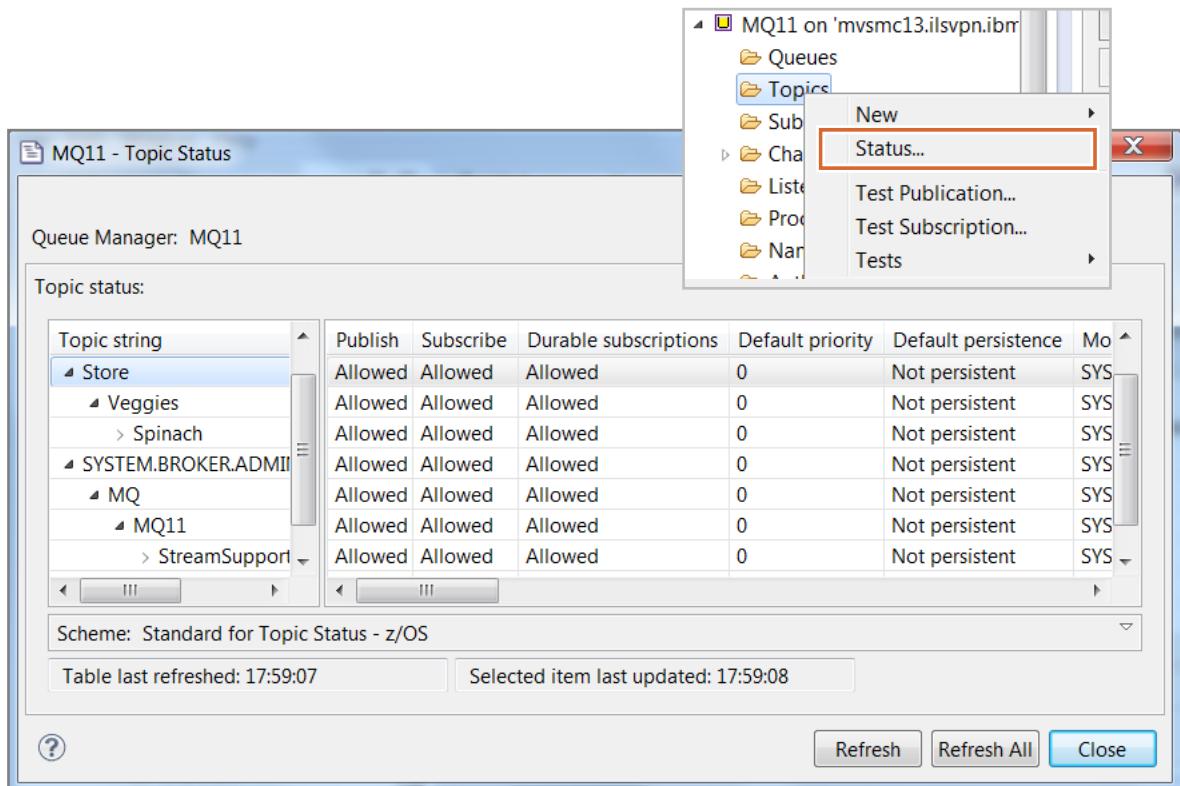
[Introduction to publish/subscribe](#)

© Copyright IBM Corporation 2017

Figure 10-13. Wildcard schemes

IBM Training

Topic status with IBM MQ Explorer



Introduction to publish/subscribe

© Copyright IBM Corporation 2017

Figure 10-14. Topic status with IBM MQ Explorer

IBM MQ Explorer provides capabilities to create topics, display the status, and also offers the capability to test a publication and subscription.

The topic string hierarchy can be seen in the IBM MQ Explorer display.

When using the test panes, the Test Subscription should be started before the Test Publication.

Subscriptions: The three aspects (1 of 2)

- Subscriptions link topics to queues
- Three aspects to subscriptions:
 1. Subscription creation and deletion
 2. Subscription queue management
 3. Subscription lifetime
- Subscription creation and deletion
 - Application-created subscription uses `MQSUB` to create and `MQCLOSE` to delete
 - Administrative or explicitly defined subscription object
- Subscription queue management
 - Managed where the subscription creates and deletes the queue:
`DESTCLASS MANAGED`
 - Unmanaged where a queue is explicitly defined or provided: `DESTCLASS PROVIDED`

```
SUB (VEGETARIAN.SEARCH)
DURABLE (YES)
SUBTYPE (ADMIN)
TOPICSTR (Store/Veggies)
DEST (SYSTEM.MANAGED.DURABLE.CDC
456AF9B060E32)
TOPICOBJ (FOODS.ANCHOR)
DESTCLAS (MANAGED)
EXPIRY (UNLIMITED)
```

```
SUB (VEGGIE.FIND)
DURABLE (YES)
SUBTYPE (ADMIN)
TOPICSTR (Store/Veggies)
DEST (VEGGIES.QLOCAL)
TOPICOBJ (FOODS.ANCHOR)
DESTCLAS (PROVIDED)
EXPIRY (UNLIMITED)
```

[Introduction to publish/subscribe](#)

© Copyright IBM Corporation 2017

Figure 10-15. Subscriptions: The three aspects (1 of 2)

Subscriptions might be one of the most confusing components of publish/subscribe. Some of the terminology that is used for subscriptions might appear to be contradictory, but breaking down the different aspects of subscription should prove helpful. Subscriptions have three distinct aspects:

- Creation and deletion. How was this subscription created? Consider two possibilities: created by an application, or created administratively by with an explicitly defined subscription object.
- Subscription queue management. How is the queue used by the subscription created or handled? If the subscription `DESTCLASS` attribute is set to `MANAGED`, the queue is taken care of by IBM MQ. These queues are dynamic queues that are prefixed with “`SYSTEM.MANAGED`.” The next node in the queue name will be `DURABLE` or `NONDURABLE`, which brings you to the third subscription aspect.
- Subscription lifetime. Two considerations. The first consideration is a durable subscription, which means that the live of the subscription is not depending on an application. The second type is non-durable; that is, an application controls the subscription’s lifetime.

The three subscription aspects are summarized in the next slide.

Subscriptions: The three aspects (2 of 2)

- Subscription lifetime
 - Durable subscriptions: Lifetime is independent of applications
 - Non-durable subscriptions: Lifetime is driven by applications

Valid subscription combinations

Queue management →	Managed		Unmanaged	
Lifetime →	Durable	Non-durable	Durable	Non-durable
Creation and deletion ↓				
Administrative	Yes	No	Yes	No
Application	Yes	Yes	Yes	Yes

Figure 10-16. Subscriptions: The three aspects (2 of 2)

This slide summarizes the three aspects of subscriptions: creation and deletion, queue management, and lifetime. You identify these three aspects in the DISPLAY SUB output.

Subscription commands: DIS SUB(VEGET*) ALL

- Lists subscription details, including the topic string to which it subscribes

```
DIS SUB (VEGET*) ALL
SUB(VEGETARIAN.SEARCH)
SUBID(C3E2D8D4D4D8F1F1404040404 ...
DURABLE(YES)
SUBTYPE(ADMIN)
DISTYPE(RESOLVED)
TOPICSTR(Store/Veggies)
DEST(SYSTEM.MANAGED.DURABLE.CDC4 ...
DESTQMGR(MQ11)
DESTCORL(C3E2D8D4D4D8F1F1404040 ...
TOPICOBJ(FOODS.ANCHOR)
PSPROP(MSGPROP)
PUBACCT(00000000000000000000000000 ...
PUBAPPID()
PUBPRTY(ASPUB)
...
...
```

... continued

```
USERDATA()
SUBUSER(INGM000)
SUBLEVEL(1)
WSCHEMA(TOPIC)
SUBSCOPE(ALL)
DESTCLAS(MANAGED)
VARUSER(ANY)
SELECTOR()
SELTYPE(NONE)
EXPIRY(UNLIMITED)
REQONLY(NO)
CRDATE(2014-09-16)
CRTIME(16.07.06)
ALTDATE(2014-09-16)
ALTTIME(16.07.06)
END SUB DETAILS
```

Introduction to publish/subscribe

© Copyright IBM Corporation 2017

Figure 10-17. Subscription commands: DIS SUB(VEGET*) ALL

This set of slides shows the MQSC commands available to display subscriptions. MQSC commands can be used with the `rwmqsc` tool. Equivalent capabilities are provided on IBM MQ Explorer.

The display for this slide shows the three aspects:

- Queue management: DESTCLASS = MANAGED, no explicit queue definition.
- Lifetime: DURABLE. See the DEST queue name that is prefixed with SYSTEM.MANAGED.
- Creation and deletion: SUBTYPE = ADMIN, explicitly defined subscription object. A subscription that an MQSUB created would display SUBTYPE = API. Review the SUBTYPE attribute in the IBM Knowledge Center for other possible MQSUB values.

An important attribute of a subscription object is PSPROP. PSPROP mitigates version-related behaviors by dealing with the way publish/subscribe message properties are added to messages sent to the subscription. You can prevent any properties from being added by setting PSPROP to NONE. Other values are applicable to PSPROP depending on the version of publish/subscribe and the applications used. Understand your publish/subscribe environment and applications, and review PSPROP attribute values in IBM Knowledge Center.

Subscription commands: DISPLAY PUBSUB ALL

- Shows how many topics subscribed to the local queue manager

```
MQ11 DIS PUBSUB ALL      ... ... ... ... ...
DIS PUBSUB DETAILS
TYPE (LOCAL)
QMNAME (MQ11)
STATUS (ACTIVE)
SUBCOUNT (8)
TPCOUNT (13)
END PUBSUB DETAILS
```

[Introduction to publish/subscribe](#)

© Copyright IBM Corporation 2017

Figure 10-18. Subscription commands: DISPLAY PUBSUB ALL

This set of slides shows the MQSC commands available to display subscriptions. MQSC commands can be used with the `runmqsc` tool. IBM MQ Explorer can also be used to administer publish/subscribe objects.

For IBM MQ Explorer, right-click the correct queue manager, and click **Status > Publish/Subscribe**.

Subscription commands: DISPLAY SUB(*) TOPICSTR DEST

- Shows all subscriptions with the focus on the topic strings they subscribe to and the associated queue

```
DIS SUB DETAILS
  SUB (VEGETARIAN.SEARCH)
    SUBID (C3E2D8D4D4D8F1F1404040404040404040CDC456AF9B8A10A9)
    DISTYPE (RESOLVED)
    TOPICSTR (Store/Veggies)
    DEST (SYSTEM.MANAGED.DURABLE.CDC456AF9B060E32)
    END SUB DETAILS
DIS SUB DETAILS
  SUB (VEGGIE.FIND)
    SUBID (C3E2D8D4D4D8F1F1404040404040404040CDC44FBF714FEE26)
    DISTYPE (RESOLVED)
    TOPICSTR (Store/Veggies)
    DEST (VEGGIES.QLOCAL)
    END SUB DETAILS
```

Introduction to publish/subscribe

© Copyright IBM Corporation 2017

Figure 10-19. Subscription commands: DISPLAY SUB(*) TOPICSTR DEST

This set of slides shows the MQSC commands available to display subscriptions.

Subscription-related command: DISPLAY QLOCAL

- Displaying queues that are associated with subscriptions

```
dis q(SYSTEM.MANAGED.DURABLE.CDC456AF9B060E32) curdepth
QUEUE (SYSTEM.MANAGED.DURABLE.CDC456AF9B060E32)
TYPE (QLOCAL)
QSGDISP (QMGR)
CURDEPTH (0)
```

```
MQ11 DIS QLOCAL(VEGGIES) CURDEPTH
QUEUE (VEGGIES)
TYPE (QLOCAL)
QSGDISP (QMGR)
CURDEPTH (1)
```

Introduction to publish/subscribe

© Copyright IBM Corporation 2017

Figure 10-20. Subscription-related command: DISPLAY QLOCAL

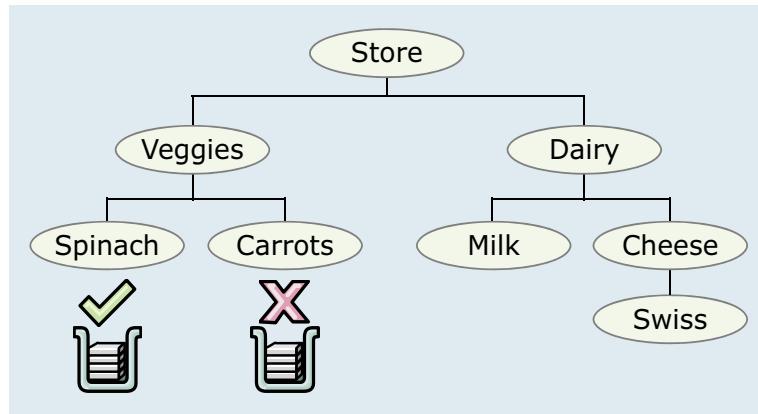
This set of slides shows the MQSC commands available to display subscriptions. MQSC commands can be used with the `runmqsc` tool. Equivalent capabilities are provided on IBM MQ Explorer.

To view the associated queues, select the Queues menu item in IBM MQ Explorer.

When looking for managed, or dynamic queues on IBM MQ Explorer, make sure to select the “Show Temporary Queues” toggle switch at the upper right of the queue display pane, then look for the SYSTEM.MANAGED.* named queues.

Publications

- What is considered a successful publication?
- IBM MQ default behavior depends on the persistence of the message and the durability of the subscription
- Persistent messages
 - If a publication cannot be delivered to one or more durable subscriptions: publish fails
 - If a publication cannot be delivered to a non-durable subscription: publish still succeeds
- Publications to non-persistent messages are deemed successful regardless of the actual outcome
- Default behavior controlled through **PMSGDLV** and **NPMSGDLV** topic attributes
- Publication ending up in the dead-letter queue is considered successful and is controlled through the **USEDLQ** topic attribute



Introduction to publish/subscribe

© Copyright IBM Corporation 2017

Figure 10-21. Publications

When you start to work with publish/subscribe it is important to determine what constitutes a successful return code on a publication, as this feedback might be misleading. Always test that a subscription was indeed received.

Topic attributes influence the handling of publications. These attributes might be inherited from the SYSTEM.BASE.TOPIC. PMSGDLV and NPMSGDLV determine how to treat persistent, or non-persistent messages during a publication.

- ASARENT: Behavior inherited.
- ALL: Message must be delivered to all subscribers. If one subscriber fails, no other subscribers receive the message and the call fails.
- ALLAVAIL: Message is delivered to all available subscribers regardless of whether any subscriber fails.
- ALLDUR: Delivery failures to non-durable subscribers do not send an error to the MQPUT call. Delivery failures to durable subscribers cause no subscribers to receive the message and the MQPUT call to fail.

USEDLQ can be set to YES or NO. If NO, or if the queue manager does not specify a DLQ value, message is treated per PMSGDLV or NPMSGDLV. If USEDLQ is set to YES, then delivery to the dead-letter queue is treated as a successful publication.

Retained publications

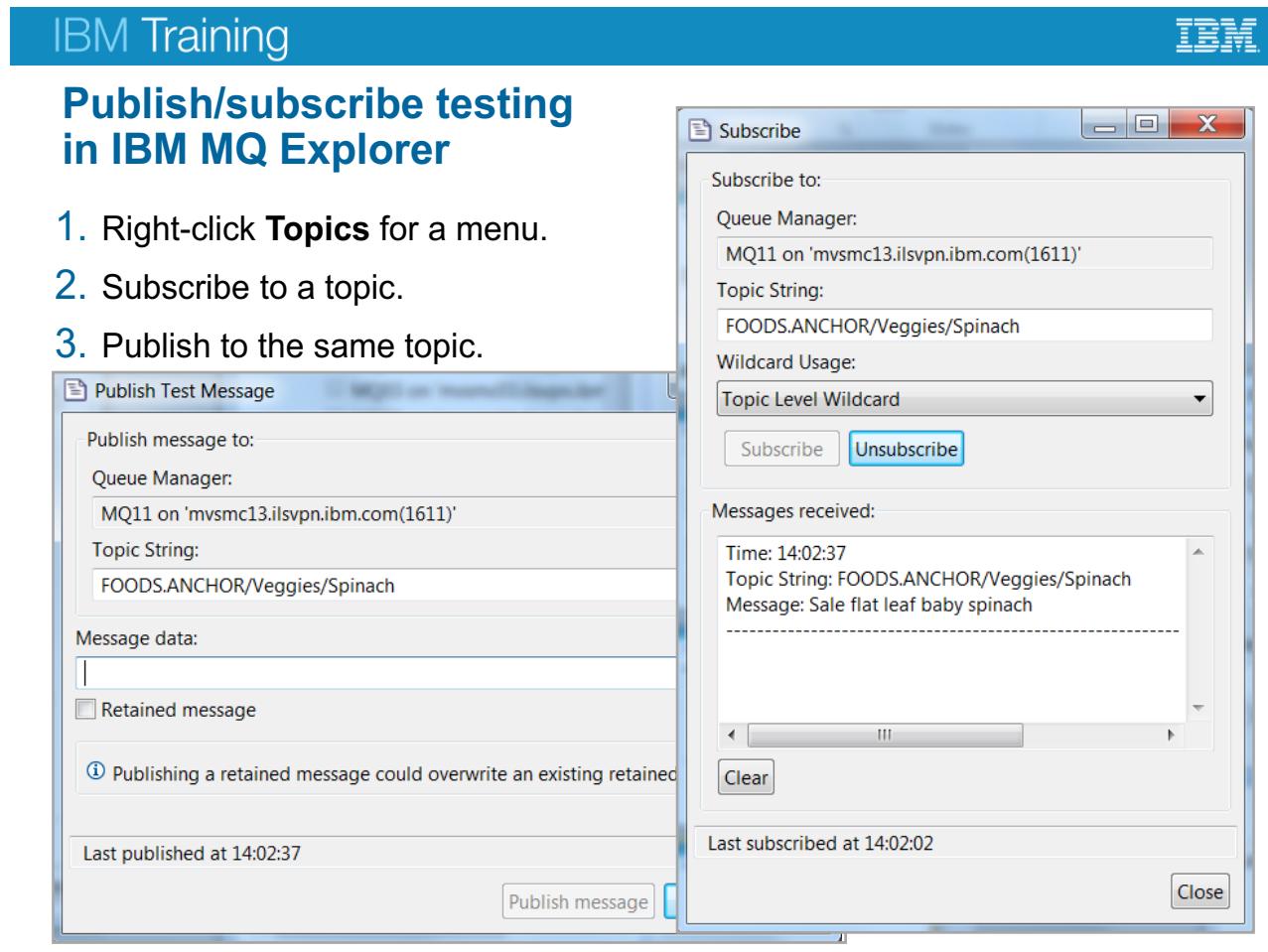
- Applications that are subscribed to the topic receive messages that are published to a topic string at the time the message is published
- Applications that subscribe to the topic after the publication occurs do not get the messages
- Retained publications
 - The queue manager retains the most recent publication for each topic
 - Applications that subscribe after the publication occurs receive the most recent retained message
- The use of retained options is for specific use cases, such as infrequent publications



Using retained publications in a large multi-queue manager environment can lead to unanticipated problems: **plan carefully** when introducing retained publications

Figure 10-22. Retained publications

Retained publications are specified by using the `MQPMO_RETAIN` MQPMO option by the application.



Introduction to publish/subscribe

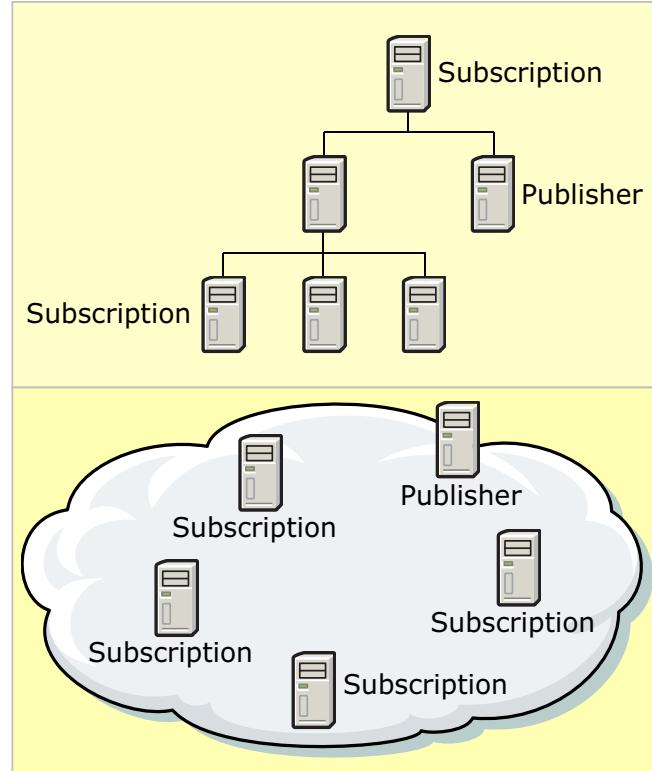
© Copyright IBM Corporation 2017

Figure 10-23. Publish/subscribe testing in IBM MQ Explorer

Another look at the IBM MQ Explorer publish/subscribe testing capabilities. When using these panes, always subscribe to the topic first.

Distributed publish/subscribe

- Extends subscriptions and publications from a single queue manager to other queue managers in the IBM MQ infrastructure
- Uses distributed IBM MQ infrastructure
- Hierarchies:
 - Indirect many-to-many connectivity
 - Direct one-to-many connectivity
- Publish/subscribe clusters:
 - Many-to-many connectivity
 - Direct routing or topic host routing many-to-many connectivity
 - Based on IBM MQ clustering



[Introduction to publish/subscribe](#)

© Copyright IBM Corporation 2017

Figure 10-24. Distributed publish/subscribe

As a recap you learned about the basics of publish/subscribe, but all the items so far, topics, subscriptions, and publications might take place within the same queue manager. In this slide, you learn about extending publish/subscribe to the IBM MQ infrastructure.

A publish/subscribe infrastructure can consist of:

- Publish/subscribe hierarchies
- Publish/subscribe clusters
- A combination of hierarchies and clusters

Publish/subscribe clusters can be direct, or topic host based.

Publish/subscribe lifecycle descriptions

- Provide a good understanding on how and why to select different attributes for the way publish/subscribe is implemented
- Aid in planning and design stages
- Three available lifecycle scenarios:
 - Managed, non-durable subscriber
 - Managed, durable subscriber
 - Unmanaged, durable subscriber

Introduction to publish/subscribe

© Copyright IBM Corporation 2017

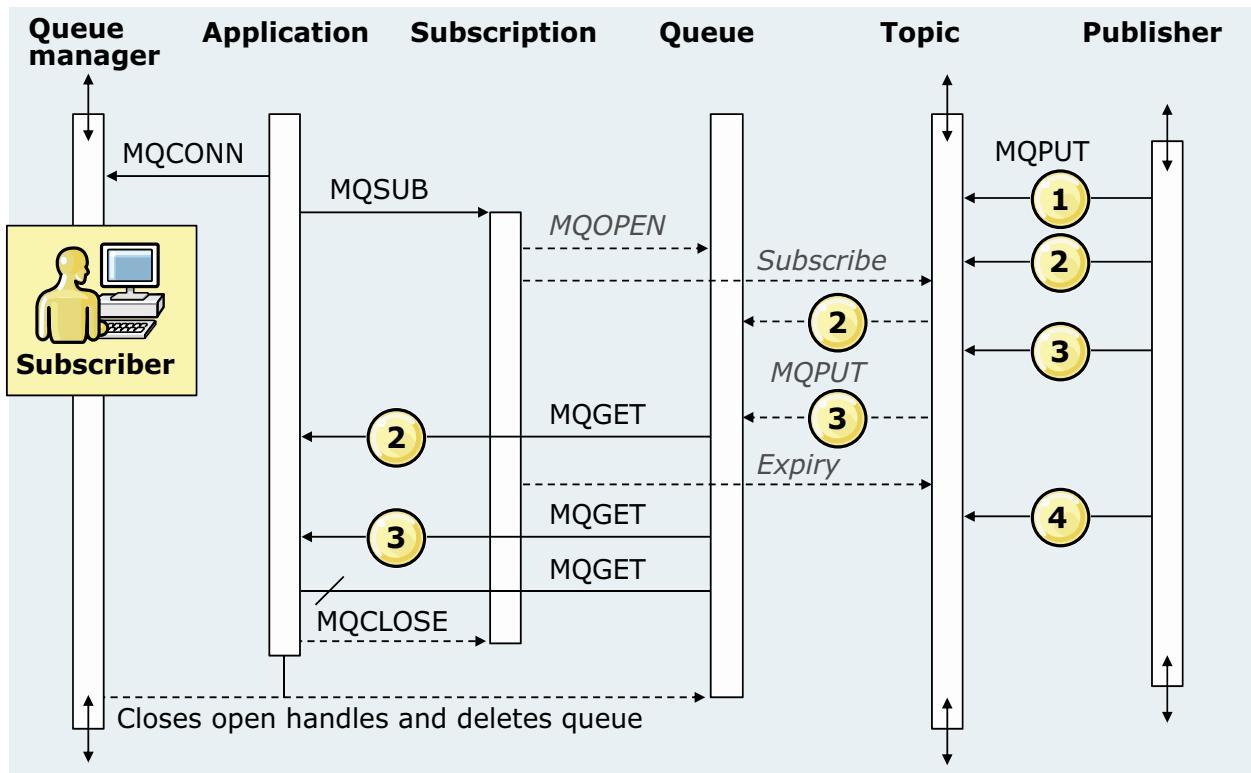
Figure 10-25. Publish/subscribe lifecycle descriptions

The publish/subscribe topic can fill over a week-long course. While this class must adhere to the basic scope, the publish/subscribe lifecycle descriptions are introduced, so you can use as a follow-up if you need to develop publish/subscribe applications. The lifecycle topics are helpful when designing and troubleshooting publish/subscribe applications.

The IBM Knowledge Center contains three lifecycle descriptions. One of these lifecycle descriptions is shown as an illustration.

IBM Training

Publish/subscribe lifecycles: Managed non-durable subscriber



Introduction to publish/subscribe

© Copyright IBM Corporation 2017

Figure 10-26. Publish/subscribe lifecycles: Managed non-durable subscriber

This slide is used to familiarize you with the lifecycle descriptions available in the IBM Knowledge Center. *The text in this slide is copied directly from the IBM Knowledge Center.*

1. The application creates a subscription on a topic that was already published to twice. When the subscriber receives its first publication, it receives the second publication, which is the currently retained publication.
2. The queue manager creates a temporary subscription queue and a subscription for the topic.
3. The subscription has an expiry. When the subscription expires no more publications on the topic are sent to this subscription, but the subscriber continues to get messages published before the subscription expired. Subscription expiry does not affect publication expiry.
4. The fourth publication is not placed on the subscription queue so the last MQGET does not return a publication.
5. Although the subscriber closes its subscription, it does not close its connection to the queue or the queue manager.

The queue manager cleans up shortly after the application ends. Because the subscription is managed and non-durable, the subscription queue is deleted

MQSUB for subscriber

```

MQSD sd = {MQSD_DEFAULT}; /* Subscription Descriptor*/
MQHOBJ Hsub = MQHO_NONE; /* object handle */
sd.Options = MQSO_CREATE
    | MQSO_NON_DURABLE
    | MQSO_FAIL_IF_QUIESCING
    | MQSO_MANAGED;
sd.ObjectString.VSPtr = argv[1];
sd.ObjectString.VSLength = (MQLONG) strlen(argv[1]);
MQSUB(Hcon,           →/* connection handle */
       &sd,          ←→/* queue object descriptor*/
       &Hobj,         ←→/* object handle (output) */
       &Hsub,         ←→/* object handle (output) */
       &S_CompCode,   ←→/* completion code */
       &Reason);     ←→/* reason code */

```

Introduction to publish/subscribe

© Copyright IBM Corporation 2017

Figure 10-27. MQSUB for subscriber

The MQSUB function call is used in integrated publish/subscribe to create a subscription.

When you need to get message for your subscription, when you use MQSUB, MQSUB passes the object handle to your MQGET call.

The MQSUB uses a subscription descriptor, or MQSC structure.

Look at the different options that are passed to the MQSUB by using the MQSD.

Subscription description structure (MQSD)

```

MQCHAR4    StrucId;          /* Structure identifier           */
MQLONG     Version;          /* Structure version number      */
MQLONG     Options;          /* Options associated with subscribing*/
MQCHAR48   ObjectName;       /* Object name                   */
MQCHAR12   AlternateUserId;  /* Alternate user identifier     */
MQBYTE40   AlternateSecurityId; /* Alternate security identifier */
MQLONG     SubExpiry;        /* Expiry of Subscription       */
MQCHARV    ObjectString;     /* Object long name             */
MQCHARV    SubName;          /* Subscription name            */
MQCHARV    SubUserData;      /* Subscription user data      */
MQBYTE24   SubCorrelId;      /* Correl Id related to this subscr */
MQLONG     PubPriority;      /* Priority set in publications */
MQBYTE32   PubAccountingToken; /* Accounting Token set in publctns*/
MQCHAR32   PubApplIdentityData; /* Appl Id Data set in publictns */
MQCHARV    SelectionString;   /* Message selector structure   */
MQLONG     SubLevel;         /* Subscription level           */
MQCHARV    ResObjectString;   /* Resolved long object name    */

```

[Introduction to publish/subscribe](#)

© Copyright IBM Corporation 2017

Figure 10-28. Subscription description structure (MQSD)

MQSD options

Access or creation options:

MQSO_CREATE

MQSO_RESUME

MQSO.Alter

Durability options:

MQSO.DURABLE

MQSO.NON.DURABLE

Destination options:

MQSO.MANAGED

MQSO.NO.MULTICAST

Scope options:

MQSO.SCOPE.QMGR

Registration options:

MQSO.GROUP.SUB

MQSO.ANY.USERID

MQSO.FIXED.USERID

Publication options:

MQSO.NOT.OWNPUBS

MQSO.NEW_PUBLICATIONS_ONLY

MQSO.PUBLICATIONS_ON_REQUEST

Read ahead options:

MQSO.READ_AHEAD_AS_QDEF

MQSO.NO_READ_AHEAD

MQSO.READ_AHEAD

Wildcard options:

MQSO.WILDCARD_CHARACTER

MQSO.WILDCARD_TOPIC

Other options:

MQSO.ALTERNATE_USER_AUTHORITY

MQSO.SET_CORRELATION_ID

MQSO.SET_IDENTITY_CONTEXT

MQSO.FAIL_IF QUIESCING

Working with topic strings

- Full topic that is composed of two fields in publish/subscribe MQI calls
 - `TOPICSTR` attribute of the topic object, which is named in the `ObjectName` field
 - `ObjectString` parameter that defines the subtopic provided by the application
- Resulting topic string is in output field `ResObjectString`

TOPICSTR	ObjectString	Full topic name	Comment
Stores/Veggies	' '	Stores/Veggies	The <code>TOPICSTR</code> is used alone
' '	Stores/Veggies	Stores/Veggies	The <code>ObjectString</code> is used alone
Stores	Veggies	Stores/Veggies	A "/" character is added at the concatenation point
Stores	/Veggies	Stores/ /Veggies	An "empty node" is produced between the two strings
/Stores	Veggies	/Stores/Veggies	The topic starts with an "empty node"

Introduction to publish/subscribe

© Copyright IBM Corporation 2017

Figure 10-30. Working with topic strings

MQOPEN for publisher

```

MQOD  od = {MQOD_DEFAULT}; /* Object Descriptor      */
MQMD  md = {MQMD_DEFAULT}; /* Message Descriptor    */
MQPMO pmo = {MQPMO_DEFAULT};/* put message options */
od.ObjectString.VSPtr=argv[1]; <= input parm topic name
od.ObjectString.VSLength=(MQLONG)strlen(argv[1]);
od.ObjectType = MQOT_TOPIC;
od.Version = MQOD_VERSION_4;

MQOPEN(Hcon,           /* connection handle      */
       &od,            /* topic object descript */
       MQOO_OUTPUT | MQOO_FAIL_IF_QUIESCING,/*opts */
       &Hobj,          /* object handle         */
       &OpenCode,        /* MQOPEN comp code     */
       &Reason);        /* reason code          */

```

Introduction to publish/subscribe

© Copyright IBM Corporation 2017

Figure 10-31. MQOPEN for publisher

You publish by using the MQPUT function call. However, the object type set by the MQOPEN for the put is to a topic, or `MQOT_TOPIC`, object type.

MQPUT for publisher

- Business as usual MQPUT to a handle that represents a topic

```

MQPUT (Hcon,           /* connection handle          */
       Hobj,            /* object handle              */
       &md,              /* message descriptor         */
       &pmo,             /* default options (datagram) */
       messlen,          /* message length             */
       buffer,           /* message buffer             */
       &CompCode,         /* completion code           */
       &Reason);        /* reason code               */

```

[Introduction to publish/subscribe](#)

© Copyright IBM Corporation 2017

Figure 10-32. MQPUT for publisher

The MQPUT looks like a put for a point-to-point application, but the object handle is a topic, instead of a queue.

Unit summary

- Differentiate between publish/subscribe and point-to-point messaging
- Describe how the history of publish/subscribe influences its functions and terminology
- Identify the basic components of publish/subscribe
- Describe key properties of topics, subscriptions, and publications
- Explain how to write an application that subscribes to a topic by using the MQSUB function call
- Describe how to code the MQOPEN and MQPUT calls to write an application that publishes to a topic

Introduction to publish/subscribe

© Copyright IBM Corporation 2017

Figure 10-33. Unit summary

Review questions (1 of 2)

1. True or False: The newest V7+ publish/subscribe implementation is referred to as *queued publish/subscribe* and *command-based publish/subscribe*.
2. Select the two best answers. You need to code a publish application. Along with passing the topic name by using the `ObjectName` or the `ObjectString` field of the object descriptor in the `MQOPEN` call, what other information do you need to provide to the object descriptor structure?
 - a. `ObjectType` needs to be set to `MQOT_TOPIC`
 - b. `DynamicQueue` name field must be set to `DURABLE`
 - c. `ObjectQMgrName` must be set to the enabled queue manager
 - d. `Version` must be set to `MQOD_VERSION_4`
3. True or False: If the `PSMODE` in the queue manager is set to `COMPAT`, the *queued publish/subscribe* interface is running.

[Introduction to publish/subscribe](#)

© Copyright IBM Corporation 2017

Figure 10-34. Review questions (1 of 2)

Write your answers here:

- 1.
- 2.
- 3.



Review questions (2 of 2)

4. How can you tell if a subscription was explicitly defined, or created with `MQSUB` in the output of `DIS SUB` or `DIS SBSTATUS` commands?
 - a. Check the displayed subscription attribute `USERDATA`
 - b. Check the displayed queue manager attribute `PSMODE`
 - c. Check the displayed subscription attribute `PSPROP`
 - d. Check the displayed subscription attribute `SUBTYPE`
5. True or False: In public/subscribe terminology, “managed” and “administered” have opposite meanings.



Introduction to publish/subscribe

© Copyright IBM Corporation 2017

Figure 10-35. Review questions (2 of 2)

Write your answers here:

4.

5.

Review answers (1 of 3)

- True or False: The newest V7+ publish/subscribe is referred to as *queued publish/subscribe* and *command-based publish/subscribe*.

The answer is: False. The V7+ publish/subscribe is referred to as integrated publish/subscribe. Queued publish/subscribe is the older version.

- Select the two best answers. You need to code a publish application. Along with passing the topic name by using the `ObjectName` or the `ObjectString` field of the object descriptor in the `MQOPEN` call, what other information do you need to provide to the object descriptor structure?
 - `ObjectType` needs to be set to `MQOT_TOPIC`
 - `DynamicQueue` name field must be set to `DURABLE`
 - `ObjectQMgrName` must be set to the enabled queue manager
 - `Version` must be set to `MQOD_VERSION_4`

The answer is: a and d. The `ObjectType` field needs to be set to `MQOT_TOPIC`, and the `Version` field must be set to `MQOD_VERSION_4`.

Introduction to publish/subscribe

© Copyright IBM Corporation 2017

Figure 10-36. Review answers (1 of 3)



Review answers (2 of 3)

3. True or False: If the `PSMODE` in the queue manager is set to `COMPAT`, the *queued publish/subscribe* interface is running.
The answer is: False. `PSMODE(ENABLED)` indicates that both the queued publish/subscribe engine, and the publish/subscribe engine are active.
4. How can you tell if a subscription was explicitly defined, or created with `MQSUB` in the output of `DIS SUB` or `DIS SBSTATUS` commands?
 - a. Check the displayed subscription attribute `USERDATA`
 - b. Check the displayed queue manager attribute `PSMODE`
 - c. Check the displayed subscription attribute `PSPROP`
 - d. Check the displayed subscription attribute `SUBTYPE`

The answer is: d. An application defined subscription shows `SUBTYPE(API)`, while a subscription object shows `SUBTYPE(ADMIN)`.



Review answers (3 of 3)

4. True or False: In public/subscribe terminology, “managed” and “administered” have opposite meanings.

The answer is: True. A managed queue is dynamically defined, while an administered topic or subscription object is explicitly defined.



Introduction to publish/subscribe

© Copyright IBM Corporation 2017

Figure 10-38. Review answers (3 of 3)

Working with publish/subscribe basics

Introduction to publish/subscribe

© Copyright IBM Corporation 2017

Figure 10-39. Working with publish/subscribe basics

Exercise objectives

- Add code to an application to use the subscription descriptor and the MQSUB function call to create a subscription
- Explain how to pass the managed queue handle from the MQSUB function call to access the managed queue in a subsequent MQGET function call
- Convert an application that puts messages to a queue to publish messages to a topic
- Use IBM MQ Explorer to obtain more details about your subscription



Introduction to publish/subscribe

© Copyright IBM Corporation 2017

Figure 10-40. Exercise objectives

Unit 11. Advanced Message Queuing Protocol (AMQP) and IBM MQ Light

Estimated time

01:00

Overview

In this unit, you learn about the Advanced Message Queuing Protocol (AMQP), IBM MQ Light, and how to exchange messages with IBM MQ. You also learn about the node.js IBM MQ Light client implementation. Next, you use what you learned about IBM MQ Light and IBM MQ publish/subscribe to exchange messages between IBM MQ and IBM MQ Light. The unit includes mapping considerations between IBM MQ and IBM MQ Light, and how to enable an IBM MQ queue manager to use AMQP channels. You also learn how to use IBM MQ publish/subscribe sample programs to test the exchange of messages between a queue manager and IBM MQ Light.

How you will check your progress

Accountability:

- Review questions
- Lab Exercises

References

IBM Knowledge Center for IBM MQ V9 documentation

IBM MQ Light developer tools: <https://developer.ibm.com/messaging/mq-light/docs/>

IBM MQ Light API and clients: <https://developer.ibm.com/messaging/mq-light/docs/api-reference>

IBM MQ Light downloads: <https://developer.ibm.com/messaging/ibm-mq-light-downloads/>

npm package manager for JavaScript mqlight client: <https://www.npmjs.com>

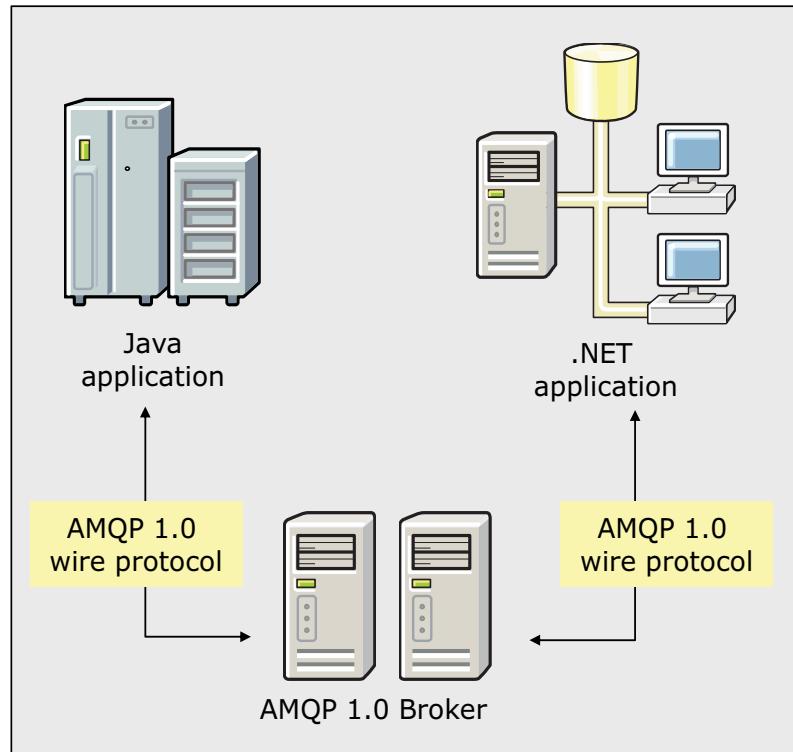
Unit objectives

- Describe the Advanced Message Queuing Protocol
- Describe typical IBM MQ application and IBM MQ Light application interface scenarios
- Describe basic IBM MQ Light concepts and list the components
- Describe the quality of service categories available with IBM MQ Light
- Describe how to write code to send and receive messages between IBM MQ applications and IBM MQ Light node.js applications
- Describe how to map headers and properties between IBM MQ applications and AMQP applications
- Explain how to enable and configure IBM MQ – IBM MQ Light interface
- Describe the commands that are used to check the IBM MQ AMQP channel connections
- Explain where to locate the logs that hold IBM MQ Light-related information
- Describe the security options for IBM MQ Light

Advanced Message Queuing Protocol (AMQP)

An open Internet (“wire”) protocol standard for message-queuing communications

OASIS
ISO 19464



Advanced Message Queuing Protocol (AMQP) and IBM MQ Light

© Copyright IBM Corporation 2017

Figure 11-2. Advanced Message Queuing Protocol (AMQP)

The Advanced Message Queuing Protocol, or AMQP, is an Internet protocol that was developed as ISO 19464 with the objective of further simplifying enterprise connectivity. It is a wire protocol, in that it allows programs that run in different operating systems to interconnect with each other regardless of the difference in platforms.

AMQP advocates the use of open source, and it does not contain proprietary components.

The significant picture for AMQP is:

- Ensure that all devices can connect to ISO 19464 networks
- Access data by using any application
- To connect either business or government organizations in a secure and cost-efficient way

When you first learn about AMQP, it might seem like one more standard until you review the success and following experienced by ISO 19464.

The available versions – AMQP, AMQP 0.9.1, and AMQP 1.0 – are significantly different.

AMQP 0.9.1, implemented by other vendors, includes features that describe how messages should be routed. AMQP 1.0, used by IBM MQ Light, concentrates on how you get a message from the client to the broker.



Attention

An AMQP 1.0 client application is not compatible with an AMQP 0.9.x client application.

Partial list of AMQP products and adopters

Products

- Apache Qpid, an Apache project
- Fedora Linux AMQP Infrastructure
- IIT Software's SwiftMQ
- INETCO's AMQP protocol analyzer
- JORAM: Open reliable asynchronous messaging, 100% pure Java implementation of JMS
- Kaazing's AMQP Web Client
- Microsoft Windows Azure Service Bus
- JBoss A-MQ by Red Hat built from Qpid
- StormMQ a cloud hosted messaging service based on AMQP
- VMware Inc RabbitMQ, also supported by SpringSource
- **IBM MQ Light**

Source: <http://www.amqp.org>

Over 500 commercial users

- The Deutsche Börse
- JPMorgan
- National Science Foundation
- NASA
- Red Hat
- VMware
- Google
- UIDAI, Government of India
- Mozilla
- OpenStack
- AT&T
- INETCO
- Smith Electric Vehicles
- IBM

Advanced Message Queuing Protocol (AMQP) and IBM MQ Light

© Copyright IBM Corporation 2017

Figure 11-3. Partial list of AMQP products and adopters

The partial list of existing AMQP implementations, which have an impressive list of commercial users, is a testament to the popularity and growth of the protocol.

IBM is one of the companies that offer AMQP implementations, with products such as IBM MQ Light and IBM Message Hub. The rest of this unit focuses on IBM MQ Light, and how IBM MQ applications can interface with IBM MQ Light.

IBM MQ Light

- Developer-centric messaging that is designed to create responsive and scalable applications
- 5 minutes from download to code removes delays and dependency on infrastructure services
- Developer-focused tools
- Flexibility to use a choice of AMQP1.0 client implementations

[Advanced Message Queuing Protocol \(AMQP\) and IBM MQ Light](#)

© Copyright IBM Corporation 2017

Figure 11-4. IBM MQ Light

You can think of IBM MQ Light as a message gateway or broker implementation of the AMQP 1.0 protocol. IBM MQ Light gives you the ability to create scalable micro-services by easily offloading work from your application to IBM MQ Light for asynchronous processing.

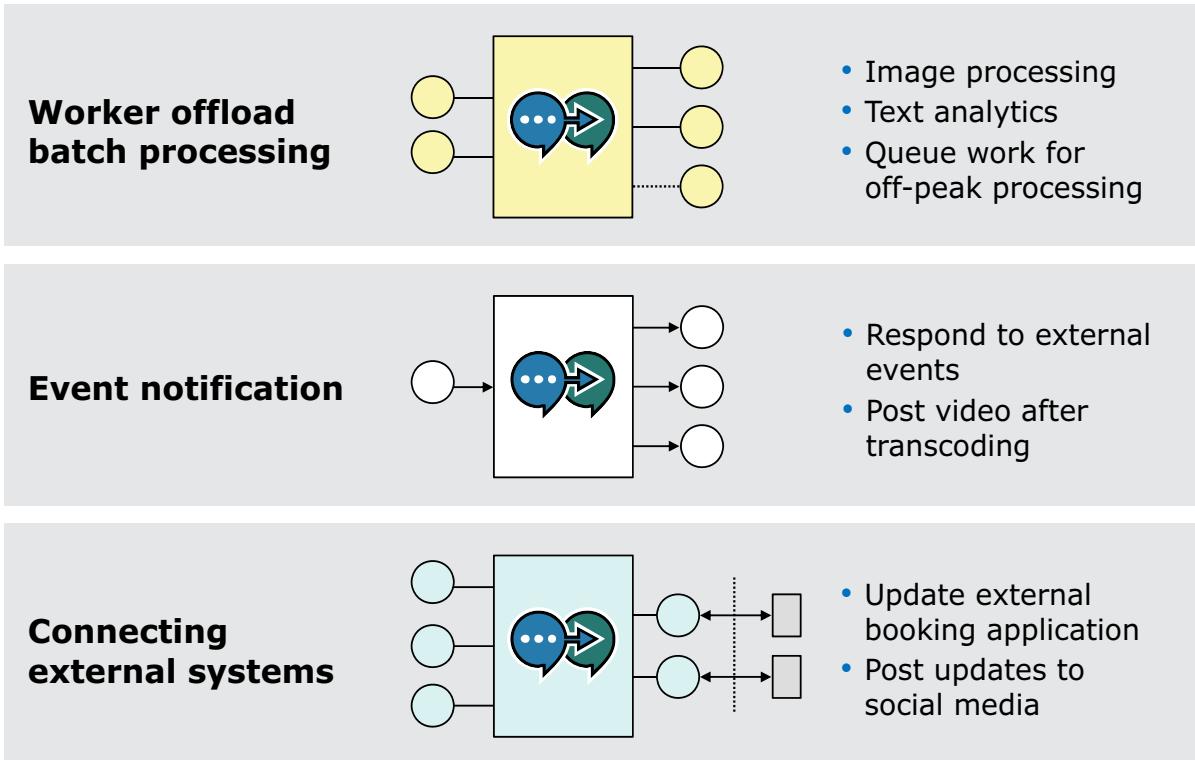
IBM MQ Light uses publish/subscribe technology to route messages from sender to destination, or rather from publisher to subscriber.

Another focus area for IBM MQ Light was to provide tools that the developer might use to track the results of sending and receiving messages with a simple, intuitive approach, and to aid problem diagnosis of your application.

The API for IBM MQ Light implementations is available in several programming languages that provide extra flexibility for programmers and organizations. IBM MQ Light provides developers the ability to write and deploy scalable applications.

IBM MQ Light applications can be deployed, and can interface with an IBM MQ manager. IBM MQ Light also interfaces with other non-IBM AMQP clients. However, you must ensure that the AMQP version that is used to implement the non-IBM client is compatible with AMQP 1.0.

Some use cases for IBM MQ Light



Advanced Message Queuing Protocol (AMQP) and IBM MQ Light

© Copyright IBM Corporation 2017

Figure 11-5. Some use cases for IBM MQ Light

IBM MQ Light can benefit the infrastructure by its ability to distribute work and connect external systems.

- An application might receive above average traffic, yet it needs to keep response times to meet service agreements. IBM MQ Light can be associated with worker applications to which IBM MQ Light can distribute the workload. When a spike in work occurs, the number of worker applications can be increased without requiring changes to the applications or to IBM MQ Light.
- A variation of the offload process is the ability to batch work for later processing, which can also be accomplished by using IBM MQ Light.
- IBM MQ Light can also be used for event notification. In this scenario, IBM MQ Light serves as a “man-in-the-middle” to send updates to a messaging service that can manage the storage and distribution of messages to the interested applications.
- With AMQP as its messaging core, IBM MQ Light is a perfect hub or gateway with which to connect to other devices and enterprise systems.

In the TUTORIALS section of the IBM DeveloperWorks site, the examples walk through these scenarios. The tutorials can be found at: <https://developer.ibm.com/messaging/mq-light/docs/>.

IBM MQ Light concepts

- All data is carried as **messages** that might contain:
 - Data that is exchanged between applications such as text, XML, JSON, binary
 - IBM MQ Light attribute name-value pairs
- **Topics** that route messages to interested applications:
 - Topic can be single string such as Sports
 - Topic can be a hierarchy, such as Sports/Soccer/OrlandoSC
 - Can use wildcards to subscribe
 - Multi-level wildcard #
 - Single-level wildcard +
- **Destinations** are associated with:
 - Time-to-live (TTL)
 - A topic, hierarchy, multi-level, or single-level wildcard
 - Applications have exclusive or **shared** use of a destination
- **Message delivery assurance:** Quality of service (**qos**) option

Sports/Soccer/#
Sports/Soccer/OrlandoSC
Sports/Soccer/Orlando

Sports/+/Orlando
Sports/Soccer/Orlando
Sports/Basketball/Orlando

Figure 11-6. IBM MQ Light concepts

The following concepts describe details about IBM MQ Light.

- **Messages.** The data that is exchanged in IBM MQ Light is in the form of messages. These messages can contain XML, text, JSON, or binary formats.
- **Topics.** Topics help route messages to subscribers by using an agreed-upon string or hierarchy of strings that are governed by needs of the enterprise. Topics can also be subscribed to on a pattern basis by using a multiple or single level wildcard. Pattern-based topics help subscribers select publications that are grouped by topic level.
 - The number sign # or **multilevel wildcard** allows selection of all topics that match the selected string regardless of the content of the wildcard. For example, selecting Sports/Soccer/# would select Sports/Soccer/OrlandoSc, Sports/Soccer/Orlando, Sports/Soccer/Tampa, and other hierarchies that start with "Sports/Soccer". Sports/#/Orlando would select topics such as Sports/Soccer/Orlando and Sports/Baseball/Orlando. The multilevel wildcard can be used in more than one level in the topic hierarchy. The multilevel wildcard must be specified on its own, such as Sports/Soccer#.
 - Sport/Soccer#/ is incorrect.

- The plus sign `+` or **single level wildcard** matches one level of the pattern-based topic only. For example, selecting Sports/`+`/Orlando would return Sports/Soccer/Orlando, Sports/Baseball/Orlando, and Sports/Basketball/Orlando.
- **Destinations** are related to a topic or topics by using a pattern to subscribe. Applications obtain messages from destinations.
 - A number of destinations can be associated with a topic.
 - A messaging service can create or remove a destination automatically when an application uses the APIs that the messaging service provides.
 - Destinations are created with a time-to-live value. The time-to-live value is reset when an application uses the destination.
 - If a destination remains unused for a time that exceeds the time-to-live value, IBM MQ Light removes the destinations and messages that are held in the destination.
- **Time-to-live**, or `ttl`, is the message lifetime (in milliseconds) which is applied to both the message and the destination that a client is subscribed to. The way time-to-live is set or reset varies according to the method used and existing conditions. You learn more about time-to-live later in this unit.
- **Delivery assurance** is the **quality of service**, or QoS, applied between the client and the server. IBM MQ Light supports two quality of service options:
 - **At most once** is the initial default value. With `At most once` quality of service, if the recipient does not receive the message, the message is never transferred more than one time. `At most once` quality of service is appropriate for scenarios where data loss is tolerated.
 - **At least once**. With `At least once` quality of service, the sender must get a confirmation that the message was received before the message is considered successfully transferred. `At least once` quality of service is for situations where data loss is unacceptable. When `At least once` is used, a message might arrive, as the option states, more than one time. Applications receiving messages that use `At least once` QoS must handle the possibility of a message that arrives more than one time to mitigate unwanted results.

Quality of service is specified in the methods that send and subscribe to messages. You learn more about quality of service later in this unit.

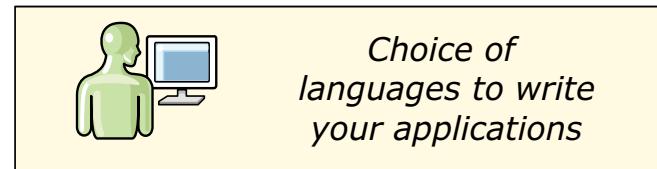
IBM MQ Light components

- IBM MQ Light servers
 - Stand-alone IBM MQ Light server
 - IBM MQ server Enterprise
 - Requires V8.0.0.4 build or higher

- IBM MQ Light SDK

- IBM MQ Light API and sample clients
<https://developer.ibm.com/messaging/mq-light/docs/api-reference/>
 - Node.js
 - Java
 - Ruby
 - Python
 - Implementations that use OASIS Standard AMQP 1.0 wire protocol such as any of the Apache Qpid Proton clients

IBM MQ
contains AMQP
channel type



Advanced Message Queuing Protocol (AMQP) and IBM MQ Light

© Copyright IBM Corporation 2017

Figure 11-7. IBM MQ Light components

The primary IBM MQ Light engine is referred to as an IBM MQ Light server, and sometimes as the “broker”. As of the time that this course was written, two *IBM MQ Light servers are available*:

1. The “stand-alone” IBM MQ Light server. This server can be obtained by using your normal software provisioning channels such as Passport Advantage. The stand-alone server:
 - Connects to AMQP or IBM MQ Light clients
 - Is a single user server, one user defined
 - Tools consist of the Web UI



Information

You can also obtain a no-cost developer copy by searching the IBM web pages for “IBM MQ Light downloads”. As of the time that this course was written, the URL for the no-cost developer download is:

<https://developer.ibm.com/messaging/ibm-mq-light-downloads/>

2. The IBM MQ V8.0.0.4 manufacturing refresh and higher versions of IBM MQ provide the support for the IBM MQ Server Enterprise version that includes support for IBM MQ Light. The IBM MQ Enterprise version:

- Interfaces with IBM MQ Light applications and other AMQP client applications
- Uses all enterprise features of IBM MQ
- Tools include the IBM MQ administrative options, and include PCF and Tivoli

Other IBM MQ Light components are the IBM MQ Light SDK, the API, and sample clients available in several popular languages. The sample clients can be used as a starting point to learn the API. The IBM MQ Light clients can be obtained from the package manager for the language that you are using, for example: npm for Node.js, rubygems.org for Ruby, or Maven for Java.

The IBM MQ Light API supports the following messaging features:

- At-most-once and at-least-once message delivery
- Destination addressing by using a topic string
- Message and destination durability
- Shared destinations to allow multiple subscribers to share workload
- Client takeover to resolve unresponsive clients
- Ability to read ahead messages that can be configured
- Ability to acknowledge messages can be configured

In this unit, you focus on the IBM MQ support for AMQP. You explore scenarios where IBM MQ and IBM MQ Light (or any other AMQP client) interface.



Reminder

This unit concentrates on the IBM MQ interface with IBM MQ Light. Always refer to IBM Knowledge Center for more use cases such as HA configuration and sending data to message-driven beans.

IBM MQ Light API for node.js: `mqlight.createClient()`

```
mqlight.createClient([options], [callback])
```

- `options`
 - `service`
 - `id`
 - `user`
 - `password`
 - `sslTrustCertificate`
 - `sslVerifyName`
- `callback`

Source:

mqlight client at <https://www.npmjs.com>

Advanced Message Queuing Protocol (AMQP) and IBM MQ Light

© Copyright IBM Corporation 2017

Figure 11-8. IBM MQ Light API for node.js: `mqlight.createClient()`



Information

IBM MQ Light can be used with a choice of languages. In this course, node.js samples were selected. You can also use Java, Python, and Ruby. Check the IBM MQ Light documentation, as extra languages might be added.

To send or receive messages, you must first create a client by using the `mqlight.createClient` method. `mqlight.createClient` returns a Client object that represents the client instance. The client that is created is an event emitter. Listeners can be registered for the following events: started, stopped, restarted, error, drain, malformed, and message.

The syntax for this method is: `mqlight.createClient([options], [callback])`

The properties or parameters that are passed to this method are a combination of positional and name-value pair parameters. The `service` property of the `mqlight.createClient` options is required. The remaining properties are optional.

The `options` parameter contains different properties for the client.

- The `service` property consists of a string that can contain the URL, or an array that contains a list of URLs to attempt to connect to.
 - The user name and password might be embedded into the URL, such as:
`amqp://user1:passw0rd@hostname.ibm.com`
 - If you use the SSL properties that are detailed later in this slide, the service URL must contain `amqps` instead of `amqp`.
- `ID` is an optional string that serves as a unique identifier for this client. Only one client that uses the value of this property can connect to an IBM MQ Light server. Instances of the client (as identified by the value of this property) can be connected to an IBM MQ Light server at a specific point in time. If another instance of the same client connects, then the previously connected instance is disconnected.
- `user` is an optional user name that is used for authentication. Alternatively, the user name can be embedded in the URL passed in the `service` property. If you specify a name for this property, and also embed the name in the URL, it results in an error. If a user name is specified, a password must be also specified or the method results in an error.
- `password` is an optional string for authentication. Alternatively, a password can be embedded in the URL that is passed by using the `service` property.
- `sslTrustCertificate` is an optional string that is used when SSL authentication is required for the IBM MQ Light server and the `service` specifies `amqps`.
- `sslVerifyName` is an optional Boolean value that indicates whether to check the IBM MQ Light server's common name in the certificate to see whether it matches the actual server DNS name.
- `callback` is an optional function that can be started to indicate success or failure.



Information

To learn more about the IBM MQ Light API, search the IBM developer for your preferred language at: developer.ibm.com.

As of the time that this course was written, the links to the different APIs are at:
<https://developer.ibm.com/messaging/ibm-mq-light-downloads/>.

IBM MQ Light API for node.js: `mqlight.Client.send()`

```
mqlight.Client.send(topic, data, [options], [callback])
```

- `topic`
- `data`
- `options`
 - `qos`
 - `ttl`
 - `properties`
- `callback`
 - `error`
 - `topic`
 - `data`
 - `options`

Source: *mqlight client at <https://www.npmjs.com>*

Advanced Message Queuing Protocol (AMQP) and IBM MQ Light

© Copyright IBM Corporation 2017

Figure 11-9. IBM MQ Light API for node.js: `mqlight.Client.send()`

The syntax for this call is: `mqlight.Client.send(topic, data, [options], [callback])`

The `mqlight.createClient` method sends the value that is specified in the data argument to the specified topic. The `quality of service`, `qos` and `time-to-live`, `ttl` properties can be specified in this method.

- `topic` is a string that contains the topic where the message is sent.
- `data` contains the message to send.
- `options` contains more properties.
 - `qos` is an optional number that contains the quality of service to use for the message. The initial default value is 0, which denotes at most once. Use 1 for the at least once quality of service.
 - `ttl` is an optional number in milliseconds, which contains the time-to-live specification for the message. This value must be greater than zero, or the result is an error.
 - `properties` is an optional set of key-value attribute pairs that are carried with the message. These values must not be null, and can be Boolean, number, string, or Buffer.
- `callback` is a function that is required when the `qos` property is at least once. If the `qos` property is at most once, the options argument is optional.

IBM MQ Light applications can send and receive string messages, byte buffers, binary messages, and JSON objects. The message type is preserved in the queue manager. If the publishing application sends a JSON object, the receiving application receives a JSON object.



Information

To learn more about the IBM MQ Light API, search the IBM developer for your preferred language at: developer.ibm.com.

As of the time that this course was written, the links to the different APIs are at:
<https://developer.ibm.com/messaging/mq-light/docs/api-reference/>.

IBM MQ Light API send sample as implemented in node.js

*Source:
mqlight client at <https://www.npmjs.com>*

- The client is created in starting state
- **service** has a URL or list of URLs for the service to connect to
- The client is considered an “event emitter” for which listeners can register for specific events

Positional options show service property

```
var mqlight = require('mqlight');
var sendClient =
  mqlight.createClient({service:
    'amqp://localhost'});
var topic = 'Sports/soccer';
sendClient.on('started', function() {
  sendClient.send(topic, 'Orlando 3,
    Miami 1', function (err, data) {
      console.log('Sent: %s', data);
      sendClient.stop();
    });
});
```

Advanced Message Queuing Protocol (AMQP) and IBM MQ Light

© Copyright IBM Corporation 2017

Figure 11-10. IBM MQ Light API send sample as implemented in node.js

In an earlier slide, you looked at the syntax of the `mqlight.createClient` and `mqlight.Client.send` methods. The syntax for the respective methods is:

- `mqlight.createClient([options], [callback])` and
- `mqlight.Client.send(topic, data, [options], [callback])`

In this slide, you see a snippet of code that is adapted from the node.js send sample, which implements these methods.

- In the `createClient` method, you see the required `service` attribute as `localhost`. If a port is not specified, the default is 5672.
- In the `sendClient` method, you see the topic followed by the data.
 - `qos` is not specified and the client uses the initial default of `At most once` quality of service.
 - `ttl` is not specified. For the `sendClient` method, the initial default value is 7 days in milliseconds.
 - The maximum `ttl` value that can be specified is 30 days.

IBM MQ Light API node.js: `mqlight.Client.subscribe()`

```
mqlight.Client.subscribe(topicPattern, [share],  
[options], [callback])  
  • topicPattern  
  • share  
  • options  
    - autoConfirm  
    - credit  
    - qos  
    - ttl  
  • callback  
    - error  
    - topicPattern  
    - share
```

Source: IBM MQ Light client at <https://www.npmjs.com>

Advanced Message Queuing Protocol (AMQP) and IBM MQ Light

© Copyright IBM Corporation 2017

Figure 11-11. IBM MQ Light API node.js: `mqlight.Client.subscribe()`

The `mqlight.Client.subscribe` method is used to receive messages by subscribing to a topic or topic pattern, and optionally share arguments. `on.message` events are emitted when messages arrive.

- `topicPattern` is a required string that the messaging service matches against when a message is sent or published. This pattern can contain multilevel or single-level wildcards.
- `share` is an optional property that allows sharing of workload by receiving messages from a shared destination. When the `share` attribute is omitted, only a specific instance of this client can receive messages. The initial value is a private destination, not shared.

- `options` is an extra object that allows more properties to be set for the subscription.
 - `autoConfirm` is an optional Boolean value that is applicable only if the `qos` property is set to At least once. When `autoConfirm` is set to true, which is the initial default value, the client automatically confirms delivery of messages when all of the listeners that are registered for the client message event are returned. When set to false, application code is responsible for confirming the delivery of messages by using the `confirmDelivery` method.
 - `credit` is an optional number, which represents the maximum number of unconfirmed messages a client can have before the server stops delivery of new messages to the client.
 - `qos` is an optional number that indicates the subscription quality of service. The default, 0, is for At most once `qos`. The second allowed value is 1 for At least once `qos`.
 - `ttl` is a number with the optional time-to-live value, which is applied to the destination that the client is subscribed to. If specified, the value replaces any previous value when the destination pre-exists. Time-to-live starts the countdown when no instances of a client are subscribed to a destination, and is reset each time a new instance of the client subscribes to the destination. IBM MQ deletes the destination by discarding messages when the countdown reaches 0. The initial default value is 0, which means the destination is deleted as soon as no clients are subscribed to it.
- `callback` is an optional function that `callback` is to be notified when the `subscribe` operation completes. The `callback` function is passed the following arguments:
 - `error` is an object when `callback` is being requested. It indicates that the `subscribe` call failed. If the `subscribe` call completes successfully, then the value `null` is supplied for this argument.
 - `topicPattern` is an argument that is supplied to the corresponding `subscribe` method call.
 - `share` is present if it was supplied to the corresponding `subscribe` method call.



Information

To learn more about the IBM MQ Light API, search the IBM developer for your preferred language at: developer.ibm.com

As of the time that this course was written, the links to the different APIs are at:
<https://developer.ibm.com/messaging/mq-light/docs/api-reference/>.

IBM MQ Light API receive sample as implemented in node.js

Source:
IBM MQ Light client at <https://www.npmjs.com>

Positional options show
service property

- Subscribe call subscribes the client to a topic or topic pattern and share arguments

```
var mqlight = require('mqlight');  
  
var recvClient =  
  mqlight.createClient({service:  
    'amqp://localhost'});  
var topicPattern = 'Sports/soccer';  
recvClient.on('started', function() {  
  recvClient.subscribe(topicPattern);  
  recvClient.on('message', function(data  
, delivery) {  
    console.log('Recv: %s', data);  
  });  
});
```

Advanced Message Queuing Protocol (AMQP) and IBM MQ Light

© Copyright IBM Corporation 2017

Figure 11-12. IBM MQ Light API receive sample as implemented in node.js

The syntax for the `mqlight.Client.subscribe` method, which allows clients to receive messages for which they registered interest, is:

```
mqlight.Client.subscribe(topicPattern, [share], [options], [callback])
```

The receive or subscribing client also starts by using the `mqlight.createClient` method, followed by the receiving `mqlight.Client.subscribe` method.

The `service` attribute for the `mqlight.createClient` method in this display shows that this client is also connecting to the default port in the local host because a port is not specified.

- The required `topicPattern` is set as “Sports/soccer”.
- The absence of the `share` attribute denotes a private subscription.
- All option parameter attributes take the initial default values.
- No callback attribute is specified.



Information

To learn more about the IBM MQ Light API, search the IBM developer for your preferred language at: developer.ibm.com

As of the time that this course was written, the links to the different APIs are at:
<https://developer.ibm.com/messaging/mq-light/docs/api-reference/>.

IBM MQ and IBM MQ Light scenario terminology baseline

- IBM MQ V7 and later publish/subscribe application
 - Uses the integrated publish/subscribe engine
 - Is active if the PSMODE attribute of the queue manager is set to either ENABLED or COMPAT

- IBM MQ queued publish/subscribe
 - Older version of publish/subscribe
 - Applications require a publish/subscribe RFH2 header
 - Is active if the PSMODE attribute of the queue manager is set to ENABLED

- IBM MQ queue-based application is an application that currently:
 - Puts messages to a queue by using MQPUT, or gets messages from a queue by using MQGET
 - Does not use any form of publish/subscribe
 - Underlying functionality is enabled through IBM MQ publish/subscribe-related object definitions



[Advanced Message Queuing Protocol \(AMQP\) and IBM MQ Light](#)

© Copyright IBM Corporation 2017

Figure 11-13. IBM MQ and IBM MQ Light scenario terminology baseline

In the first IBM MQ and IBM MQ Light scenario, you looked at IBM MQ Light, and possibly any other AMQP client use IBM MQ as the messaging provider, when all the code is IBM MQ Light API, and not IBM MQ code.

Next, you look at scenarios where IBM MQ and IBM MQ Light exchange messages.

The evolution of publish/subscribe might create confusion with the terminology used. This slide reviews and establishes a baseline of the terminology that is used in IBM MQ publish/subscribe.

- **IBM MQ V7 and higher** publish/subscribe refer to the IBM MQ integrated publish/subscribe engine, or broker. This engine is active if the PSMODE attribute of the queue manager is either ENABLED or COMPAT. You look at how IBM MQ V7 and higher and IBM MQ Light interface.
- **“Queued” publish/subscribe** refers to the older implementations of publish/subscribe where publish/subscribe information was included in the RFH2 header.
 - Queued publish/subscribe is enabled when the PSMODE attribute of the queue manager is set to ENABLED.
 - Queued publish/subscribe is not active when PSMODE is set to COMPAT.
 - Use of queued publish/subscribe is similar to the use of IBM MQ V7, except that the IBM MQ application must be coded with publish/subscribe attributes in the RFH2 header.
 - Refer to the IBM MQ V7 scenarios for queued publish/subscribe - IBM MQ Light use.

- **IBM queue-based application** refers to an IBM MQ application that puts and gets messages to and from queues, and does not use publish/subscribe.

When referring to applications, it is important to distinguish between “queued” publish/subscribe, queue-based applications, and integrated or V7 and later publish/subscribe.

Special terminology clarification

- The next few slides might refer to IBM MQ V7, or IBM MQ V7 ***applications***, that is:
 - Applications that are coded for the integrated (not queued) publish/subscribe
 - Applications that use the “native” or publish/subscribe MQI
 - Applications where no publish/subscribe administered object definitions are required
- The queue manager that exchanges these messages must be a V8.0.0.4 queue manager, at command level 801 or higher
 - The course lab environment uses a command level of 902

Figure 11-14. Special terminology clarification

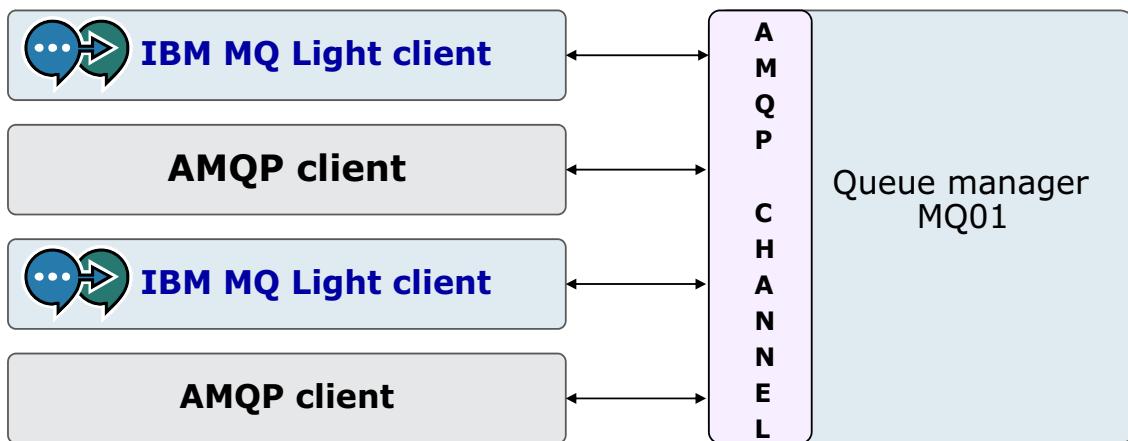
The use of “IBM MQ V7” exclusively refers to the MQI *with which the IBM MQ applications were developed*. IBM MQ V7 made the “native”, or publish/subscribe API available for integrated publish/subscribe. With the “native” publish/subscribe API, applications do not require publish/subscribe predefined objects such as subscriptions or topics, and can use “managed” queues (that is, dynamic queues). The use of V7 to describe the way that the application was developed *does not imply* that you can exchange messages with an IBM MQ Light or other AMQP application.

You must use IBM MQ V8.0.0.4 or higher to exchange messages for the scenarios that follow this slide.

Use of “IBM MQ V7”, versus using the “native MQI”, was kept to be consistent with information you might find in the IBM Knowledge Center, or other publications and blogs that use the same terminology.

IBM MQ as a messaging provider for AMQP applications

- IBM MQ Light or other AMQP clients are the only applications that send or receive messages
- Messages are sent to topics or topic patterns
- Applications subscribe to a topic string or pattern
- Applications can choose certain attributes such as quality of service, and time-to-live



Advanced Message Queuing Protocol (AMQP) and IBM MQ Light

© Copyright IBM Corporation 2017

Figure 11-15. IBM MQ as a messaging provider for AMQP applications

This example of IBM MQ and IBM MQ Light, or other AMQP client interface, is the case where IBM MQ is used as the messaging provider for the AMQP applications.



Information

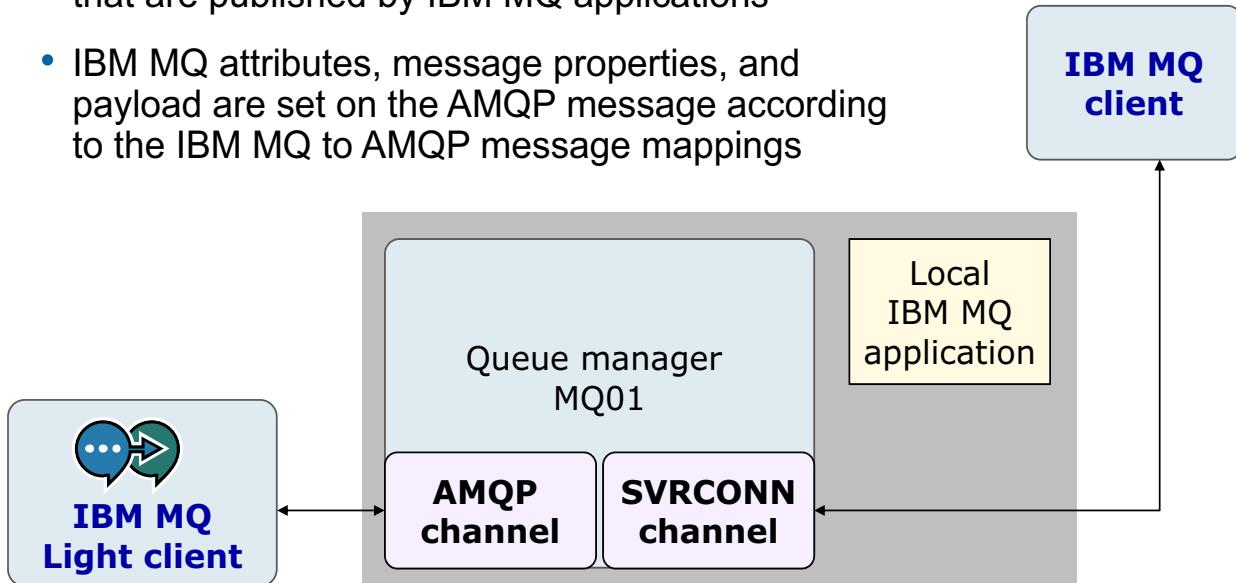
In this unit, an IBM MQ Light client can be a topic producer or a topic consumer.

In this example, IBM MQ is acting as the “broker” and messaging provider. You look at the configuration for the AMQP channel later in this unit. First, you look at the node.js version of the IBM MQ Light sample applications, and see how IBM MQ Light messages are sent and received.

Since the node.js examples are used, `npm` and `node.js` are assumed to be installed. You can find information for `node.js`, or any other available language you choose in the IBM MQ Light reference link that is provided later in this unit.

Exchanging messages with IBM MQ V7 and later

- IBM MQ applications can receive messages that are published by IBM MQ Light applications
- IBM MQ Light applications can receive messages that are published by IBM MQ applications
- IBM MQ attributes, message properties, and payload are set on the AMQP message according to the IBM MQ to AMQP message mappings



Advanced Message Queuing Protocol (AMQP) and IBM MQ Light

© Copyright IBM Corporation 2017

Figure 11-16. Exchanging messages with IBM MQ V7 and later

In this scenario, you look at how to exchange messages between IBM MQ V7 and later and IBM MQ Light applications.

In a previous unit, you learned how to use the IBM MQ API to publish and subscribe to IBM MQ topics. In this IBM MQ V7+ integrated native publish/subscribe MQI scenario, you see how to use the IBM MQ publish/subscribe API and the IBM MQ Light API methods to exchange messages across IBM MQ applications and IBM MQ Light applications.



Note

The use of IBM MQ V7 refers to the publish/subscribe native MQI application only.

The scenarios that are presented assume that AMQP is configured in the queue manager. Later in this unit, you learn how the IBM MQ Light support is configured for IBM MQ. First, you look at how the interfaces work.

Send messages from IBM MQ Light to IBM MQ

IBM MQ Light application



- IBM MQ Light application uses the `client.send(...)` method to publish a message

```
client.send("Sport/data",
    "Orlando 3 Miami 1", { qos:
        mqlight.QOS_AT_LEAST_ONCE },
    sendResultCallback);
```

IBM MQ V7 or later application

- Create a subscription descriptor
- Set basic subscription options
- Specify the topic string to subscribe

```
MQSD sd = {MQSD_DEFAULT};
sd.Options = MQSO_CREATE
    | MQSO_NON_DURABLE
    | MQSO_FAIL_IF_QUIESCING
    | MQSO_MANAGED;
sd.ObjectString.VSPtr = "Sport/data";
sd.ObjectString.VSLength =
(MQLONG)strlen("Sport/data");
```

- MQSUB to create subscription
- MQGET to consume messages

```
MQSUB(Hcon, &sd, &Hobj, &Hsub, &SubCC,
&SubRC)
MQGET(Hcon, Hobj, &md, &getMsgOpts,
buflen, buffer, &msgLen, &GetCC,
&GetRC)
```

Figure 11-17. Send messages from IBM MQ Light to IBM MQ

In this use case, an IBM MQ V7 application is interested in information that is published by an IBM MQ Light client. The IBM MQ application might have hardcoded parameters, or it might take in arguments and be configured to the designed topic and attributes.

No extra configuration is required for the publish/subscribe application. It is “business as usual.” To send messages from IBM MQ Light to IBM MQ:

- In the **IBM MQ side**, which is shown to the right of the diagram:
 - Create a subscription descriptor.
 - Determine the subscription options.
 - Determine whether the topic is coded or parameterized to subscribe to topic “Sport/data”.
 - Use the MQSUB call to create the subscription.
 - Use the MQGET to receive messages.
- In the **IBM MQ Light side**, which is shown to the left of the diagram:
 - Set the `qos`, `ttl`, and other properties per the requirements of the application.
 - Create the IBM MQ Light client by using the `createClient` method (omitted from diagram).

- Use the `client.send` method to publish the messages to topic “Sport/data”.

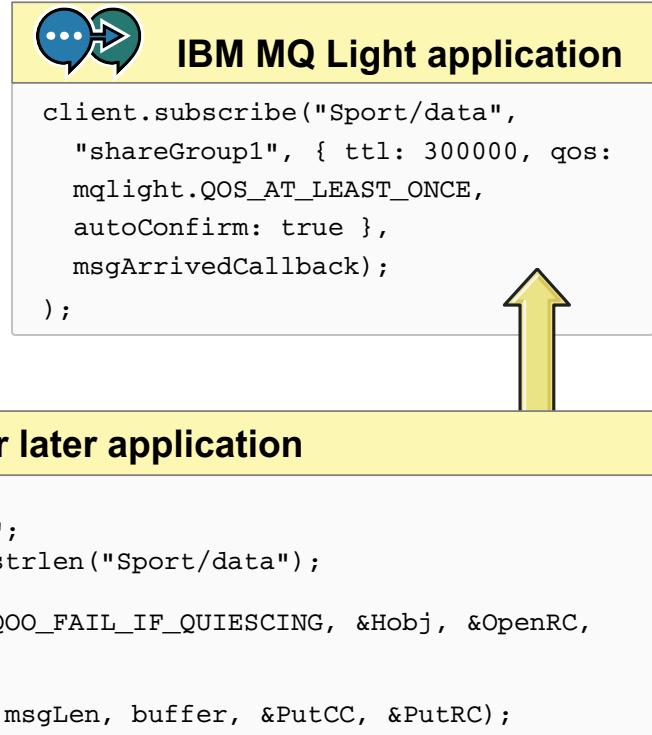


Note

If you do not want to use the MQI for subscribing, you can use an IBM MQ JMS subscriber instead.

Send messages from an IBM MQ to IBM MQ Light

- IBM MQ Light client must first subscribe by using the `client.subscribe()` API call
- IBM MQ application
 - Creates TOPIC object descriptor
 - Uses MQOPEN to open topic
 - Uses MQPUT to publish the message



Advanced Message Queuing Protocol (AMQP) and IBM MQ Light

© Copyright IBM Corporation 2017

Figure 11-18. Send messages from an IBM MQ to IBM MQ Light

To send messages from IBM MQ to IBM MQ Light, you follow a similar process.

- From IBM MQ Light, shown on the upper right of the diagram:
 - Create the IBM MQ Light client by using the `createClient` method (omitted from diagram).
 - Determine the correct `ttl`, `qos`, or other properties as required by the application.
 - Use the `client.subscribe` method to subscribe to “Sport/data”.
- From the IBM MQ side:
 - Set your IBM MQ object descriptor and length to topic “Sport/data” and corresponding length.
 - Use the `MQOPEN` function to open the topic that is specified in the object descriptor.
 - Use the `MQPUT` function to publish to the specified topic.

In the cases that are reviewed, you are working with topics. No queues are involved.

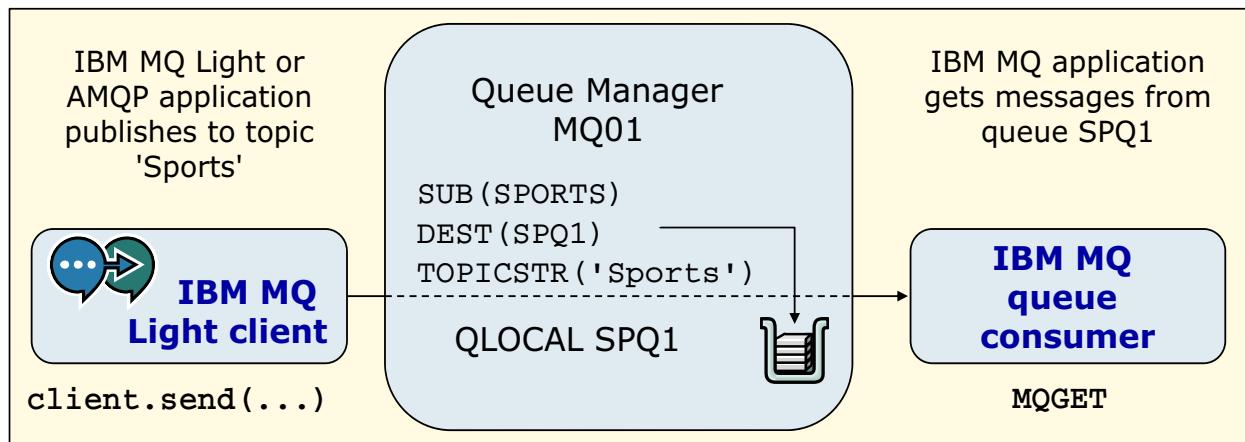
However, existing applications might be coded to put and messages from queues, not to publish or subscribe to topics. The next two slides look at the case of applications that put and get messages from queue.

**Note**

If you do not want to use the MQI for publishing, you can use an IBM MQ JMS subscriber instead. Some developers might not use the MQI, but might be familiar with JMS.

From IBM MQ Light to IBM MQ queue-based application

- This use case is for applications that are written to get messages from a queue instead of subscribing to a topic
- Define an IBM MQ SUBSCRIPTION object where:
 - The topic string matches the topic that the IBM MQ Light or other AMQP application publishes to
 - The DEST queue in matches the queue that the IBM MQ application gets messages from



Advanced Message Queuing Protocol (AMQP) and IBM MQ Light

© Copyright IBM Corporation 2017

Figure 11-19. From IBM MQ Light to IBM MQ queue-based application

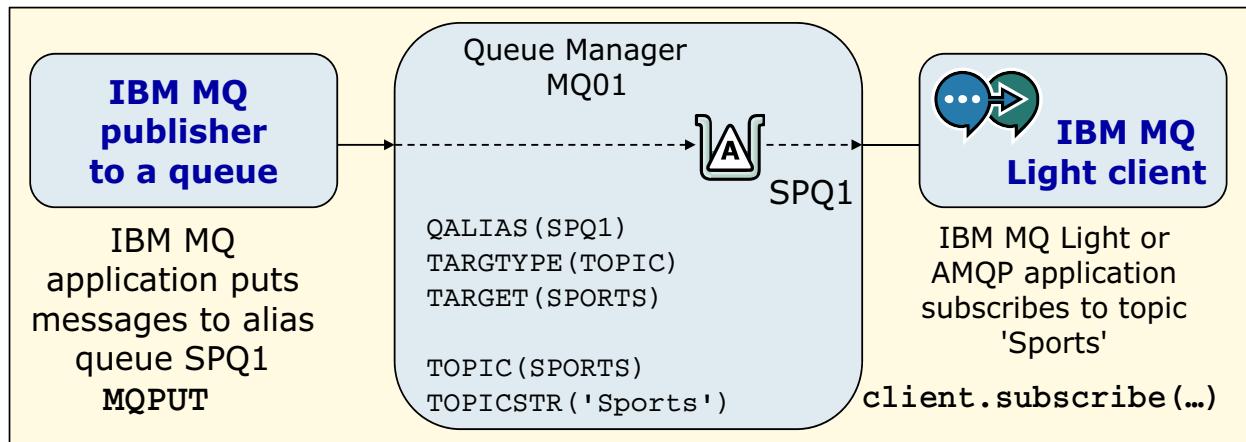
This case assumes that AMQP support is enabled and configured in the queue manager.

The IBM MQ “queue consumer” on the right is an existing application that gets messages from queues. To send messages from IBM MQ Light to an existing “queued” application:

- Ensure that the service attribute of the IBM MQ Light client in the `createClient` method points to the queue manager that needs to get the messages.
- Confirm the topic string that the IBM MQ Light client is sending messages to.
- In the queue manager, describe a subscription object by using `DEFINE SUB`.
 - Use the correct topic string in the `TOPICSTR` attribute of the subscription object.
 - Use the name of the queue that the application gets messages from in the `DEST` attribute of the subscription object.
- Continue to use the IBM MQ application that gets messages from the queue that is indicated in the subscription object, unchanged.

From an IBM MQ queue-based application to IBM MQ Light

- This use case is for applications that are written to put messages to a queue instead of sending messages to a topic
- Define an IBM MQ TOPIC object where the topic string matches the topic that the IBM MQ Light or other AMQP application publishes to
- Define a QALIAS where the target type is a topic, and the target topic matches the name of the corresponding IBM MQ topic object



Advanced Message Queuing Protocol (AMQP) and IBM MQ Light

© Copyright IBM Corporation 2017

Figure 11-20. From an IBM MQ queue-based application to IBM MQ Light

For this case, the IBM MQ application that puts the messages to a queue is on the left of the diagram. The IBM MQ Light client subscriber that needs to receive the messages is on the right of the diagram. This case also assumes that AMQP support is enabled and configured in the queue manager.

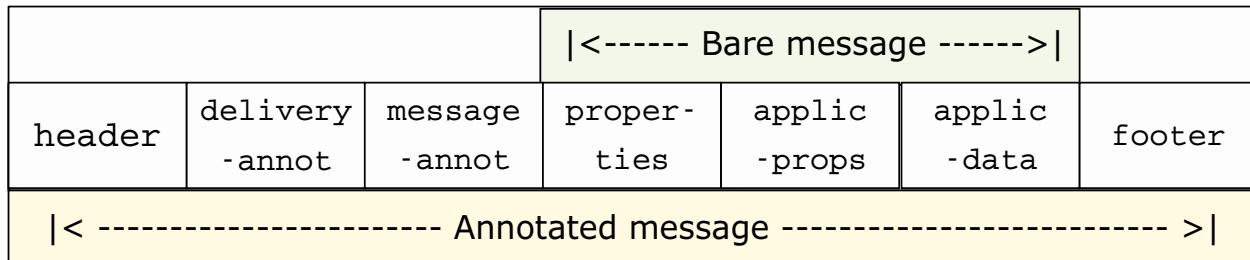
To send messages now put to a queue by an application, in the IBM MQ Light side:

- Use the application that subscribes to the correct string.

In the IBM MQ side:

- Define a TOPIC object with a TOPICSTR property that matches the string that the IBM MQ Light application subscribes to.
- Define a queue alias with the name of the queue that the application puts messages to.
 - For the target type, or TARGTYPE attribute of the QALIAS, use the string TOPIC to indicate that this QALIAS object points to a topic, not a queue.
 - For the TOPIC attribute of the QALIAS object, include the name of the topic that was defined earlier.
 - Continue to use the IBM MQ application that puts messages to the alias queue. The messages now go to the IBM MQ Light subscriber.

AMQP message format and terminology



Source: <http://www.amqp.org>

- Bare message
 - Message as supplied by the sender
 - Unchangeable by the message broker
- Annotated message is the message as seen by the receiver
- Two types of annotations
 - Annotations that travel with the message
 - Annotations that are used by the next node

Advanced Message Queuing Protocol (AMQP) and IBM MQ Light

© Copyright IBM Corporation 2017

Figure 11-21. AMQP message format and terminology

AMQP messages are composed of the following parts:

- A header
- Delivery annotations
- Message annotations
- Properties: Part of bare message
- Application properties: Part of bare message
- Body: Part of bare message
- A footer

AMQP has a concept of the “bare message”, which is the message as supplied by the sender. In the three boxes encompassed by the bare message label in the diagram, you can see that the bare message is composed of message properties (properties), application properties, and the data or payload.

The “annotated” message is what the message receivers “see”, or can view when they access the message. Some of the annotations travel with the message, others are used in the next node.

Next, you see how the AMQP fields map to the IBM MQ fields.

Mapping IBM MQ to AMQP message (1 of 2)

- **header**

Field	Value	Default setting
durable	True if MQMD.Persistence is set to MQPER_PERSISTENT. False if not set.	false
priority	From mq_amqp.Hdr.Pri, if set, or otherwise from MQMD.Priority, if set. If neither set, set to 4.	4
ttl	MQMD.Expiry in milliseconds. If the value of MQMD.Expiry is MQEI_UNLIMITED, then set to the maximum value for the AMQP ttl field.	n/a
first-acquirer	From mq_amqp.Hdr.Fac, if set, or false otherwise.	false
delivery-count	From mq_amqp.Hdr.Dct, if set, or 0 otherwise.	0

- **delivery-annotation** is set as necessary by AMQP
- **message-annotation** is not included

IBM MQ RFH2 header has an AMQP folder called **mq_amqp**



Important

The mappings that are shown in these slides were taken directly from the IBM Knowledge Center documentation at the time this course was written. While most of this information is not expected to change, always refer to the IBM Knowledge Center for the most up-to-date mappings.

Two mapping cases exist:

- IBM MQ to AMQP, or outgoing messages
- AMQP to IBM MQ, or incoming messages

The IBM MQ to AMQP, or “outgoing” mappings are described in the first set of mapping slides.

IBM MQ examines the message and determines how to build the AMQP message. IBM MQ takes care of the mappings as detailed in the IBM MQ to AMQP mapping sections of the IBM Knowledge Center. The mappings are summarized in these notes.

The IBM MQ RFH2 header now has an AMQP folder called **mq_amqp**. The word AMQP precedes each field in bold to stress that the information is going to **AMQP** (from IBM MQ).

- **AMQP header:**

- The header is included *if and only if* the corresponding value in the IBM MQ message is different from the initial AMQP default value for the property.
- If included, the fields are mapped according to the table that is included in the slide.
- The IBM MQ message might have properties, which if they exist, are set as AMQP headers:
 - IBM MQ message property AMQPFirstAcquirer is set as AMQP message header.first-acquirer.
 - IBM MQ message property AMQPDeliveryCount is set as AMQP message header.delivery-count.
- **AMQP delivery annotation:** Is set as required by the AMQP channel.
- **AMQP message annotation:** Is not included.

IBM MQ to AMQP mappings are continued in the next slide.

Mapping IBM MQ to AMQP message (2 of 2)

- **properties:** If not set by equivalent mq_amqp.Prp, properties generated as shown

Field	Value
message-id	The MQMD.MsgId is set as binary.
user-id	The UTF-8 form of the MQMD.UserId is set as binary in network byte-order.
to	The queue that the message was from which the message was received, or, for a publication, the topic string.
subject	Not set.
reply-to	The MQMD.ReplyToQ if non-blank, otherwise not set.
correlation-id	The MQMD.CorrelId is set as binary if non-blank, otherwise not set.
content-type	Not set.
content-encoding	Not set.
absolute-expiry-time	Not set.
creation-time	The MQMD.PutDate and MQMD.PutTime fields are used to generate a time stamp.
group-id	Not set.
group-sequence	Not set.
reply-to-group-id	Not set.

Advanced Message Queuing Protocol (AMQP) and IBM MQ Light

© Copyright IBM Corporation 2017

Figure 11-23. Mapping IBM MQ to AMQP message (2 of 2)

IBM MQ sets the following AMQP properties by using the listed rules:

- **AMQP properties:**
 - If the MQMD PutApplType is MQAT_AMQP, all the mq_amqp.Prp properties as they appear are copied into the corresponding AMQP message fields.
 - If the MQMD PutApplType is not MQAT_AMQP, any mq_amqp.Prp properties that happen to exist in the IBM MQ message are ignored. The properties are generated based on the rules in the table that is displayed in this slide.
- **AMQP application properties:** Any message properties in the RFH2 usr space are copied to the respective AMQP application properties.
- **AMQP body:** The AMQP channel converts the IBM MQ payload into UTF-8.
- **AMQP footer:** The footer is not included.



Attention

The mapping values shown were taken from the IBM Knowledge Center at the time that this course was written. Always consult IBM Knowledge Center for updates.

Mapping AMQP to IBM MQ message (1 of 6)

• message descriptor 1 of 2

Field	Value
StruclId	MQMD_STRUC_ID
Version	MQMD_VERSION_1
Report	MQRO_NONE
MsgType	MQMT_DATAGRAM
Expiry	Value that is taken from the <code>ttl</code> field in the AMQP message header.
Feedback	MQFB_NONE
Encoding	MQENC_NORMAL
CodedCharSetId	1208 (UTF-8)
Format	See Payload.
Priority	Value that is taken from the priority field in the AMQP message header. If set, limited to a maximum of 9. If not set, takes the default value of 4.
Persistence	If the durable field in the AMQP message header is set to true, set to MQPER_PERSISTENT. Otherwise, set to MQPER_NOT_PERSISTENT.
MagId	The queue manager allocates a unique 24-byte MsgId.
CorrelId	Value that is taken from the correlation-id field in the AMQP properties, if set. Set to a 24-byte binary value. Otherwise, set to MQCI_NONE.

Advanced Message Queuing Protocol (AMQP) and IBM MQ Light

© Copyright IBM Corporation 2017

Figure 11-24. Mapping AMQP to IBM MQ message (1 of 6)

In summary, when mapping from AMQP to IBM MQ:

- IBM MQ includes an RFH2 header on all incoming messages.
- Some AMQP header fields are set as MQMD fields, such as:
 - AMQP header.ttl is set as MQMD.expiry (converted to 10ths of seconds)
 - AMQP header.priority is set as MQMD.priority
 - AMQP properties.correlation-id is set as MQMD.correlid
- All AMQP headers are set as IBM MQ message properties with a similar mapped name
- All AMQP properties are also set as IBM MQ message properties as noted in the table, which continues for six slides.
- All AMQP application-properties are copied to the IBM MQ RFH2 user space and use similar naming conventions.

Mapping AMQP to IBM MQ message (2 of 6)

- **message descriptor 2 of 2**

Field	Value
BackoutCount	0
ReplyToQ	" "
ReplyToQMgr	" "
UserIdentifier	Set to the identifier of the authenticated user that connected to the AMQP channel
AccountingToken	MQACT_NONE
ApplIdentityData	Hexadecimal string. Set to the last 8 bytes of the IBM MQ connection identifier of the AMQP channel.
PutApplType	MQAT_AMQP
PutApplName	Contains the name of the application that put the message.
PutDate	Value that is taken from the creation-time field of the AMQP properties, if set. Otherwise, set to the current date.
PutTime	Value that is taken from the creation-time field of the AMQP properties, if set. Otherwise, set to the current time.
ApplOriginData	" "

Advanced Message Queuing Protocol (AMQP) and IBM MQ Light

© Copyright IBM Corporation 2017

Figure 11-25. Mapping AMQP to IBM MQ message (2 of 6)

Notice that the MQMD PutApplType is set to MQAT_AMQP, since in this case the message came from AMQP.



Attention

The mapping values shown were taken from the IBM Knowledge Center at the time that this course was written. Always consult IBM Knowledge Center for updates.

Mapping AMQP to IBM MQ message (3 of 6)

- message properties 1 of 4

Property	RFH2	Type	Description
AMQPLListener	mq_amqp.Lis	MQTYPE_STRING	An identifying string for the AMQP channel. It is used to generate the message, so that interested parties can tell which version put the message (for example, the service team when diagnosing problems). The queue manager does not validate the value, and the value must not be documented externally.
AMQPVersion	mq_amqp.Ver	MQTYPE_STRING	The version of the AMQP message. If not present, "1.0" is assumed. The queue manager does not validate the value.
AMQPClient	mq_amqp.Cli	MQTYPE_STRING	An identifying string for the API. It is used to send the AMQP message to the channel, so that interested parties can tell which version put the message (for example, the service team when diagnosing problems). The queue manager does not validate the value, and the value must not be documented externally.
AMQPDurable	mq_amqp.Hdr.Dur	MQTYPE_BOOLEAN	The value of the durable field in the AMQP message header, if set.

Advanced Message Queuing Protocol (AMQP) and IBM MQ Light

© Copyright IBM Corporation 2017

Figure 11-26. Mapping AMQP to IBM MQ message (3 of 6)



Attention

The mapping values shown were taken from the IBM MQ IBM Knowledge Center at the time that this course was written. Always consult IBM Knowledge Center for updates.

Mapping AMQP to IBM MQ message (4 of 6)

- message properties 2 of 4

Property	RFH2	Type	Description
AMQPPriority	mq_amqp.Hdr.Pri	MQTYPE_INT32	The value of the priority field in the AMQP message header, if set.
AMQPTtl	mq_amqp.Hdr.Ttl	MQTYPE_INT64	The value of the ttl field in the AMQP message header, if set.
AMQPFirstAcquirer	mq_amqp.Hdr.Fac	MQTYPE_BOOLEAN	The value of the first-acquirer field in the AMQP message header, if set.
AMQPDeliveryCount	mq_amqp.Hdr.Dct	MQTYPE_INT64	The value of the delivery-count field in the AMQP message header, if set.
AMQPMsgId	mq_amqp.Prp.Mid	MQTYPE_STRING	The value of the message-id field in the AMQP properties, if set as a string.
		MQTYPE_BYTE_STRING	The value of the message-id field in the AMQP properties, if set as a byte string.
AMQPUserId	mq_amqp.Prp.Uid	MQTYPE_BYTE_STRING	The value of the user-id field in the AMQP properties, if set.

Advanced Message Queuing Protocol (AMQP) and IBM MQ Light

© Copyright IBM Corporation 2017

Figure 11-27. Mapping AMQP to IBM MQ message (4 of 6)



Attention

The mapping values shown were taken from the IBM Knowledge Center at the time that this course was written. Always consult IBM Knowledge Center for updates.

Mapping AMQP to IBM MQ message (5 of 6)

- message properties 3 of 4

Property	RFH2	Type	Description
AMQPTo	mq_amqp.Prp.To	MQTYPE_STRING	The value of the to field in the AMQP properties, if set.
AMQPSubject	mq_amqp.Prp.Sub	MQTYPE_STRING	The value of the subject field in the AMQP properties, if set.
AMQPReplyTo	mq_amqp.Prp.Rto	MQTYPE_STRING	The value of the reply-to field in the AMQP properties, if set.
AMQPCorrelationId	mq_amqp.Prp.Cid	MQTYPE_STRING	The value of the correlation-id field in the AMQP properties, if set as a string.
		MQTYPE_BYTETR STRING	The value of the correlation-id field in the AMQP properties, if set as a byte string.
AMQPContentType	mq_amqp.Prp.Cnt	MQTYPE_STRING	The value of the content-type field in the AMQP properties, if set.
AMQPContentEncoding	mq_amqp.Prp.Cne	MQTYPE_STRING	The value of the content-encoding field in the AMQP properties, if set.

Advanced Message Queuing Protocol (AMQP) and IBM MQ Light

© Copyright IBM Corporation 2017

Figure 11-28. Mapping AMQP to IBM MQ message (5 of 6)



Attention

The mapping values shown were taken from the IBM Knowledge Center at the time that this course was written. Always consult IBM Knowledge Center for updates.

Mapping AMQP to IBM MQ message (6 of 6)

- **message properties 3 of 4**

Property	RFH2	Type	Description
AMQP.AbsoluteExpiryTime	mq_amqp.Prp.Aet	MQTYPE_STRING	The value of the absolute-expiry-time field in the AMQP properties, if set.
AMQP.CreationTime	mq_amqp.Prp.Crt	MQTYPE_STRING	The value of the creation-time field in the AMQP properties, if set.
AMQP.GroupId	mq_amqp.Prp.Gid	MQTYPE_STRING	The value of the group-id field in the AMQP properties, if set.
AMQP.GroupSequence	mq_amqp.Prp.Gsq	MQTYPE_INT64	The value of the group-sequence field in the AMQP properties, if set.
AMQP.ReplyToGroupId	mq_amqp.Prp.Rtg	MQTYPE_STRING	The value of the reply-to-group-id field in the AMQP properties, if set.

- **payload/message data**

Message format	Format
AMQP body with single binary section, binary data (less the AMQP bits) is put.	MQFMT_NONE
AMQP body with a single string data section, string data (fewer AMQP bits) is put.	MQFMT_STRING
If not put with MQFMT_NONE or MQFMT_STRING, payload is formed as is.	MQFMT_AMQP

Advanced Message Queuing Protocol (AMQP) and IBM MQ Light

© Copyright IBM Corporation 2017

Figure 11-29. Mapping AMQP to IBM MQ message (6 of 6)

The IBM MQ message format is set according to the format of the incoming message as described in the payload table in this slide.



Attention

The mapping values shown were taken from the IBM Knowledge Center at the time that this course was written. Always consult IBM Knowledge Center for updates.

Enabling the IBM MQ AMQP environment

- Install the IBM MQ version that supports AMQP
- Create or use an existing queue manager
- Start the queue manager
 - Use command level parameter if required

```
The IBM MQ command level is higher than 801, no command level parameter required.
```

```
C:\>strmqm MQ05
```

```
C:\>strmqm -e CMDLEVEL=801 MQ05
```

```
IBM MQ queue manager 'MQ05' starting.  
The queue manager is associated with installation 'IBMMQV8'.  
5 log records accessed on queue manager 'MQ05' during the log replay phase.  
Log replay for queue manager 'MQ05' complete.  
Transaction manager state recovered for queue manager 'MQ05'.  
Migrating objects for queue manager 'MQ05'.  
Default objects statistics : 3 created. 0 replaced. 0 failed.
```

New functions up to command level 801 enabled. 

Figure 11-30. Enabling the IBM MQ AMQP environment

To enable support for IBM MQ Light, which includes AMQP channels in IBM MQ, you need to:

- Install the IBM MQ V8.0.0.4 manufacturing refresh. IBM MQ Light support is not made available by applying fix pack 4 to IBM MQ V8, so you must install the V8.0.0.4 package.
- Create or select an existing test queue manager. If an existing queue manager is selected, stop the queue manager.
- Use the `strmqm` command with the `-e` attribute to convert the queue manager to command level 801 (*or higher*) by typing: `strmqm -e CMDLEVEL=801 <queue manager name>`
 - When you type `strmqm -e` you convert the queue manager command level, but do not leave the queue manager in running status.
 - Check that you see message “New functions up to command level 801 enabled” as the last output of the `strmqm -e` command.
 - After you see the message that the command level 801 was enabled, start the queue manager as usual by typing: `strmqm <queue manager name>`

Confirm the AMQP functionality in the queue manager

```
C:\>runmqsc MQ05
5724-H72 (C) Copyright IBM Corp. 1994, 2015.
Starting MQSC for queue manager MQ05.
dis chl(*) chltype(AMQP)
    1 : dis chl(*) chltype(AMQP)
AMQ8414: Display Channel details.
    CHANNEL(SYSTEM.DEF.AMQP)          CHLTYPE(AMQP)
dis service(*)
    2 : dis service(*)
AMQ8629: Display service information details.
    SERVICE(SYSTEM.AMQP.SERVICE)
AMQ8629: ...
dis qmgr amqpcap
    3 : dis qmgr amqpcap
AMQ8408: Display Queue Manager details.
    QMNAME(MQ05)                      AMQPCAP(YES)
```

Advanced Message Queuing Protocol (AMQP) and IBM MQ Light

© Copyright IBM Corporation 2017

Figure 11-31. Confirm the AMQP functionality in the queue manager

You can run some checks to ensure that the environment was enabled for IBM MQ Light and AMQP clients. To run these checks, open an IBM MQ command window by typing `runmqsc <queue manager name>`. After `runmqsc` starts, run the checks that are shown in these notes:

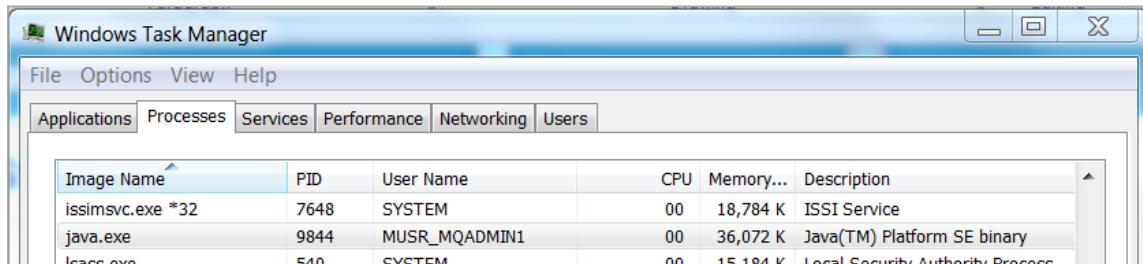
- Display that the default AMQP channel definition object, `SYSTEM.DEF.AMQP`, exists by typing:
`DIS CHL(*) chltype(AMQP)`
The system AMQP channel should display.
- Check that the AMQP service, `SYSTEM.AMQP.SERVICE`, exists by typing: `DIS SERVICE(*)`
You might see several services; ensure that `SYSTEM.AMQP.SERVICE` is listed.
- Ensure that the AMQP capability, `AMQPCAP`, is enabled in the queue manager by typing: `DIS QMGR AMQPCAP`
It should display: `AMQPCAP(YES)`

Start and check the AMQP service

- Start the AMQP service

```
start service(SYSTEM.AMQP.SERVICE)
  1 : start service(SYSTEM.AMQP.SERVICE)
AMQ8733: Request to start Service accepted.
```

- Check that a JVM started
 - For Linux, type `ps -ef | grep java`
 - For Windows, check Task Manager



- Display service status by using the DIS SVSTATUS MQSC command

Advanced Message Queuing Protocol (AMQP) and IBM MQ Light

© Copyright IBM Corporation 2017

Figure 11-32. Start and check the AMQP service

After you check that the AMQP functions are enabled, start the AMQP service. From a `rwmqsc` command line, type `start service(SYSTEM.AMQP.SERVICE)` and check the results. You can check in two ways:

- The first method is to look for a running Java process. The process is different for Windows or Linux:
 - For Linux, type `ps -ef | grep java` and check for the process.
 - For Windows, check the Windows Task Manager.
- The second method is to use the `DIS SVSTATUS` command from the `rwmqsc` command line. Proceed to the next slide for more on `DIS SVSTATUS`.

DIS SVSTATUS(SYSTEM.AMQP.SERVICE)

```
dis svstatus(SYSTEM.AMQP*) all
    3 : dis svstatus(SYSTEM.AMQP*) all
AMQ8632: Display service status details.

    SERVICE(SYSTEM.AMQP.SERVICE)           STATUS(RUNNING)
    PID(9880)                            SERVTYPE(SERVER)
    STARTDA(2015-12-08)                  STARTTI(08.27.36)
    CONTROL(MANUAL)
    STARTCMD(C:\Program Files\IBM\IBM MQ\bin\amqp.bat)
    STARTARG(start -m MQ05 -d "C:\IBM\IBM MQ\qmgrs\MQ05\\." -g
"C:\IBM\IBM MQ\\.")
    STOPCMD(C:\Program Files\IBM\IBM MQ\bin\amqp.bat)
    STOPARG(stop -m MQ05 -d "C:\IBM\IBM MQ\qmgrs\MQ05\\." -g
"C:\IBM\IBM MQ\\.")
    DESCRIPTOR(Manages clients that use the AMQP protocol)
    STDOUT(C:\IBM\IBM MQ\qmgrs\MQ05\amqp.stdout)
    STDERR(C:\IBM\IBM MQ\qmgrs\MQ05\amqp.stderr)
```

[Advanced Message Queuing Protocol \(AMQP\) and IBM MQ Light](#)

© Copyright IBM Corporation 2017

Figure 11-33. DIS SVSTATUS(SYSTEM.AMQP.SERVICE)

The DIS SVSTATUS(SYSTEM.AMQP.SERVICE) command indicates the status of the AMQP service. Along with the status, which is RUNNING in the display, the command also shows other details about the service, such as its process ID (PID), and the date and time the service was started. You can also see the start arguments, such as the location of some of the output log files amqp.stdout and amqp.stderr.

Configure the AMQP channel

- CHLTYPE is AMQP
- The MCAUSER for the channel should be set to “nobody”, and the channel should be authorized by using a CHLAUTH rule
- If port is not specified, initial default port is 5672

```
define channel (MQ05AMQP) chltype(AMQP) port(5673) mcauser('nobody')
        4 : define channel (MQ05AMQP) chltype(AMQP) port(5673)
              mcauser('nobody')
AMQ8014: IBM MQ channel created.
start ch1 (MQ05AMQP)
      5 : start ch1 (MQ05AMQP)
AMQ8018: Start IBM MQ channel accepted.
```

Figure 11-34. Configure the AMQP channel

Before you create or use a channel, you must observe some basic IBM MQ security considerations.

- Ideally, the ID used in the MCAUSER should have no privileges.
- The channel can then be authorized with a channel authentication (CHLAUTH) rule.
- IBM MQ objects and users should be authorized by using `setmqaut`.
- The channel authentication attribute in the queue manager also needs to be addressed.

Before you proceed, look at how to resolve some of the problems that can surface after you start the channel and try to connect with the client. These items should be addressed before defining a new channel to prevent time that might be lost to resolve connection problems.



Troubleshooting

Connection authentication can cause connection problems. If not properly addressed, these connection problems might be difficult to resolve. In this course, connection authentication is set to OPTIONAL. In your organization, connection authentication must be set according to your security requirements.

If you set the correct `setmqaut` permissions and still have connection problems, check the following areas:

- Assuming that the queue manager is new or is set up to use the system IDs (not LDAP), ensure that the `SYSTEM.DEFAULT.AUTHINFO.IDPWOS` record has the `CHCKCLNT` attribute set to OPTIONAL.

`AMQ8566: Display authentication information details.`

`AUTHINFO(SYSTEM.DEFAULT.AUTHINFO.IDPWOS)`

`AUTHTYPE(IDPWOS)`

`ADOPTCTX(NO)`

`DESCR()`

`CHCKCLNT(OPTIONAL)`

`CHCKLOCL(OPTIONAL)`

`FAILDELAY(1)`

Note: The `CHCKLOCL` attribute does *not* apply to AMQP. It is `CHCKCLNT` that applies to AMQP.

- If you needed to change `SYSTEM.DEFAULT.AUTHINFO.IDPWOS`, you must refresh security for the changes to take effect, or the connection problems continue.
- If a client fails in connecting to a channel, you must also stop and restart the channel before changes take effect, or connection failures continue.
- If you have error messages that state the connection or channel is “blocked”, you might have a channel authentication (CHLAUTH) or connection authentication (CONNAUTH) problem. Depending on the security requirements of your organization, if you see “blocked channel” messages, you either:
 - Type channel authentication rules to get around the failure
 - If allowed, disable channel authentication rules by typing `ALTER QMGR CHLAUTH(DISABLED)` in a `runmqsc` command line.

If you receive a security error when your IBM MQ Light client attempts to connect to the queue manager, check the queue manager logs `AMQERRxx.log` for more detailed information about the cause of the error.

You now continue with the AMQP channel setup. Assume that the required security configuration is correctly set.

Next, still in the `runmqsc` command prompt, you define an AMQP channel, or use the `SYSTEM.DEF.AMQP` channel. In the display, you see how the AMQP channel type is defined.

- In the display, the channel name is `MQ05AMQP`.
- The channel type is `AMQP`.
- If you do not specify a port, it defaults to 5672. In the display channel `MQ05AMQP` is at port 5673.

After the definition is successful, you start the AMQP channel by typing the command: `start chl(MQ05AMQP)`

In the next slide, you check the results of starting the channel.

Display the AMQP channel and channel status

```

dis chl(MQ05AMQP) chltype(AMQP)
    6 : dis chl(MQ05AMQP) chltype(AMQP)
AMQ8414: Display Channel details.

    CHANNEL(MQ05AMQP)                      CHLTYPE(AMQP)
    ALTDATE(2015-12-03)                     ALTTIME(18.13.19)
    CERTLBL( )                            DESCRL( )
    AMQPKA(AUTO)                          LOCLADDR( )
    MAXINST(999999999)                    MAXMSGL(4194304)
    MCAUSER(nomad789)                     PORT(5673)
    SSLCAUTH(REQUIRED)                  SSLCIPH( )
    SSLPEER( )                           TPROOT(SYSTEM.BASE.TOPIC)
    USECLTID(NO)

dis chs(MQ05AMQP) chltype(AMQP)
    7 : dis chs(MQ05AMQP) chltype(AMQP)
AMQ8417: Display Channel Status details.

    CHANNEL(MQ05AMQP)                      CHLTYPE(AMQP)
    CONNECTIONS(0)                         STATUS(RUNNING)

```

Advanced Message Queuing Protocol (AMQP) and IBM MQ Light

© Copyright IBM Corporation 2017

Figure 11-35. Display the AMQP channel and channel status

You continue the work in the `runmqsc` command prompt.

In addition to the command shown in the display, you might also type the message that is shown to have the `chstatus` command return a list of connected clients:

```
DIS CHSTATUS(*) CHLTYPE(AMQP) CLIENTID(*)
```

The first command that is displayed shows the channel definition. Notice the following parameters:

- The channel type is AMQP.
- The port is specified as 5673; it is not using the default port.

The second command in the slide displays the channel status. One difference when you display the channel status for an AMQP channel type is that the CHLTYPE attribute must be included. If the CHLTYPE attribute is omitted, “no channel status” is reported. In the display, the status shows as RUNNING

You might reach the point where your channel is running, and the channel might start without problems. *If security is not correct in the queue manager*, it is when you try to connect to the channel with a client application that you see connection failures. So while the channel in RUNNING status is good, a running channel is not an indication that security is correctly set.

Test with the IBM MQ Light sample applications

- Successful result of IBM MQ Light sample receiver application connection to AMQP channel

```
C:\MQLight\node_modules\mqlight\samples>
node recv.js -s amqp://localhost:5673
Connected to amqp://localhost:5673 using client-id recv_da338ed
Subscribed to pattern: public
```



- IBM MQ Light sample sends connection to AMQP channel

```
... ...
samples> node send.js -s amqp://localhost:5673
Connected to amqp://localhost:5673 using client-id send_fe01236
Sending to: public
Hello World!
```



```
... ...
Connected to amqp://localhost:5673 using client-id recv_da338ed
Subscribed to pattern: public
Hello World! 
```



Figure 11-36. Test with the IBM MQ Light sample applications

After the channel starts, you test with the IBM MQ Light client applications. To run the applications that are shown in the display:

- The AMQP service and channel are started.
- node.js and the IBM MQ Light clients are installed in your system. You might also choose the Java or Ruby versions, but for this course use node.js samples.

This test uses all IBM MQ Light clients. Later you use IBM MQ to exchange messages with IBM MQ Light. To start testing, exit `rwmqsc` and open a command prompt or terminal window:

- First box of screen display:** Use the `recv.js` sample application to connect to the channel. `recv.js` takes an `-s` attribute to change the service option to use port 5673. Since the AMQP channel uses port 5673, the service attribute in the client must be changed to use the correct port. If the `-s` attribute is not passed, the client uses port 5672.
 - If the connection was successful, the node `recv.js` sample returns the client-id for the connection.
 - If no topic was specified, the sample subscribes to the “public” default topic.
 - The `recv.js` client continues to run until the process is explicitly stopped.

- **Second box of screen display:** Open a separate command prompt or terminal window, and start a “publisher” or sender application.
 - You can use the send.js application to send a message to the subscriber.
 - The same attributes to indicate that the specific port must be provided to the node send.js sample.
 - If no topic or data is specified, the send.js sample publishes the message Hello World to the “public” default topic.
- **Third box of screen display:** Shows that the recv.js application obtained the message that the send.js application published. In this case, the message is the default application text, Hello World!

Until now, you tested with IBM MQ Light clients. In the next few slides, you see how to test to and from IBM MQ and IBM MQ Light.

Subscribe and publish across IBM MQ and IBM MQ Light (1 of 2)

```
C:\MQLight\node_modules\mqlight\samples>node
  recv.js -s amqp://localhost:5673
  -t 'Sports/soccer'
```



Connected to amqp://localhost:5673 using
client-id recv_575d648

Subscribed to pattern: 'Sports/soccer'

- Applications are subscribed to topic 'Sports/soccer'

```
C:\MQLight\node_modules\mqlight\samples>amqssub 'Sports/soccer' MQ05
Sample AMQSSUBA start
Calling MQGET : 30 seconds wait time
```

Figure 11-37. Subscribe and publish across IBM MQ and IBM MQ Light (1 of 2)

To continue testing, you now use both IBM MQ and IBM MQ Light samples to subscribe to a topic 'Sports/soccer'.

- The top box shows the IBM MQ Light subscription. The sample `recv.js` takes several parameters. In this example, both the host name and port number are passed to the service property by using the `-s` option. The topic to subscribe is passed by using the `-t` option.
- The bottom box shows a similar subscription, but this time it uses IBM MQ. The `amqssub` sample program is invoked to subscribe to topic 'Sports/soccer' in queue manager MQ05. The `amqssub` sample subscription looks for published messages for 30 seconds.

You now look at the publishers.

Subscribe and publish across IBM MQ and IBM MQ Light (2 of 2)

An IBM MQ Light and an IBM MQ application publish to topic 'Sports/soccer'

```
C:\MQLight\node_modules\mqlight\samples>
node send.js -s amqp://localhost:5673 -t
'Sports/soccer' Southampton4 Arsenal0
Connected to amqp://localhost:5673 using
client-id send_4ea6e8c
Sending to: 'Sports/soccer'
Southampton4
Arsenal0
-----
amqspub 'Sports/soccer' MQ05
Sample AMQSPUBA start
publish options are 0
target topic is 'Sports/soccer'
Orlando3 Miami1
Sample AMQSPUBA end
```



[Advanced Message Queuing Protocol \(AMQP\) and IBM MQ Light](#)

© Copyright IBM Corporation 2017

Figure 11-38. Subscribe and publish across IBM MQ and IBM MQ Light(2 of 2)

In this display, you use IBM MQ Light and IBM MQ to publish to topic 'Sports/soccer'.

- The IBM MQ Light application is publishing the results of a football match between Southampton and Arsenal. The send.js sample is passing:
 - The host and port number
 - The subscription string
 - The message payload or data
- The IBM MQ application amqspub is publishing to topic "Sports/soccer" in queue manager MQ05. amqspub is providing the results of a match between the Orlando and Miami teams.

You see the results next.

Both IBM MQ and IBM MQ Light applications get messages

```
C:\MQLight\node_modules\mqlight\samples>node
  e recv.js -s amqp://localhost:5673
  -t 'Sports/soccer'

Connected to amqp://localhost:5673 using
  client-id recv_575d648

Subscribed to pattern: 'Sports/soccer'

Southampton4
Arsenal10
Orlando3 Miami1
```



```
amqssub 'Sports/soccer' MQ05
```

Sample AMQSSUBA start

Calling MQGET : 30 seconds wait time

message <Southampton4>

Calling MQGET : 30 seconds wait time

message <Arsenal10>

Calling MQGET : 30 seconds wait time

message <Orlando3 Miami1>

Calling MQGET : 30 seconds wait time

no more messages Sample AMQSSUBA end

- Publications received by both subscriptions from IBM MQ Light and IBM MQ

Advanced Message Queuing Protocol (AMQP) and IBM MQ Light

© Copyright IBM Corporation 2017

Figure 11-39. Both IBM MQ and IBM MQ Light applications get messages

Two applications published match results; the IBM MQ Light application published the scores between Arsenal and Southampton, and the IBM MQ application published the scores between Orlando and Miami.

The IBM MQ Light application received both publications, and the IBM MQ application received both publications.

You can use similar samples to start the exchange of messages between IBM MQ applications and IBM MQ Light applications.

Display topic status for the queue manager

- `dis tpstatus('#') where (subcount gt 0) all`

AMQ8754: Display topic status details.

<code>TOPICSTR('Sports/soccer')</code>	<code>ADMIN()</code>
<code>CLUSTER()</code>	
<code>COMMINFO(SYSTEM.DEFAULT.COMMINFO.MULTICAST)</code>	
<code>MDURMDL(SYSTEM.DURABLE.MODEL.QUEUE)</code>	
<code>MNDURMDL(SYSTEM.NDURABLE.MODEL.QUEUE)</code>	
<code>CLROUTE(NONE)</code>	<code>DEFPSIST(NO)</code>
<code>DEFPRTY(0)</code>	<code>DEFPRESP(SYNC)</code>
<code>DURSUB(YES)</code>	<code>PUB(ENABLED)</code>
<code>SUB(ENABLED)</code>	<code>PMSGDLV(ALLDUR)</code>
<code>NPMMSGDLV(ALLAVAIL)</code>	<code>RETAINED(NO)</code>
<code>MCAST(DISABLED)</code>	<code>PUBCOUNT(0)</code>
<code>SUBCOUNT(2)</code>	<code>PUBSCOPE(ALL)</code>
<code>SUBSCOPE(ALL)</code>	<code>USEDLQ(YES)</code>

Figure 11-40. Display topic status for the queue manager

Another command that you can type in addition to the display, is `DIS SBSTATUS`, which can be used to find out which IBM MQ queue backs the subscription that is created by an AMQP client.

AMQP channel log files

- The default data directory on Windows is `C:\ProgramData\IBM\MQ`
- The default data directory on Linux is `/var/mqm`
- The AMQP channel writes log information to the following log files, found in the IBM MQ data directory:
 - `amqp.stdout`, written to the `qmgrs/<QM-name>` folder
 - `amqp.stderr`, written to the `qmgrs/<QM-name>` folder
 - `amqp_*.log`, written to the `qmgrs/<QM-name>/errors` folder
- Any FDC files are created as `AMQP*.FDC` files, which are written to the `<data-directory>/errors` folder

Figure 11-41. AMQP channel log files

If you enable trace by using `strmqtrc`, AMQP trace files are written to `<data-directory>/trace` with the other IBM MQ trace files.

Security

- SSL/TLS
 - AMQP channel can be configured for SSL like any other IBM MQ channel type
 - The IBM MQ Light client uses the `options` attribute `sslTrustCertificate`, and `sslVerifyName` fields of the `client.create` function
- MCAUSER channel attribute
- Channel authentication rules
 - Rules to allow or block channels based on selected criteria
 - Criteria includes originating IP address, client name, and SSL information
- Connection authentication

CHCKCLNT value	Client credential action	SASL mechanism supported by IBM MQ
<ul style="list-style-type: none"> • REQUIRED or REQDADM: Connection refused • NONE or OPTIONAL: Accepted 	Not sent	ANONYMOUS
<ul style="list-style-type: none"> • REQUIRED, REQDADM, or OPTIONAL: Checked by the queue manager • NONE: Refused 	Name and password sent	PLAIN

[Advanced Message Queuing Protocol \(AMQP\) and IBM MQ Light](#)

© Copyright IBM Corporation 2017

Figure 11-42. Security

Unit summary

- Describe the Advanced Message Queuing Protocol
- Describe typical IBM MQ application and IBM MQ Light application interface scenarios
- Describe basic IBM MQ Light concepts and list the components
- Describe the quality of service categories available with IBM MQ Light
- Describe how to write code to send and receive messages between IBM MQ applications and IBM MQ Light node.js applications
- Describe how to map headers and properties between IBM MQ applications and AMQP applications
- Explain how to enable and configure IBM MQ – IBM MQ Light interface
- Describe the commands that are used to check the IBM MQ AMQP channel connections
- Explain where to locate the logs that hold IBM MQ Light-related information
- Describe the security options for IBM MQ Light

Review questions (1 of 2)

1. Which of the statements that are listed are true for IBM MQ Light?
 - a. IBM MQ Light servers use topics and destinations to route the application data
 - b. IBM MQ Light servers communicate by using the AMQP protocol
 - c. A connection between IBM MQ and IBM MQ Light can use SSL/TLS
 - d. All of the above
2. True or False: IBM MQ support for IBM MQ Light and AMQP channels was first made available with the IBM MQ V8.0.0.4 manufacturing refresh for distributed platforms.



Advanced Message Queuing Protocol (AMQP) and IBM MQ Light

© Copyright IBM Corporation 2017

Figure 11-44. Review questions (1 of 2)

Write your answers here:

1.

2.

Review questions (2 of 2)

3. Select the best answer. You specify delivery assurance for IBM MQ Light applications by taking the following action:
 - a. Ensuring that the interfacing IBM MQ application sets persistence
 - b. Setting the IBM MQ Light `create.client qos` attribute in the service option
 - c. Setting the `qos` value in either the `client.send` or `client.subscribe` method
 - d. Configuring the IBM MQ AMQP support for persistence
4. True or False: AMQP channels cannot be configured to use SSL/TLS.
5. In the `mqlight.client.send` method, what are the default and maximum TTL values?
 - a. Default is 7 days, and maximum is 30 days in milliseconds
 - b. Default is 1 day and maximum is 30 days in milliseconds
 - c. Default is 7 days and maximum is 15 days in milliseconds
 - d. Default is 7 hours and maximum is 30 days in milliseconds

Advanced Message Queuing Protocol (AMQP) and IBM MQ Light

© Copyright IBM Corporation 2017

Figure 11-45. Review questions (2 of 2)

Write your answers here:

- 3.
- 4.
- 5.

Review answers (1 of 3)

1. Which of the statements that are listed are true for IBM MQ Light?
 - a. IBM MQ Light servers use topics and destinations to route the application data
 - b. IBM MQ Light servers communicate by using the AMQP protocol
 - c. A connection between IBM MQ and IBM MQ Light can use SSL/TLS
 - d. All of the above

The answer is: d, all are true for IBM MQ Light.
2. True or False: IBM MQ support for IBM MQ Light and AMQP channels was first made available with the IBM MQ V8.0.0.4 manufacturing refresh for distributed platforms.

The answer is: True. The packaged refresh must be installed.
The application of fix pack 4 does not contain the AMQP support feature. IBM MQ V9 and later include the support.



Review answers (2 of 3)

3. Select the best answer. You specify delivery assurance for IBM MQ Light applications by taking the following action:
 - a. Ensuring that the interfacing IBM MQ application sets persistence
 - b. Setting the IBM MQ Light `create.client qos` attribute in the service option
 - c. Setting the `qos` value in either the `client.send` or `client.subscribe` method
 - d. Configuring the IBM MQ AMQP support for persistence

The answer is: c. The default value is 0, at most once. A value of 1 is at least once.
4. True or False: AMQP channels cannot be configured to use SSL/TLS. The answer is: False. SSL is configured as usual in the AMQP channel definition.



Review answers (3 of 3)

5. In the `mqlight.client.send` method, what are the default and maximum TTL values?
- a. Default is 7 days, and maximum is 30 days in milliseconds
 - b. Default is 1 day and maximum is 30 days in milliseconds
 - c. Default is 7 days and maximum is 15 days in milliseconds
 - d. Default is 7 hours and maximum is 30 days in milliseconds.

The answer is: a. For the `mqlight.client.send` method, if no value is specified the default is 7 days. The maximum value is 30 days.



Connecting IBM MQ Light applications to IBM MQ applications

Advanced Message Queuing Protocol (AMQP) and IBM MQ Light

© Copyright IBM Corporation 2017

Figure 11-49. Connecting IBM MQ Light applications to IBM MQ applications

Exercise objectives (1 of 2)

- Examine the IBM MQ Light components that are configured in the queue manager
- Start and check the status of the AMQP service
- Start and check the status of an AMQP channel
- Use an IBM MQ Light application and the sample node.js IBM MQ Light client application to subscribe to a topic of interest
- Use an IBM MQ Light application and the sample node.js IBM MQ Light client application to publish messages to interested subscribers
- Use a sample IBM MQ publish/subscribe application to publish messages of interest to subscribed IBM MQ and IBM MQ Light node.js applications
- Use a sample IBM MQ publish/subscribe application to subscribe to messages of interest that might proceed from IBM MQ applications or IBM MQ Light publish applications



Exercise objectives (2 of 2)

- Examine the IBM MQ objects that are necessary for a queue-based application to receive messages of interest from an IBM MQ Light message producer application
- Use a sample IBM MQ application to get messages from a queue that an IBM MQ Light application publishes
- Examine the IBM MQ object that is necessary for a queue-based application to put messages on a queue that can go to an interested IBM MQ Light client application
- Use a sample IBM MQ application to put messages to a queue destined to interested IBM MQ Light subscriber applications



Unit 12. Course summary

Estimated time

00:15

Overview

This unit summarizes the course and provides information for future study.

Unit objectives

- Explain how the course met its learning objectives
- Access the IBM Training website
- Identify other IBM Training courses that are related to this topic
- Locate appropriate resources for further study

[Course summary](#)

© Copyright IBM Corporation 2017

Figure 12-1. Unit objectives

Course objectives

- Describe key IBM MQ components and processes
- Explain the effect of design and development choices in the IBM MQ environment
- Describe common queue attributes and how to control these attributes in an application
- Differentiate between point-to-point and publish/subscribe messaging styles
- Describe the calls, structures, and elementary data types that compose the message queue interface
- Describe how IBM MQ determines the queue where messages are placed
- Explain how to code a program to get messages by either browsing or removing the message from the queue

[Course summary](#)

© Copyright IBM Corporation 2017

Figure 12-2. Course objectives

Course objectives

- Describe how to handle data conversion across different platforms
- Explain how to put messages that have sequencing or queue manager affinities
- Explain how to commit or back out messages in a unit of work
- Describe how to code programs that run in an IBM MQ Client
- Explain the use of asynchronous messaging calls
- Describe the basics of writing publish/subscribe applications
- Describe the Advanced Message Queuing Protocol (AMQP)
- Differentiate among the various IBM MQ Light AMQP implementations
- Explain how to use IBM MQ applications to interface with IBM MQ Light

[Course summary](#)

© Copyright IBM Corporation 2017

Figure 12-3. Course objectives

Earn an IBM Badge

- After completing this course, you might be ready to take an IBM Badge test
- Use IBM Badges to share verified proof of your IBM credentials
- Find your Badge test on either of these sites:
 - <https://www.youracclaim.com/organizations/ibm/badges>
 - <http://www.ibm.com/developerworks/middleware/services/badges>



Course summary

© Copyright IBM Corporation 2017

Figure 12-4. Earn an IBM Badge

To learn more on the subject

- IBM Training website:
www.ibm.com/training
- IBM MQ V9 IBM Knowledge Center:
https://www.ibm.com/support/knowledgecenter/SSFKSJ_9.0.0

Course summary

© Copyright IBM Corporation 2017

Figure 12-5. To learn more on the subject

Enhance your learning with IBM resources

Keep your IBM Cloud skills up-to-date

- IBM offers resources for:
 - Product information
 - Training and certification
 - Documentation
 - Support
 - Technical information



- To learn more, see the IBM Cloud Education Resource Guide:
www.ibm.biz/CloudEduResources

Course summary

© Copyright IBM Corporation 2017

Figure 12-6. Enhance your learning with IBM resources

Unit summary

- Explain how the course met its learning objectives
- Access the IBM Training website
- Identify other IBM Training courses that are related to this topic
- Locate appropriate resources for further study

[Course summary](#)

© Copyright IBM Corporation 2017

Figure 12-7. Unit summary



IBM

Course completion

You have completed this course:

IBM MQ V9 Application Development (Windows Labs)

Any questions?



[Course summary](#)

© Copyright IBM Corporation 2017

Figure 12-8. Course completion

Appendix A. List of abbreviations

AIX	Advanced IBM UNIX
AMQP	Advanced Message Queuing Protocol
APAR	authorized program analysis report
API	application programming interface
ASCII	American Standard Code for Information Interchange
ATM	automated teller machine
CCDT	client channel definition table
CCSID	coded character set identifier
CICS	Customer Information Control System
COA	confirm on arrival
COBOL	Common Business Oriented Language
COD	confirm on delivery
DBCS	double-byte character set
DHCP	Dynamic Host Configuration Protocol
DNS	domain name server
EBCDIC	Extended Binary Coded Decimal Interchange Code
FDC	failure data capture
FFDC	first failure data capture
FIFO	first-in first-out
GMO	get message options
HA	high availability
HP-UX	Hewlett Packard UNIX
HTTP	Hypertext Transport Protocol
HTTPS	Hypertext Transport Protocol Secure
IBM	International Business Machines Corporation
IMS	Information Management System
IP	Internet Protocol
ISO	International Organization for Standardization
IT	Information Technology
JDBC	Java Database Connectivity API
JMS	Java Message Service
JORAM	Java Open Reliable Asynchronous Messaging

JSON	JavaScript Object Notation
JVM	Java virtual machine
LDAP	Lightweight Directory Access Protocol
LTS	long term support
MCA	message channel agent
MLS	multi-level string
MQ	Message Queue
MQI	Message Queue Interface
MQMD	MQ message descriptor
MVS	Multiple Virtual Storage
NASA	National Aeronautics and Space Administration
npm	name assigned to a package manager for Java script language
OASIS	Organization for the Advancement of Structured Information Standards
PCF	programmable command format
PI	project interchange
PMO	put message option
QoS	quality of service
RFH2	rules and format header version 2
RHEL	Red Hat Enterprise Linux
RPG	Report Program Generator programming language
SASL	Simple Authentication and Security Layer
SDK	software development kit
SQL	Structured Query Language
SSL	Secure Sockets Layer
TCP	Transmission Control Protocol
TLS	Transport Layer Security
TSH	transmission segment header
TTL	time to live
UI	user interface
UIDAI	Unique Identification Authority of India
UNIX	Uniplexed Information and Computing System
URI	Uniform Resource Identifier
URL	Uniform Resource Locator
UTF	Unicode Transformation Format
XA	extended architecture

XML	Extensible Markup Language
z/OS	zSeries operating system



IBM Training



© Copyright International Business Machines Corporation 2017.