IBM WebSphere MQ V5.3
Application Programming
(Course Code SW313)

Student Notebook
ERC 5.3

WebSphere Training and Technical Enablement
Trademarks

IBM® is a registered trademark of International Business Machines Corporation.

The following are trademarks of International Business Machines Corporation in the United States, or other countries, or both:

AIX®
CICS®
DB2®
iSeries™
MVS/ESA™
OS/400®
S/390®
VisualAge®
z/OS®

AS/400®
CICS Connection®
Encina®
MQSeries®
OS/2®
Perform™
SupportPac™
VSE/ESA™
zSeries®

C/370™
Current®
IMS™
MVS™
OS/390®
PowerPC®
System/390®
WebSphere®

Java and all Java-based trademarks are trademarks of Sun Microsystems, Inc. in the United States, other countries, or both.

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

Intel is a trademark of Intel Corporation in the United States, other countries, or both.

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

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

Other company, product and service names may be trademarks or service marks of others.

October 2004 Edition

The information contained in this document has not been submitted to any formal IBM test and is distributed on an “as is” basis without any warranty either express or implied. The use of this information or the implementation of any of these techniques is a customer responsibility and depends on the customer’s ability to evaluate and integrate them into the customer’s operational environment. While each item may have been reviewed by IBM for accuracy in a specific situation, there is no guarantee that the same or similar results will result elsewhere. Customers attempting to adapt these techniques to their own environments do so at their own risk.

© Copyright International Business Machines Corporation 1997, 2004. All rights reserved.

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

Note to U.S. Government Users — Documentation related to restricted rights — Use, duplication or disclosure is subject to restrictions set forth in GSA ADP Schedule Contract with IBM Corp.
Contents

Trademarks ............................................................................ xi

Course Description ................................................................. xiii

Agenda ................................................................................ xlv

Unit 1. Overview ................................................................. 1-1
  Unit Objectives .................................................................. 1-2
1.1 Product Overview .......................................................... 1-3
  Islands of Automation ....................................................... 1-4
  Three Styles of Communication ......................................... 1-5
  Shielding Developers ....................................................... 1-6
  Messaging Styles .............................................................. 1-7
  Queues, Messages, Files and Records ................................. 1-8
  Queue Managers and Database Managers ......................... 1-9
  WebSphere MQ Platforms ................................................ 1-10
  WebSphere MQ Clients .................................................... 1-11
  The Supported Programming Languages ............................ 1-12
  WebSphere MQ - Commercial Messaging ......................... 1-13
1.2 Introduction to the MQI .................................................. 1-15
  WebSphere MQ APIs ........................................................ 1-16
  The Message Queue Interface Calls ................................... 1-17
  Message = Header + Application Data ............................... 1-19
  Elementary Data Types .................................................... 1-20
  Structures Used in the MQI ............................................... 1-21
  COPY and INCLUDE Files ................................................. 1-22
  MQI Constants ................................................................ 1-23
  Building a WebSphere MQ Application .............................. 1-24
  Checkpoint ..................................................................... 1-25
  Unit Summary ................................................................... 1-26
  Exercise 1 ....................................................................... 1-27

Unit 2. Major Calls - Housekeeping ..................................... 2-1
  Unit Objectives ................................................................ 2-2
2.1 MQCONN ................................................................. 2-3
  MQCONN ....................................................................... 2-4
  Connection Handle .......................................................... 2-5
  MQCONN Completion and Reason Codes ......................... 2-6
  CICS Connection ............................................................. 2-8
  Batch Connection ............................................................. 2-9
  IMS Connection ............................................................... 2-10
  Scope of Connection Handle ............................................. 2-11
  Connecting to More Than One Queue Manager .................. 2-12
  MQCONNX ................................................................. 2-13
Connection Options .............................................................. 2-14
Access Control ................................................................. 2-16
MQCONN Pseudocode ....................................................... 2-17
Checkpoint 1 - Coding an MQCONN .................................. 2-19
Coding an MQCONN in C .................................................. 2-20
Coding an MQCONN in COBOL .......................................... 2-21

2.2 MQOPEN ................................................................. 2-23
MQOPEN ........................................................................... 2-24
Object Handle ................................................................. 2-25
More Than One Object Handle ......................................... 2-26
The Object Descriptor ...................................................... 2-27
OPEN Options .................................................................... 2-29
MQOPEN Share Options .................................................... 2-30
MQOPEN Reason Codes .................................................... 2-31
MQOPEN Security ............................................................ 2-32
MQOPEN Pseudocode (1 of 2) ........................................... 2-33
MQOPEN Pseudocode (2 of 2) ........................................... 2-34
Checkpoint 2 - Coding an MQOPEN ................................. 2-35
Coding an MQOPEN Call in C ........................................... 2-36
Coding an MQOPEN Call in COBOL ................................. 2-38

2.3 MQCLOSE ................................................................. 2-41
MQCLOSE .......................................................................... 2-42
MQCLOSE Reason Codes .................................................. 2-43
MQCLOSE Pseudocode ..................................................... 2-44

2.4 MQDISC ................................................................. 2-45
MQDISC ............................................................................. 2-46
MQDISC Reason Codes ..................................................... 2-47
MQDISC Pseudocode ....................................................... 2-48
Checkpoint ........................................................................ 2-49
Unit Summary ..................................................................... 2-50

Unit 3. Major Call - MQPUT ............................................. 3-1
Unit Objectives .................................................................... 3-2

3.1 MQPUT ................................................................. 3-3
MQPUT ................................................................................. 3-4
The Message Descriptor ..................................................... 3-5
Put Message Options (MQPMO) ......................................... 3-7
MQPUT Reason Codes ....................................................... 3-9
MQPUT Pseudocode (1 of 3) .............................................. 3-10
MQPUT Pseudocode (2 of 3) .............................................. 3-11
MQPUT Pseudocode (3 of 3) .............................................. 3-12
Checkpoint ........................................................................ 3-13
Unit Summary ..................................................................... 3-14
Exercise 2 ........................................................................... 3-15

Unit 4. Opening Queues and Message Descriptor ............ 4-1
Unit Objectives .................................................................... 4-2
## 4.1 Opening Queues
- Queue Independence .................................................. 4-3
- Alias Queues ............................................................... 4-5
- Queue Name Resolution .................................................. 4-6
- Model Queue ............................................................... 4-7
- Dynamic Queue Names ................................................... 4-8
- Dynamic Reply-To Queues Pseudocode ............................... 4-9

## 4.2 The Elements of Remote Queuing
- Remote Queuing (1 of 2) ................................................. 4-12
- Remote Queuing (2 of 2) .................................................. 4-14
- The Dead Letter Queue ................................................... 4-15
- Undelivered Message ..................................................... 4-16
- Uses of the DLQ ............................................................ 4-17

## 4.3 MQMD Fields
- Persistence (1 of 2) ....................................................... 4-20
- Persistence (2 of 2) ....................................................... 4-21
- Priority ................................................................. 4-22
- Message Types ........................................................... 4-23
- Specifying a Local Reply-to Queue ................................... 4-24
- Specifying a Remote Reply-to Queue ................................. 4-25
- Reply to Queuing ....................................................... 4-26
- Report Field Options .................................................... 4-27
- Exception Reports ....................................................... 4-29
- Feedback Field Constant Values ...................................... 4-30
- Encoding ................................................................. 4-31
- CodedCharSetId .......................................................... 4-32
- Format Field ............................................................. 4-33
- What Applications Should Do ......................................... 4-34
- Checkpoint ............................................................... 4-35
- Unit Summary ............................................................ 4-36

## Unit 5. Major Calls - MQGET and MQPUT1
- Unit Objectives ............................................................ 5-2

### 5.1 MQGET
- MQGET ................................................................. 5-3
- Get Message Options (MQGMO) ....................................... 5-5
- Buffer Length (1 of 2) .................................................. 5-7
- Buffer Length (2 of 2) .................................................. 5-8
- MQGET Reason Codes .................................................... 5-9
- MQGET Pseudocode (1 of 3) ......................................... 5-11
- MQGET Pseudocode (2 of 3) ......................................... 5-12
- MQGET Pseudocode (3 of 3) ......................................... 5-13
- Sending Replies to the Correct Reply-to Queue ................. 5-14
- Replying Pseudocode ................................................... 5-15

### 5.2 MQPUT1
- MQPUT1 ................................................................. 5-17
- MQPUT1 Security ........................................................ 5-18
### Unit 6. Controlling Message Retrieval

<table>
<thead>
<tr>
<th>Topic</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Unit Objectives</td>
<td>6-2</td>
</tr>
<tr>
<td>6.1  MsgId and CorrelId</td>
<td>6-3</td>
</tr>
<tr>
<td>MsgId and CorrelId</td>
<td>6-4</td>
</tr>
<tr>
<td>Request/Reply Queue Consideration</td>
<td>6-5</td>
</tr>
<tr>
<td>MsgId, CorrelId and Application Parallelism</td>
<td>6-6</td>
</tr>
<tr>
<td>Retrieving Every Message</td>
<td>6-8</td>
</tr>
<tr>
<td>Retrieval by MsgId and CorrelId</td>
<td>6-9</td>
</tr>
<tr>
<td>MsgId, CorrelId, Reports and Replies</td>
<td>6-11</td>
</tr>
<tr>
<td>MsgId and CorrelId - An Example</td>
<td>6-13</td>
</tr>
<tr>
<td>MSGID Pseudocode Program 1</td>
<td>6-15</td>
</tr>
<tr>
<td>MSGID Pseudocode Program 2 (1 of 2)</td>
<td>6-16</td>
</tr>
<tr>
<td>MSGID Pseudocode Program 2 (2 of 2)</td>
<td>6-17</td>
</tr>
<tr>
<td>MSGID Pseudocode Programs 3 and 4</td>
<td>6-18</td>
</tr>
<tr>
<td>MSGID Pseudocode Program 5 (1 of 2)</td>
<td>6-19</td>
</tr>
<tr>
<td>MSGID Pseudocode Program 5 (2 of 2)</td>
<td>6-20</td>
</tr>
<tr>
<td>MsgId and CorrelId - An Example Solution</td>
<td>6-21</td>
</tr>
<tr>
<td>6.2  Browse</td>
<td>6-23</td>
</tr>
<tr>
<td>Browse</td>
<td>6-24</td>
</tr>
<tr>
<td>Browsing the Same Message</td>
<td>6-26</td>
</tr>
<tr>
<td>Browsing Pseudocode (1 of 2)</td>
<td>6-27</td>
</tr>
<tr>
<td>Browsing Pseudocode (2 of 2)</td>
<td>6-28</td>
</tr>
<tr>
<td>6.3  Waiting for Replies</td>
<td>6-29</td>
</tr>
<tr>
<td>WAIT</td>
<td>6-30</td>
</tr>
<tr>
<td>WAIT with WaitInterval</td>
<td>6-31</td>
</tr>
<tr>
<td>WAIT with WaitInterval Pseudocode (1 of 2)</td>
<td>6-32</td>
</tr>
<tr>
<td>WAIT with WaitInterval Pseudocode (2 of 2)</td>
<td>6-33</td>
</tr>
<tr>
<td>Another Example of WAIT</td>
<td>6-34</td>
</tr>
<tr>
<td>Server</td>
<td>6-35</td>
</tr>
<tr>
<td>MQGET with SET SIGNAL Option (z/OS and Compaq NSK Only)</td>
<td>6-36</td>
</tr>
<tr>
<td>SET_SIGNAL Example (z/OS) (1 of 3)</td>
<td>6-38</td>
</tr>
<tr>
<td>SET_SIGNAL Example (z/OS) (2 of 3)</td>
<td>6-39</td>
</tr>
<tr>
<td>SET_SIGNAL Example (z/OS) (3 of 3)</td>
<td>6-40</td>
</tr>
<tr>
<td>SIGNAL Pseudocode</td>
<td>6-41</td>
</tr>
<tr>
<td>Expiry</td>
<td>6-42</td>
</tr>
<tr>
<td>Checkpoint - Building Replies and Reports</td>
<td>6-43</td>
</tr>
<tr>
<td>REPLY</td>
<td>6-44</td>
</tr>
<tr>
<td>REPORT</td>
<td>6-45</td>
</tr>
<tr>
<td>Checkpoint</td>
<td>6-47</td>
</tr>
<tr>
<td>Unit Summary</td>
<td>6-48</td>
</tr>
</tbody>
</table>
Exercise 4 ................................................................. 6-49

Unit 7. MQINQ and MQSET ............................................ 7-1
  Unit Objectives ...................................................... 7-2
  7.1 MQINQ and MQSET .............................................. 7-3
    Why MQINQ and MQSET? ......................................... 7-4
    MQINQ ............................................................. 7-6
    MQINQ Pseudocode (1 of 2) .................................... 7-8
    MQINQ Pseudocode (2 of 2) .................................... 7-9
    Format of MQSET ................................................. 7-10
    MQSET Pseudocode (1 of 2) .................................... 7-12
    MQSET Pseudocode (2 of 2) .................................... 7-13
    Checkpoint ........................................................ 7-14
    Unit Summary ..................................................... 7-15
    Exercise 5 .......................................................... 7-16

Unit 8. Transaction Support and Triggering ....................... 8-1
  Unit Objectives ..................................................... 8-2
  8.1 Syncpoint Control Calls ...................................... 8-3
    Transactions ....................................................... 8-4
    Local and Global Transactions ................................ 8-6
    Syncpoint ........................................................ 8-8
    MQPUT within Syncpoint Control ................................ 8-9
    MQGET within Syncpoint Control ................................ 8-10
    MQBEGIN .......................................................... 8-11
    MQCMIT ............................................................ 8-12
    MQBACK ............................................................ 8-13
    Syncpoint Pseudocode ........................................... 8-14
    Poisoned Messages ............................................... 8-15
    Mark Skip Backout (z/OS Only) ................................ 8-16
    Remote Updates ................................................... 8-17
    Coordination Choices ............................................ 8-18
    Rollback Pseudocode ............................................ 8-19
    Clients and Transactions ....................................... 8-20
    Extended Transaction Managers ................................. 8-22
  8.2 Triggering ......................................................... 8-23
    Triggering .......................................................... 8-24
    Trigger Types ..................................................... 8-25
    Trigger Type Depth ............................................... 8-26
    MQTM (Trigger Message) ........................................ 8-27
    Triggered Application Pseudocode .............................. 8-29
    Triggering and Syncpoint ....................................... 8-30
    Checkpoint ........................................................ 8-31
    Unit Summary ..................................................... 8-32
    Exercise 6 .......................................................... 8-33
## Unit 9. MQI Security

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Unit Objectives</td>
<td>9-2</td>
</tr>
<tr>
<td>9.1 MQI Security</td>
<td>9-3</td>
</tr>
<tr>
<td>Message Context</td>
<td>9-4</td>
</tr>
<tr>
<td>Context</td>
<td>9-5</td>
</tr>
<tr>
<td>Default Context</td>
<td>9-6</td>
</tr>
<tr>
<td>No Context</td>
<td>9-7</td>
</tr>
<tr>
<td>Passing Context</td>
<td>9-8</td>
</tr>
<tr>
<td>Context Handling</td>
<td>9-9</td>
</tr>
<tr>
<td>Programs That Set Context</td>
<td>9-11</td>
</tr>
<tr>
<td>Pass Context Pseudocode (1 of 2)</td>
<td>9-12</td>
</tr>
<tr>
<td>Pass Context Pseudocode (2 of 2)</td>
<td>9-13</td>
</tr>
<tr>
<td>Alternate User Authority</td>
<td>9-14</td>
</tr>
<tr>
<td>Alternate UserId Pseudocode (1 of 2)</td>
<td>9-15</td>
</tr>
<tr>
<td>Alternate UserId Pseudocode (2 of 2)</td>
<td>9-16</td>
</tr>
<tr>
<td>Checkpoint</td>
<td>9-17</td>
</tr>
<tr>
<td>Unit Summary</td>
<td>9-18</td>
</tr>
<tr>
<td>Exercise 7</td>
<td>9-19</td>
</tr>
</tbody>
</table>

## Unit 10. Message Groups and Segmentation

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Unit Objectives</td>
<td>10-2</td>
</tr>
<tr>
<td>10.1 Message Groups and Message Segmentation</td>
<td>10-3</td>
</tr>
<tr>
<td>Physical and Logical Messages, Segments and Groups</td>
<td>10-4</td>
</tr>
<tr>
<td>Why?</td>
<td>10-5</td>
</tr>
<tr>
<td>Message Groups</td>
<td>10-6</td>
</tr>
<tr>
<td>Grouping Logical Messages</td>
<td>10-7</td>
</tr>
<tr>
<td>Retrieving Logical Messages (1 of 2)</td>
<td>10-9</td>
</tr>
<tr>
<td>Retrieving Logical Messages (2 of 2)</td>
<td>10-11</td>
</tr>
<tr>
<td>Spanning Units of Work</td>
<td>10-13</td>
</tr>
<tr>
<td>Message Segmentation</td>
<td>10-15</td>
</tr>
<tr>
<td>Segmentation by the Queue Manager</td>
<td>10-16</td>
</tr>
<tr>
<td>Reassembly by the Queue Manager</td>
<td>10-17</td>
</tr>
<tr>
<td>Segmentation by the Application</td>
<td>10-18</td>
</tr>
<tr>
<td>Reassembly by the Application (1 of 2)</td>
<td>10-20</td>
</tr>
<tr>
<td>Reassembly by the Application (2 of 2)</td>
<td>10-22</td>
</tr>
<tr>
<td>Segmentation and Message Group: Sample Put</td>
<td>10-24</td>
</tr>
<tr>
<td>Segmentation and Message Group: Sample Get</td>
<td>10-26</td>
</tr>
<tr>
<td>Segmented Messages and Reports</td>
<td>10-27</td>
</tr>
<tr>
<td>Checkpoint</td>
<td>10-28</td>
</tr>
<tr>
<td>Unit Summary</td>
<td>10-29</td>
</tr>
<tr>
<td>Exercise 8</td>
<td>10-30</td>
</tr>
</tbody>
</table>

## Unit 11. Distribution Lists

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Unit Objectives</td>
<td>11-2</td>
</tr>
<tr>
<td>11.1 Distribution Lists</td>
<td>11-3</td>
</tr>
<tr>
<td>What Is a Distribution List?</td>
<td>11-4</td>
</tr>
<tr>
<td>Opening a Distribution List (1 of 3)</td>
<td>11-5</td>
</tr>
</tbody>
</table>
Opening a Distribution List (2 of 3) ........................................... 11-7
Opening a Distribution List (3 of 3) ........................................... 11-8
Adding a Message to a Distribution List (1 of 3) .......................... 11-9
Adding a Message to a Distribution List (2 of 3) .......................... 11-11
Adding a Message to a Distribution List (3 of 3) .......................... 11-12
Closing a Distribution List ...................................................... 11-13
Remote Queues and Distribution Lists (1 of 3) ............................ 11-14
Remote Queues and Distribution Lists (2 of 3) ............................ 11-16
Remote Queues and Distribution Lists (3 of 3) ............................ 11-17
Distribution List Pseudocode (1 of 2) ....................................... 11-18
Distribution List Pseudocode (2 of 2) ....................................... 11-19
Checkpoint ................................................................. 11-20
Unit Summary ............................................................. 11-21
Exercise 9 ................................................................. 11-22

Appendix A. Checkpoint Solutions ........................................... A-1

Appendix B. Housekeeping Calls Solution .................................. B-1
  Housekeeping Calls Paper Exercises C Solutions ........................ .B-1

Appendix C. Reference Messages ............................................. C-1
  Reference Messages ...................................................... C-2
  The Whole Picture ...................................................... C-3
  Sending Side ............................................................ C-5
  Transmission to the Remote Node ....................................... C-7
  Receiving Side .......................................................... C-9
  Several Considerations ................................................ C-10

Bibliography ............................................................... X-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® is a registered trademark of International Business Machines Corporation.

The following are trademarks of International Business Machines Corporation in the United States, or other countries, or both:

- AIX®
- CICS®
- DB2®
- iSeries™
- MVS/ESA™
- OS/400®
- S/390®
- VisualAge®
- z/OS®
- AS/400®
- CICS Connection®
- Encina®
- MQSeries®
- OS/2®
- Perform™
- SupportPac™
- VSE/ESA™
- zSeries®
- C/370™
- Current®
- IMS™
- MVS™
- OS/390®
- PowerPC®
- System/390®
- WebSphere®

Java and all Java-based trademarks are trademarks of Sun Microsystems, Inc. in the United States, other countries, or both.

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

Intel is a trademark of Intel Corporation in the United States, other countries, or both.

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

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

Other company, product and service names may be trademarks or service marks of others.
Course Description

IBM WebSphere MQ V5.3 Application Programming

Duration: 3 days

Purpose

The purpose of this course is to teach the basic skills of designing and writing business programs which issue calls to the Message Queue Interface (MQI).

Audience

Experienced business application programmers who need to develop programs using the messaging and queuing techniques provided by the WebSphere MQ products.

Prerequisites

- A basic understanding of the concepts of the messaging and queuing model and how that model has been implemented in WebSphere MQ. This information can be gained from either:
  - Course SW312 (A Technical Introduction to IBM WebSphere MQ V5.3).
  or
  - Equivalent experience
- Experience in designing business application programs
- Experience in writing programs in one of the procedural programming languages supported by the WebSphere MQ.

Objectives

On successful completion of this course, you should be able to:

- Describe and use the Message Queuing Interface (MQI)
- Design and write programs to use the MQI as implemented on the chosen platform
- Explain the differences in program design necessitated by the messaging and queuing paradigm
- Describe, in detail, the different MQI calls
• Explain the differences in the MQI across the different MQSeries platforms.
Agenda

Day 1

Welcome
Unit 1: Overview
Lab Exercise 1: Working with MQ Objects
Unit 2: Major Calls - Housekeeping
Checkpoints 1 and 2 (Optional)
Unit 3: Major Call - MQPUT
Lab Exercise 2: Message Creation - MQPUT

Day 2

Unit 4: Opening Queues and Messaging Descriptor
Unit 5: Major Calls - MQGET and MQPUT1
Lab Exercise 3: Working with Messages: TRANSFER
Unit 6: Controlling Message Retrieval
Checkpoint 3 (Optional)
Lab Exercise 4: Dynamic Queue and Managing Messages: REQUEST
Unit 7: MQINT and MQSET
Lab Exercise 5: Working with Queue Attributes: MQINQ and MQSET
Unit 8: Transaction Support and Triggering

Day 3

Lab Exercise 6: Units of Work: RESPOND and Triggering MQTMCGET
Unit 9: MQI Security
Lab Exercise 7: Passing Context Information: TRANSFER (2)
Unit 10: Message Groups and Segmentation
Lab Exercise 8: Group Messages: MQPUTG and MQGETG
Unit 11: Distribution Lists
(01:00) Lab Exercise 9: Distribution Lists: MQDSTLST
Unit 1. Overview

What This Unit Is About

This unit includes a brief introductory review of WebSphere MQ and a descriptive introduction to the WebSphere MQ API (MQI).

What You Should Be Able to Do

After completing this unit, you should be able to:

• Describe the message/queuing paradigm in detail
• List the major calls used in the MQI.

How You Will Check Your Progress

Accountability:

• Classroom discussion
• Checkpoint questions

References

SC34-6064 WebSphere MQ Application Programming Guide
SC34-6062 WebSphere MQ Application Programming Reference
SC34-6067 WebSphere MQ Using C++
SC34-6066 WebSphere MQ Using Java
GC34-6275 MQSeries Publish/Subscribe User’s Guide
http://www.ibm.com/software/integration/wmq
IBM’s WebSphere MQ Web Page
**Unit Objectives**

- Explain the differences in designing for messaging and queuing
- Describe and use the MQI
- Describe each MQI call and explain its use

**Notes:**

The intent of this course is to provide the necessary information to enable an experienced application programmer who has already attended the WebSphere MQ Technical Introduction (or has a like level of experience gained in another manner) to understand the Message Queue Interface (MQI). On successful completion, the programmer will be equipped to complete application development assignments using the knowledge gained in conjunction with the supplied manuals and course materials.

Assuming the above mentioned experience level, this unit will move quickly through the differences between traditional means of communication and messaging as well as a high level review of the Message Queue Interface (MQI).
1.1 Product Overview
Islands of Automation

The Problems

- Mixed Environments
- Complex Programming
- Limited Design Choices
- Difficult Coordination of Data

Notes:

Even within their own enterprises, most large organizations have an inheritance of IT systems from various manufacturers.

In addition, many also need to communicate with their suppliers and customers, who may have other, disparate systems.

It is obvious that severe design limitations could result from this scenario.

Finally, data held on different systems in different databases must be kept synchronized.
Three Styles of Communication

**Conversational**

Program A  →  Program B

**Messaging**

Program A  →  Program B  →  Program C

**Call and Return**

Program A

call Program B

Program B

return to Program A

Figure 1-3. Three Styles of Communication

**Notes:**

**The Conversational Style** is characterized by two or more programs executing in a cooperative manner to perform a business transaction. They communicate with each other through an architected interface. While one program is waiting for a reply from another program with which it is communicating, it may continue with other processing. APPC, CPI-C and the sockets interface of TCP/IP are examples of this type of communication.

**The Call and Return Style** is essentially synchronous. When one program calls another program, the former is blocked and cannot perform any other processing. During this time the calling program occupies system storage unproductively. Remote procedure call (RPC) is an example of this style of communication.

**The Messaging Style** as seen with WebSphere MQ allows sophisticated applications to be constructed from a number of simple, independently operating programs, each typically getting and putting one message. This is in contrast with the conversational and call-return styles where cooperating partners must be executing at the same time.
Notes:

As you can see, once a program uses WebSphere MQ and issues calls to the MQI, the issues regarding network availability are no longer in the programmer’s domain.

In fact, even if the WebSphere MQ administrator changes the protocol used to deliver messages across a network, it is of no concern to the programmer.

WebSphere MQ allows the programmer to focus on what he or she does best: business application programming.

During this course, we will see how simple it is to use the MQI and get an opportunity to try our hand at programming in this easy to use environment.
**Messaging Styles**

**Point-to-point**

![Diagram of point-to-point messaging with Application, Queue Manager, and Application connected by messages.]

**Publish/Subscribe**

![Diagram of publish/subscribe messaging with Publisher, Broker, Subscriber, and Publishers connected by messages.]

**Notes:**

Under point-to-point messaging, an application needs to know some information about the message target. Conceptually, it needs to know what queue to send the message to.

In the Publish/Subscribe (or Pub/Sub) model, an application publishes messages classifying them with a topic. These messages are sent to a message broker. Applications that want to receive messages subscribe to a topic by sending a registration to the message broker. The message broker handles distribution of messages so that subscribers receive messages that they have an interest in.

Unlike point-to-point queuing, applications in the pub/sub model are not aware of each other.
Notes:
Queues are as easy to access as files. An application may access many queues, and a queue may be accessed by many applications. However, there are some major differences to be aware of:

- The default GET (which can be optionally changed on a GET by GET basis) is a destructive read.
- When a message is PUT on a queue in syncpoint, it becomes visible to another process as soon as it is committed.

Typically, records written to a file are not available to be read until the file is closed. The immediate nature of message delivery means that messages can still be PUT by the initiating application while the retrieving application is GETting messages. This can allow for shorter overall processing. In this respect, queues behave more like database tables.
Queue Managers and Database Managers

Notes:

Similarities between database managers and queue managers
Queue managers own and control queues in a way similar to the way database managers own and control their data storage objects. They provide a programming interface to access data, and security, authorization, recovery and administrative facilities.

Differences between database managers and queue managers
Databases are designed to provide long time data storage with sophisticated search mechanisms, queues are not. A message on a queue generally indicates that a business process is incomplete; it might represent an unsatisfied request, an unprocessed reply or an unread report.
Notes:

A WebSphere MQ platform is a system environment in which an application can issue calls to the MQI.

Platforms supported can be divided into:

- Those platforms on which an application may issue MQI calls to a queue manager running on the same system or on another system (that is platforms which may be either clients or servers),
- Those platforms on which an application may only issue MQI calls to a queue manager running on the same system (that is platforms which may only be servers),
- Those platforms on which an application may only issue MQI calls to a queue manager running on another system (that is platforms which only support clients).

Note: For the latest platform information, see the WebSphere MQ Web site.
WebSphere MQ Clients

Notes:

As well as running on the systems (typically the larger platforms, such as or z/OS, AS/400, Sun Solaris, UNIX or Windows NT) where the queue manager is installed, applications can also run on a client system which does not have a queue manager, but has code to pass the MQI call to the server queue manager to which it is connected.

Most clients are free, but not surprisingly, a fast, reliable connection is advisable. There also is a an XA-compliant fee-based client.
The Supported Programming Languages

- C
- COBOL
- C++
- C#
- Java
- PL/I
- Assembler
- RPG
- TAL
- Visual Basic

Some languages are supported on several platforms; some, such as Assembler and RPG, on only one (OS/390 and AS/400, respectively.)

Notes:

The most widely supported languages are C and COBOL. You will have a choice of C and COBOL exercises in the lab sessions.
WebSphere MQ - Commercial Messaging

- A multiplatform API
- Assured message delivery
- Faster application development
- Time independent processing
- Application parallelism

Notes:

Multiplatform API: With minor differences, the MQI is the same in all current implementations.

Assured message delivery: Messages are delivered once and once only. In addition, the originator of the message may choose to be notified of conditions such as non-delivery, delivery and expiration.

Faster application development: The programmer does not have to be aware of the details of how operating systems or network transport mechanisms work.

Time independent processing: Putting and getting applications do not have to be active at the same time. If required, messages will survive even if neither application is active. This is the most important architectural characteristic of WebSphere MQ.

Application parallelism: Using WebSphere MQ messaging to split a business application into a number of processes which execute in parallel.
1.2 Introduction to the MQI
Notes:

MQI is the API that provides full access to the underlying messaging implementation, and is available for all key languages and environments.

JMS (Java Message Service) is the Java standard providing much of the function available through the MQI. It provides a portable API for asynchronous messaging. JMS has been developed by Sun Microsystems in collaboration with IBM and other vendors interested in promoting industry wide standard frameworks.

AMI (Application Messaging Interface) simplifies the handling of messages for the application developer, allowing concentration on business logic and message content. Message characteristics such as priority are specified independently in policies.

Note: The AMI is available as a Category 3 SupportPac. The End of Service date for this SupportPac is 31 December 2005, so it is not recommended to use it for new applications.
The Message Queue Interface Calls

- The MQI is a simple CALL interface - no new language skills are required
- MAJOR calls, frequently used:
  - MQCONN
  - MQOPEN
  - MQPUT
  - MQGET
  - MQCLOSE
  - MQPUT1
  - MQDISC
- MINOR calls, less frequently used:
  - MQINQ
  - MQSET
  - MQCONNX
  - MQBEGIN
  - MQCMIT
  - MQBACK

Notes:
A brief look at the calls:
- **MQCONN** Connect to a queue manager
- **MQOPEN** Open a WebSphere MQ object
- **MQPUT** Place a message on a queue
- **MQPUT1** Put a single message on a queue
  - No independent MQOPEN required
  - No independent MQCLOSE required
  - Use for single messages only for each queue
- **MQGET** Retrieve a message from a queue
- **MQCLOSE** Close a WebSphere MQ object
- **MQDISC** Disconnect from a queue manager
- **MQINQ** Inquire about attributes of a WebSphere MQ object
- **MQSET** Set certain specific queue attributes
- **MQCONNX** Connect to a queue manager using some options
- **MQBEGIN** Begin a unit of work coordinated by the queue manager
  - Only on Version 5 distributed queue managers
  - May involve external resource managers
- **MQCMIT** Syncpoint notification for all syncpointed PUTs and GETs since last syncpoint
  - Not on AS/400, z/OS using CICS or IMS.
  - Single-phase commit process only (Two-phase commit on z/OS with RRS)
- **MQBACK** Backout notification for all syncpointed PUTs and GETs since last syncpoint
  - Not on AS/400, z/OS using CICS or IMS
  - Single-phase commit process only (Two-phase commit on z/OS with RRS)
Message = Header + Application Data

Notes:
A message has two parts: the application data and a header called the message descriptor. The message descriptor contains information about the message which is used by both WebSphere MQ and the receiving application. Some of the fields it contains are:

- The type of message (MsgType)
- An identifier for the message (MsgId)
- The priority of the message (Priority)
- The identifier of the coded character set of the character data within the application data
- The name of the queue to which a reply should be sent

There is no limitation on the contents of the application data, but there is a maximum allowable length for the message whose value is platform dependent.
Elementary Data Types

<table>
<thead>
<tr>
<th></th>
<th>C typedef</th>
<th>COBOL Picture</th>
</tr>
</thead>
<tbody>
<tr>
<td>MQBYTE</td>
<td>unsigned char MQBYTE;</td>
<td>PIC X</td>
</tr>
<tr>
<td>MQBYTE[n]</td>
<td>MQBYTE MQBYTE[n];</td>
<td>PIC X(n)</td>
</tr>
<tr>
<td>MQCHAR</td>
<td>char MQBYTE;</td>
<td>PIC X</td>
</tr>
<tr>
<td>MQCHAR[n]</td>
<td>MQCHAR[n];</td>
<td>PIC X(n)</td>
</tr>
<tr>
<td>MQHCONN</td>
<td>MQLONG MQHCONN;</td>
<td>PIC S9(9) BINARY</td>
</tr>
<tr>
<td>MQHOBJ</td>
<td>MQLONG MQHOBJ;</td>
<td>PIC S9(9) BINARY</td>
</tr>
<tr>
<td>MQLONG</td>
<td>long MQLONG;</td>
<td>PIC S9(9) BINARY</td>
</tr>
<tr>
<td>PMQLONG</td>
<td>MQLONG MQPOINTER PMGLONG;</td>
<td>POINTER</td>
</tr>
</tbody>
</table>

**Notes:**

Information about WebSphere MQ defined elementary data types can be found in the Application Programming Reference, Part 1. Data type description.

MQBYTE data is not converted in any way, while MQCHAR and MQLONG data in IBM defined structures might be converted by WebSphere MQ.
Structures Used in the MQI

- **The Message Descriptor (MQMD)** - Describes the header which accompanies each message.

- **The Object Descriptor (MQOD)** - Used by MQOPEN and MQPUT1 to identify the object, usually a queue, being opened.

- **Get Message Options (MQGMO)** - Used by the programmer to specify options on an MQGET.

- **Put Message Options (MQPMO)** - Used by the programmer to specify options on an MQPUT.

Notes:

MQI applications make frequent use of the following four structures:

- **MQMD (Message Descriptor)** - Properties of the message are specified in here, either by the queue manager or by the programmer.

- **MQOD (Object Descriptor)** - Used only at MQOPEN and MQPUT1 time.

- **MQGMO (Get Message Options)** - Set before an MQGET to refine the way in which the get works; for example, to specify a wait or a browse.

- **MQPMO (Put Message Options)** - Set before an MQPUT to refine the way in which the put works.

There are other structures but the above are the most commonly used. All structures are made of the elementary data types previously discussed.
COPY and INCLUDE Files

COBOL COPY Files

<table>
<thead>
<tr>
<th></th>
<th>With initial values</th>
<th>Without initial values</th>
</tr>
</thead>
<tbody>
<tr>
<td>Message Descriptor</td>
<td>CMQMDV</td>
<td>CMQMDL</td>
</tr>
<tr>
<td>Object Descriptor</td>
<td>CMQODV</td>
<td>CMQODL</td>
</tr>
<tr>
<td>Get Message Options</td>
<td>CMQGMOV</td>
<td>CMQGMOL</td>
</tr>
<tr>
<td>Put Message Options</td>
<td>CMQPMOV</td>
<td>CMQPMOL</td>
</tr>
</tbody>
</table>

C INCLUDE File

| CMQC.H                   | One file contains declarations for all structures. |

Figure 1-17. COPY and INCLUDE Files

Notes:
The names of the fields within the various structures are documented in the Application Programming Reference manual. There is a chapter for each structure, at the end of which the structure declaration in the various supported languages is listed, as well as initial default values.
MQI Constants

<table>
<thead>
<tr>
<th>Reason Codes</th>
<th>MQRC_NO_MSG_AVAILABLE</th>
<th>MQRC_NONE</th>
</tr>
</thead>
<tbody>
<tr>
<td>Open Options</td>
<td>MQOO_INPUT_SHARED</td>
<td>MQOO_OUTPUT</td>
</tr>
<tr>
<td>Get Message Options</td>
<td>MQGMO_WAIT</td>
<td>MQGMO_SYNCPOINT</td>
</tr>
<tr>
<td>Message Types</td>
<td>MQMT_REQUEST</td>
<td>MQMT_REPLY</td>
</tr>
</tbody>
</table>

Notes:
The symbolic constants shown above are just a few of the many provided. They are valid for use in C programs. The COBOL equivalents have hyphens rather than underscores. The C constants are included with CMQC.H an additional copy file, CMQV. The full list is provided in an appendix in the Application Programming Reference.
Building a WebSphere MQ Application

Notes:

Each platform has unique requirements when building an application. There may be special compiler requirements as well as links to consider.

Since the procedure to follow for successful compilation and link is slightly different for each platform, you should refer to the Application Programming Guide for the information that is specific to the platform you are working on.

For the sake of an example, we will look at the process of compiling and linking a C application on a Windows 2000 queue manager using the Microsoft Visual C++ compiler:

```
cl amqsget0.c /link mgm.lib
```

The above example used the supplied MQM.LIB library during the link phase. This will ensure that the program is set up as a 32-bit server.
Checkpoint

1. What is the name of the header which accompanies every message?
2. How many MQI calls are there?
3. Name four structures which are commonly used in MQI programs.

Notes:

Write your answers here.
1.
2.
3.
Unit Summary

- WebSphere MQ provides and the MQI enables:
  - A multiplatform API
  - Assured message delivery
  - Faster application development
  - Time independent processing
  - Application parallelism

Notes:

The environment that will be used in the exercises has been set up for you. However, it is worthwhile to view a short demonstration of that setup as well as to become familiar with the names given to the queues and other objects being used.
Exercise 1

Do Lab Exercise 1

Notes:
Unit 2. Major Calls - Housekeeping

What This Unit Is About

Major Calls - Housekeeping will cover the MQCONN, MQCONNX, MQOPEN, MQCLOSE and MQDISC calls. It will include discussion of security as it applies to these calls as well as the procedures and effects of opening model and alias queues.

What You Should Be Able to Do

After completing this unit, you should be able to write a program to issue a:

- MQCONN call to connect to a local queue manager
- MQOPEN call to open a queue
- MQCLOSE call to close a queue
- MQDISC call to disconnect from a local queue manager.

How You Will Check Your Progress

Accountability:

- Classroom exercise

References

SC34-6064  WebSphere MQ Application Programming Guide
SC34-6062  WebSphere MQ Application Programming Reference
Unit Objectives

- Understand how to access WebSphere MQ resources in an application
- Code simple housekeeping calls

Notes:

We will start by looking at the calls used to access WebSphere MQ queue managers and a queue manager's resources. It will be a good way to get familiar with the structure of the MQI calls.

As we complete the MQCONN and MQOPEN calls, we will do a simple paper exercise to ensure we have a grasp of the way to set up a WebSphere MQ call and some of the data structures that are frequently used.

Note: In the following MQCONN visual and in similar subsequent visuals, the arrows represent the direction of data flow. Parameters that transfer data from the application to the queue manager are called input parameters in the Application Programming Reference manual; parameters which receive data from the queue manager are called output parameters.

In C, parameters that are input-only and of type MQHCONN, MQHOBJ or MQLONG are passed by value. For all other parameters, the address of the parameter is passed by value - this means that the parameter is preceded by an ampersand.
2.1 MQCONN
**Notes:**

The MQCONN call is used to connect the application program to the queue manager. Although the call is not necessary in the CICS (z/OS) and the AS/400 environments, it can be included. On other platforms, a program must connect to the queue manager using MQCONN (or, optionally, MQCONNX where supported).

The first parameter supplied for this call is the queue manager name. If left blank, this indicates the connect is intended to be the default queue manager.

The queue manager to which an application connects is known as the local queue manager.

If the call is successful, a 4-byte connection handle (Hconn) is returned along with a completion code and a reason code.
## Connection Handle

<table>
<thead>
<tr>
<th>Call</th>
<th>Parameters</th>
</tr>
</thead>
<tbody>
<tr>
<td>MQCONN</td>
<td>(name, Hconn, . . . . .)</td>
</tr>
<tr>
<td>MQOPEN</td>
<td>(Hconn, . . . . .)</td>
</tr>
<tr>
<td>MQPUT</td>
<td>(Hconn, . . . . .)</td>
</tr>
<tr>
<td>MQGET</td>
<td>(Hconn, . . . . .)</td>
</tr>
<tr>
<td>MQCLOSE</td>
<td>(Hconn, . . . . .)</td>
</tr>
<tr>
<td>MQOPEN</td>
<td>(Hconn, . . . . .)</td>
</tr>
<tr>
<td>MQPUT</td>
<td>(Hconn, . . . . .)</td>
</tr>
<tr>
<td>MQGET</td>
<td>(Hconn, . . . . .)</td>
</tr>
<tr>
<td>MQCLOSE</td>
<td>(Hconn, . . . . .)</td>
</tr>
<tr>
<td>MQDISC</td>
<td>(Hconn, . . . . .)</td>
</tr>
</tbody>
</table>

**Notes:**

The connection handle returned from the MQCONN call must be the first parameter passed on all subsequent MQI calls.

The connection handle is unique. It is not intended to be used by the program in any way but to interface with the MQI.
MQCONN Completion and Reason Codes

- Completion Codes
  - MQCC_OK
  - MQCC_WARNING
  - MQCC_FAILED

- Reason Codes
  - MQRC_NONE
  - MQRC_ALREADY_CONNECTED
  - MQRC_ANOTHER_Q_MGR_CONNECTED
  - MQRC_Q_MGRQUIESCING
  - MQRC_NOT AUTHORIZED
  - MQRC_Q_MGR NOT AVAILABLE

Notes:

All calls will result in one of the three listed completion codes:
- MQCC_OK a successful call completed
- MQCC-WARNING the call completed but reason code indicates certain conditions
- MQCC_FAILED the call did not complete successfully

The programmer needs to handle the check of completion codes and, if a warning or failure occurred, the program needs to examine the reason code to determine whether the program should continue or not.

The list of reason codes is an example of some of the most common that might occur when an MQCONN is attempted.
- MQRC_NONE will be set to this if completion code is MQCC_OK
- MQRC_ALREADY CONNECTED returned if attempt to connect to same queue manager as already connected. Completion code is MQCC_WARNING
• MQRC_ANOTHER_Q_MGR_CONNECTED indicates an attempt to connect to a second queue manager simultaneously. Completion code is MQCC_FAILED

• MQRC_Q_MGRQUIESCING is an indication that the queue manager is shutting down. Completion code is MQCC_FAILED

• MQRC_NOT_AUTHORIZED is returned if the program is not authorized to access the queue manager specified. Completion code is MQCC_FAILED

• MQRC_Q_MGR_NOT_AVAILABLE usually means that the queue manager has not been started or has stopped running. Completion code is MQCC_FAILED

There are many other possible reason codes that may be returned from an unsuccessful MQCONN attempt; the above are the most common. A complete listing is included with the description of the MQCONN call in the Application Programming Reference manual.

On all calls, the programmer should test for three types of status on:

• Everything OK, no special conditions reported (MQCC_OK and MQRC_NONE)

• Expected special conditions which can happen on various calls, which are application specific (for example MQRC_NO_MSG_AVAILABLE, MQRC_TRUNCATED_MESSAGE ACCEPTED on an MQGET)

• Unexpected conditions, such as MQRC_NOT_AUTHORIZED on an MQCONNECT or MQOPEN and MQRC_UNKNOWN_OBJECT_NAME on an MQOPEN.
Notes:

As stated earlier, it is not required to issue an MQCONN for a CICS (z/OS) program. CICS performs the connects and there is no choice of queue manager. There can only be one queue manager within a CICS region.

Inclusion of an MQCONN in a CICS program, although not required, might help you port the application to a non-CICS environment with a minimum of recoding. Be aware that MQCONN always completes successfully in a CICS environment.
Batch Connection

Notes:

Batch programs must issue an MQCONN command. However, unlike CICS, a batch program can select which queue manager to connect to (if multiple queue managers have been set up on the system). It is important to note that a batch program can generally only be connected to one queue manager at a time. This is the type of behavior that is common to most other queue managers. It is possible, by attaching additional TCBs to a batch application, to connect to additional queue managers concurrently. However, WebSphere MQ will not coordinate syncpointing or resource sharing between TCBs.

A special case exists for IMS and we will look at that next.
**Notes:**

In IMS, a program can connect to more than one queue manager simultaneously. As long as the queue manager is known to the IMS control region, a program running in the IMS-dependent region can connect to it, regardless of whether the program is already connected to other queue managers.
Scope of Connection Handle

- Scope of the handle restricted to smallest unit of parallel execution, for example:
  - z/OS
    - CICS task
    - IMS task, up to the next syncpoint
    - Batch or TSO task
  - UNIX, Windows NT/2000/XP, OS/2
    - Thread
  - OS/400
    - Job
  - Compaq NSK, Windows Client
    - Process
  - DOS Client
    - System
  - Linux
    - Thread

- Automatic MQDISC and MQCLOSE when unit ends

Notes:

The handle represents the connection to the queue manager. Used on all subsequent calls issued by the application when interacting with the queue manager, it ceases to be valid when an MQDISC call is issued. This is also true when the unit of processing that defines the scope of the connect handle terminates.

The scope of the connection handle is restricted to the smallest unit of parallel processing within the environment concerned.

The handle is not valid for use outside this scope.
Connecting to More Than One Queue Manager

- General rule:
  - Can only be connected to one queue manager at a time
  - OS/390 BATCH, TSO and IMS allow multiple concurrent connections
  - On Windows systems and OS/2, each thread can connect to a different queue manager
  - Client applications can connect to more than one queue manager within a thread

Notes:

As you can see, the rules for connecting to multiple queue managers differ among the supported platforms and environments. Therefore, it is important to consider this when writing your program. The multiple concurrent MQCONN calls supported in some environments (such as IMS) will result in a failing completion code in others with a reason MQRC_ANOTHER_Q_MGR_CONNECTED.

If there is a possibility that you will want to port your application, you need to consider the most restricted environment you will be running in and design for that situation.
Notes:

On some platforms, an alternative to MQCONN has been provided. MQCONNX connects an application to a queue manager and provides the connection handle to be used by the application in subsequent calls; but, MQCONNX allows options to be specified that control how the call works.

The connect options are passed in the MQCNO structure as one of the parameters on the MQCONNX call. For example, the options can control the type of MQ binding that will be used.

MQCONNX is supported on all Version 5 queue managers, including z/OS and certain clients.
Connection Options

- **Default Option:**
  - MQCNO_NONE

- **Binding Options:**
  - MQCNO_STANDARD_BINDING
  - MQCNO_FASTPATH_BINDING

- **Connection-tag Options (only on z/OS):**
  - MQCNO_SERIALIZE_CONN_TAG_Q_MGR
  - MQCNO_SERIALIZE_CONN_TAG_QSG
  - MQCNO.Restrict_CONN_TAG_Q_MGR
  - MQCNO.Restrict_CONN_TAG_QSG

- **Handle-sharing Options (only on UNIX, Windows, OS/400):**
  - MQCNO_HANDLE_SHARE_NONE
  - MQCNO_HANDLE_SHARE_BLOCK
  - MQCNO_HANDLE_SHARE_NO_BLOCK

**Notes:**

The default option MQCNO_NONE results in the same behavior like the MQCONN call.

The binding options can control the type of WebSphere MQ binding that will be used.

- MQCNO_STANDARD_BINDING: Standard binding causes the application and the local queue manager agent to run in separate units of execution, protecting the queue manager from corruption by an errant program.
- MQCNO_FASTPATH_BINDING: Fastpath binding allows the application and agent to be part of the same unit of execution, useful when use of multiple processes results in significant performance overhead. ONLY TRUSTED APPLICATIONS SHOULD USE THIS OPTION.

The connection-tag options are supported on z/OS only. These options control the use of the connection tag ConnTag. This is a tag that the queue manager associates with the resources that are affected by the application during this connection. Each application or application instance should use a different value for the tag, so that the queue manager can correctly serialize access to the affected resources.
The handle-sharing options control the sharing of handles between different threads (units of parallel processing) within the same process. These options are supported in the following environments: AIX, HP-UX, OS/400, Solaris, Linux, Windows.

- MQCNO_HANDLE_SHARE_NONE: No handle sharing between threads.
- MQCNO_HANDLE_SHARE_BLOCK: Serial handle sharing between threads, with call blocking.
- MQCNO_HANDLE_SHARE_NO_BLOCK: Serial handle sharing between threads, without call blocking.
Access Control

Notes:
An application running under a specific userid may not be allowed to connect to a queue manager. There is little an application can do if access is denied (MQCC_FAILED and MQRC_NOT_AUTHORIZED) except report the problem to the WebSphere MQ administrator. It may be possible that a change to a profile or access control list may be required.

In any case, no connection handle will be returned, and it is pointless for the application to continue.
### MQCONN Pseudocode

```
DEFINE DEFAULT_QMGR AS CONSTANT ' ' 1
DEFINE QMGR AS MQCHAR48 2
DEFINE CONN_HANDLE AS MQHCONN 3
DEFINE COMP_CODE AS MQLONG 4
DEFINE REASON_CODE AS MQLONG 5

QMGR = DEFAULT_QMGR 6

CALL MQCONN(QMGR, 7
    CONN_HANDLE, 8
    COMP_CODE, 9
    REASON_CODE) 10

IF COMP_CODE NOT = MQCC_OK 11
    write an error message detailing the REASON_CODE 12
    terminate the program 13
    STOP 14
END-IF 15
```

---

**Notes:**
Checkpoint 1 - Coding an MQCONN

What This Exercise Is About

The purpose of this exercise and the next one is to familiarize you with the reference materials available to you to complete some basic calls. By spending a little time now, and finding necessary structures and labels, you will be better equipped when we work on our live exercises later.

The following excerpts from the C and COBOL programs show the include/copybook statements as well as the additional labels to be used. The contents of the include (C) or copybooks (COBOL) that you will need are in the Application Programming Reference manual.

What You Should Be Able to Do

1. Learn where to look to find out how to code the calls.
2. Succeed in completing the code requested.
Coding an MQCONN in C

The following excerpts from a C program show the include statement, structure definitions and an uncompleted MQCONN call. Complete the CALL.

/******* includes for MQI **********************/

#include <cmqc.h>

/******* Declare MQI structures needed **************/

MQOD odG = {MQOD_DEFAULT}; /* Object Descriptor for GET */
MQOD odP = {MQOD_DEFAULT}; /* Object Descriptor for PUT */
MQCHAR48 QManager; /* queue manager name */
MQHCONN Hcon; /* connection handle */
MQHOBJ Hin; /* handle for MQGET */
MQHOBJ Hout; /* handle for MQPUT */
MQLONG O_options; /* MQOPEN options */
MQLONG C_options; /* MQCLOSE options */
MQLONG CompCode; /* completion code */
MQLONG Reason; /* reason code */
MQLONG CReason; /* reason code (MQCONN) */

/******* Connect to Default Queue Manager ***********/

strcpy(QManager, "");

MQCONN(

/* report reason and stop if it failed */
if (CompCode == MQCC_FAILED)
{
    printf("MQCONN ended-reason code %ld\n", CReason);
    exit(CReason);
}
Coding an MQCONN in COBOL

The following excerpts from a COBOL program show COPY statements, variable definitions and an uncompleted MQCONN call. Complete the CALL.

****** DEFINE MQI STRUCTURES AND VARIABLES ******

DATA DIVISION.
WORKING-STORAGE SECTION.
*
** Declare MQI structures needed
* MQI named constants
 01 MY-MQ-CONSTANTS.
    COPY CMQV.
* Object Descriptor
 01 OBJECT-DESCRIPTOR.
    COPY CMQODV.
** note: sample uses defaults where it can
 01 QM-NAME       PIC X(48) VALUE SPACES.
 01 HCONN         PIC S9(9) BINARY.
 01 GQ-HANDLE     PIC S9(9) BINARY.
 01 PQ-HANDLE     PIC S9(9) BINARY.
 01 OPTIONS       PIC S9(9) BINARY.
 01 COMPLETION-CODE PIC S9(9) BINARY.
 01 OPEN-CODE     PIC S9(9) BINARY.
 01 CON-REASON    PIC S9(9) BINARY.
 01 REASON        PIC S9(9) BINARY.

****** CONNECT TO DEFAULT QUEUE MANAGER ******

CALL "MQCONN"
    USING

*       report reason and stop if it failed
    IF COMPLETION-CODE IS EQUAL TO MQCC-FAILED
        DISPLAY "MQCONN ended-reason code " CON-REASON
        STOP RUN
    END-IF.

*  

END OF LAB
2.2 MQOPEN
MQOPEN

MQOPEN (Hconn, ObjDesc, Options, Hobj, CompCode, Reason)

Notes:
Before doing anything with any object (MQINQ, MQSET, MQGET, MQPUT and so forth), the object must be opened for the functions desired. By opening an object, the developer is declaring an intent on it.

All processing options are established at open time.

The only time an MQOPEN is not required is when using the MQPUT1 call to place a single message on a queue. We will look at MQPUT1 later.

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.
Object Handle

MQCONN (name, Hconn, ..........
MQOPEN (Hconn, ObjDesc, Options, Hobj, ..........
MQPUT (Hconn, Hobj, ..........
MQGET (Hconn, Hobj, ..........
MQCLOSE (Hconn, Hobj, ..........
MQOPEN (Hconn, ObjDesc, Options, Hobj, ..........
MQPUT (Hconn, Hobj, ..........
MQGET (Hconn, Hobj, ..........
MQCLOSE (Hconn, Hobj, ..........
MQDISC (Hconn, ...........

Figure 2-15. Object Handle

Notes:

Note in this example, the first parameter passed on all calls after the MQCONN is the value called Hconn (the connection handle). On return from any successful MQOPEN, a second handle (seen here as a label Hobj) is returned. The object handle represents the access that has been established to the queue, and is used in all subsequent gets and puts.

Because the MQCLOSE will invalidate the contents of Hobj, it is valid to reuse that label as shown here. Essentially, a new value will be returned in the second MQOPEN and will only be associated with the object in the second MQOPEN. However, in most cases it will be easier to maintain a separate object handle for each queue accessed, as seen on the next page.
More Than One Object Handle

Notes:

If an application wants to have more than one object opened at the same time (a common occurrence), then the programmer must set up SEPARATE handles for each one. In the example, two queues (STOCK and COST) are to be opened and worked with at the same time.

Because there is no MQCLOSE, it is important to keep the two queues separate when issuing calls to the MQI. This is done with the two different object handles (Hcost and Hstock). Both would have been defined as MQHOBJ elementary data types.

The queue manager to which your application is connected can thus identify which queue the MQPUT or MQGET is intended for.

Figure 2-16. More Than One Object Handle
The Object Descriptor

- StrucId
- Version
- ObjectType
- ObjectName
- ObjectQMgrName
- DynamicQName
- AlternateUserId

Notes:

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

As with many structures defined to WebSphere MQ, the object descriptor begins with a 4-byte constant. Called the structure identifier (MQOD_StrucId), it contains the constant value "OD". We will examine other structures as we progress.

Some of the fields in the object descriptor may be initialized by the application (defaults can be used in many cases), 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:
Fields present in Versions 1, 2 and 3 of the Object Descriptor

- StrucId is the identifier
- Version allows for potential different MQOD layouts
  - Currently this can be 1, 2 or 3. The default is 1.
- ObjectType specifies what kind of WebSphere MQ object is being opened
- ObjectName contains the actual name of the object being opened
- ObjectQMgrName can contain the name of the queue manager where the object is defined
- DynamicQName is a special value used when dynamically creating a queue
  - Dynamic and model queues will be covered later
- AlternateUserId is used if the queue is being opened on behalf of a request from another program/user

Looking up and putting a sticky note on the MQOD where the field data names are listed helps during labs.

Fields Present in Versions 2 and 3 of the Object Descriptor

Some fields were added in Version 5.0 of the distributed queue managers. These were to support distribution lists, not available on OS/930.

Fields Present in Version 3 of the Object Descriptor

Further fields were subsequently added. They include the resolved queue name and a Windows NT security identifier.
OPEN Options

- Open Options specify operations required
  - MQOO_INPUT_SHARED
  - MQOO_INPUT_EXCLUSIVE
  - MQOO_INPUT_AS_Q_DEF
  - MQOO_OUTPUT
  - MQOO_BROWSE
  - MQOO_INQUIRE
  - MQOO_SET
  - MQOO_FAIL_IF_QUIESCING

- Object can be opened for a combination of options
  - Options must be valid for object type
  - User must be authorized for all options requested
  - Specified options must not be contradictory

Notes:

OPEN options are the means by which the application declares its intent on the object being opened. Those listed are some of the most common but there are others.

It is valid to combine (add together) multiple options as long as they do not conflict. For instance, a common pair of options is MQOO_INPUT_SHARED and MQOO_BROWSE. Generally, a browse will be done to determine if there are any messages that meet some processing criteria. If so, the application will want to take the message off the queue for processing. However, combining MQOO_INPUT_EXCLUSIVE and MQOO_INPUT_SHARED is not valid.

Note that the program can defer to the queue definition for the input option to be used.
**Notes:**

If a program opens a queue using MQOO_INPUT_SHARED, other programs can also open the queue for input (shared). However, if any program successfully opens the queue using MQOO_INPUT_EXCLUSIVE, other programs attempting to open the queue for any input option will fail with MQRC_OBJECT_IN_USE.

It is important to realize that successfully opening a queue for input (exclusive) does not keep other applications from accessing the queue for other purposes (for example MQOO_OUTPUT).

Any attempt to open a queue for input (exclusive) while another program already has it opened for input (shared or exclusive) will result in an MQRC_OBJECT_IN_USE failure.
MQOPEN Reason Codes

- MQRC_NONE
- MQRC_HANDLE_NOT_AVAILABLE
- MQRC_HCONN_ERROR
- MQRC_NOT_AUTHORIZED
- MQRC_OBJECT_IN_USE
- MQRC_OD_ERROR
- MQRC_OPTION_NOT_VALID_FOR_TYPE
- MQRC_OPTIONS_ERROR
- MQRC_PAGESET_ERROR
- MQRC_STORAGE_NOT_AVAILABLE
- MQRC_UNEXPECTED_ERROR
- MQRC_UNKNOWN_OBJECT_NAME

Notes:

There are many reason codes that can be returned if an MQOPEN is unsuccessful. The Application Programming Reference manual contains a description of each along with recommended actions.
Notes:

When an MQOPEN is issued, it is possible that the user/application is not authorized for all the options requested. If not, the MQOPEN will fail with a MQRC_NOT_AUTHORIZED reason code.

If the options specified are accurate, then the WebSphere MQ system administrator would have to ensure that the proper updates to permit access were completed.
Figure 2-22. MQOPEN Pseudocode (1 of 2)

Notes:
MQOPEN Pseudocode (2 of 2)

<table>
<thead>
<tr>
<th>Line</th>
<th>Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>25</td>
<td>OBJ_DESC.StrucId = MQOD_STRUC_ID</td>
</tr>
<tr>
<td>26</td>
<td>OBJ_DESC.Version = MQOD_VERSION_1</td>
</tr>
<tr>
<td>27</td>
<td>OBJ_DESC.ObjectType = MQOT_Q</td>
</tr>
<tr>
<td>28</td>
<td>OBJ_DESC.ObjectName = &quot;MYQUEUE&quot;</td>
</tr>
<tr>
<td>29</td>
<td>OBJ_DESC.ObjectQMgrName= DEFAULT_QMGR</td>
</tr>
<tr>
<td>30</td>
<td>OPEN_OPTIONS = MQOO_OUTPUT + MQOO_FAIL_IF QUIESCING</td>
</tr>
<tr>
<td>31</td>
<td>CALL MQOPEN(CONN_HANDLE, OBJ_DESC, OPEN_OPTIONS, MYQUEUE_HANDLE, COMP_CODE, REASON_CODE)</td>
</tr>
<tr>
<td>32</td>
<td>IF COMP_CODE NOT = MQCC_OK</td>
</tr>
<tr>
<td>33</td>
<td>write an error message detailing the REASON_CODE</td>
</tr>
<tr>
<td>34</td>
<td>terminate the program</td>
</tr>
<tr>
<td>35</td>
<td>STOP</td>
</tr>
<tr>
<td>36</td>
<td>END-IF</td>
</tr>
</tbody>
</table>

Notes:
Checkpoint 2 - Coding an MQOPEN

What This Exercise Is About

This exercise is intended to give you practice in writing another call.

What You Should Be Able to Do

1. Learn where to look to find out how to code the calls.
2. Succeed in completing the code requested.
Coding an MQOPEN Call in C

The following excerpts from a C program show the include statement, structure definitions and uncompleted MQOPEN calls. Complete the CALLs.

/******* includes for MQI ****************************/

#include <cmqc.h>

/****** Declare MQI structures needed **************/

MQOD    odG = {MQOD_DEFAULT};  /* Object Descriptor for GET */
MQOD    odP = {MQOD_DEFAULT};  /* Object Descriptor for PUT */
MQCHAR48 QManager;       /* queue manager name */
MQHCONN  Hcon;           /* connection handle */
MQHOBJ   Hin;    /* handle for MQGET */
MQHOBJ   Hout;           /* handle for MQPUT */
MQLONG   O_options;      /* MQOPEN options */
MQLONG   C_options;      /* MQCLOSE options */
MQLONG   CompCode;       /* completion code */
MQLONG   Reason;         /* reason code */
MQLONG   CReason;        /* reason code (MQCONN) */
/********** Initialize Queue Names *******************/

strcpy(odG.ObjectName, "LAB00.INPUT");
strcpy(odP.ObjectName, "LAB00.OUTPUT");

/*** Open the input queue for shared input ***********/

O_options = MQOO_INPUT_SHARED /* open queue for shared input */
    + MQOO_FAIL_IF_QUIESCING; /* but not if MQM stopping */

MQOPEN(

    /* report reason if any; stop if it failed      */
    if (Reason != MQRC_NONE)
    {
        printf("MQOPEN (i/p) ended-reason code %ld\n", Reason);
    }
    if (CompCode == MQCC_FAILED)
    {
        exit(Reason);
    }

    /**** Open the ouput queue *******************/

O_options = MQOO_OUTPUT        /* open queue for output      */
    + MQOO_FAIL_IF_QUIESCING; /* but not if MQM stopping */

MQOPEN(

    /* report reason if any; stop if it failed      */
    if (Reason != MQRC_NONE)
    {
        printf("MQOPEN (o/p) ended-reason code %ld\n", Reason);
    }
    if (CompCode == MQCC_FAILED)
    {
        exit(Reason);
    }
Coding an MQOPEN Call in COBOL

The following excerpts from a COBOL program show COPY statements, variable definitions and an uncompleted MQCONN call. Complete the CALLs.

****** DEFINE MQI STRUCTURES AND VARIABLES ******

DATA DIVISION.
WORKING-STOREAGE SECTION.
*
** Declare MQI structures needed
* MQI named constants
  01 MY-MQ-CONSTANTS.
      COPY CMQV.
* Object Descriptor
  01 OBJECT-DESCRIPTOR.
      COPY CMQODV.
** note: sample uses defaults where it can
  01 QM-NAME PIC X(48) VALUE SPACES.
  01 HCONN PIC S9(9) BINARY.
  01 GQ-HANDLE PIC S9(9) BINARY.
  01 PQ-HANDLE PIC S9(9) BINARY.
  01 OPTIONS PIC S9(9) BINARY.
  01 COMPLETION-CODE PIC S9(9) BINARY.
  01 OPEN-CODE PIC S9(9) BINARY.
  01 CON-REASON PIC S9(9) BINARY.
  01 REASON PIC S9(9) BINARY.
***** OPEN SHARED INPUT QUEUE *****************************************

OPENS.
  MOVE SPACES to MQD-OBJECTNAME.
  MOVE "LAB00.INPUT" TO MQD-OBJECTNAME.
  ADD MQOO-INPUT-SHARED MQOO-FAIL-IF-QUIESCING
      GIVING OPTIONS.
  CALL "MQOPEN"
  USING

*                   report reason, if any; stop if failed
IF REASON IS NOT EQUAL TO MQRC-NONE
  DISPLAY "MQOPEN (i/p) ended-reason code " REASON
END-IF.

IF OPEN-CODE IS EQUAL TO MQCC-FAILED
  DISPLAY "unable to open queue for input"
  STOP RUN
END-IF.

****** OPEN OUTPUT QUEUE ***********************************************

MOVE "LAB00.OUTPUT" to MQD-OBJECTNAME.
ADD MQOO-OUTPUT MQOO-FAIL-IF-QUIESCING
      GIVING OPTIONS.
CALL "MQOPEN"
USING

IF REASON IS NOT EQUAL TO MQRC-NONE
  DISPLAY "MQOPEN (o/p) ended-reason code " REASON
END-IF.

IF OPEN-CODE IS EQUAL TO MQCC-FAILED
  DISPLAY "unable to open queue for output"
  STOP RUN
END-IF.

END OF LAB
2.3 MQCLOSE
**Notes:**

The MQCLOSE call includes the connection handle and the object handle that was issued at the MQOPEN. In addition, there are a few options that can be set.

- **MQCO_NONE** - this must be specified for predefined queues, temporary dynamic queues created by someone else and objects other than queues.

- **MQCO_DELETE** - Delete the queue
  - If a temporary dynamic queue, this option purges messages and deletes the queue if the call is issued by the creator of the queue.
  - If a permanent dynamic queue, it will be deleted only if the queue is empty and there are no outstanding uncommitted get or put requests.

- **MQCO_DELETE_PURGE** - Purge messages and delete the queue
  - If a temporary dynamic queue, this option has the same effect as the options above.
  - If a permanent dynamic queue, it will be deleted only if there are no outstanding uncommitted get or put requests.
MQCLOSE Reason Codes

- MQRC_NONE
- MQRC_HCONN_ERROR
- MQRC_HOBJ_ERROR
- MQRC_OPTION_NOT_VALID_FOR_TYPE
- MQRC_OPTIONS_ERROR
- MQRC_PAGESET_ERROR
- MQRC_STORAGE_NOT_AVAILABLE
- MQRC_UNEXPECTED_ERROR

Notes:

Although the program may have finished with the object, a failure to close properly could be an indication of a wider problem. Therefore, all reason codes should be reported and properly handled.

- MQRC_NONE - Close succeeded
- MQRC_HCONN_ERROR - Connection handle not valid
- MQRC_HOBJ_ERROR - Object handle not valid
- MQRC_OPTION_NOT_VALID_FOR_TYPE - Attempting to use MQCO_DELETE or MQCO_DELETE_PURGE and the object was not a dynamic queue.
- MQRC_OPTIONS_ERROR - Attempted to supply invalid option or to combine options that are mutually exclusive.
- MQRC_PAGESET_ERROR - In z/OS, error occurred attempting to access the page set data set.
- MQRC_STORAGE_NOT_AVAILABLE - Insufficient storage is available to the queue manager to complete the call.
- MQRC_UNEXPECTED_ERROR - An error with an unknown cause occurred.
MQCLOSE Pseudocode

```
DEFINE CONN_HANDLE AS MQHCONN 1
DEFINE MYQUEUE_HANDLE AS MQOBJ 2
DEFINE COMP_CODE AS MQLONG 3
DEFINE REASON_CODE AS MQLONG 4
DEFINE CLOSE_OPTIONS AS MQLONG 5

CLOSE_OPTIONS=MQCO_NONE 6

CALL MQCLOSE(CONN_HANDLE, 7
   MYQUEUE_HANDLE, 8
   CLOSE_OPTIONS, 9
   COMP_CODE, 10
   REASON_CODE) 11

IF COMP_CODE NOT = MQCC_OK 12
   write an error message detailing the REASON_CODE 13
   terminate the program 14
   STOP 15
END-IF 16
```

Figure 2-26. MQCLOSE Pseudocode

Notes:
2.4 MQDISC
**Notes:**

Once a program is finished working with a queue manager and its resources, it should disconnect from the queue manager using the MQDISC.

The MQDISC is a simple call and has very few parameters; notice there are no options. The result of a successful MQDISC in most cases is the invalidation of the connection handle.

As we work in labs, you will see that a program might check a reason code that was saved from the MQCONN. If the program arbitrarily issues an MQDISC and was running in the same thread or process of, perhaps, a calling program, it can invalidate the connection handle for the other program as well. An example of this is the way the supplied trigger monitor `runmqtrm` behaves if it starts a program synchronously.
MQDISC Reason Codes

- MQRC_NONE
- MQRC_HCONN_ERROR
- MQRC_CONNECTION_BROKEN
- MQRC_PAGESET_ERROR
- MQRC_STORAGE_NOT_AVAILABLE
- MQRC_UNEXPECTED_ERROR

Notes:
Although a failed MQDISC may not seem important to the disconnecting application, it is important to check for and report any unexpected results.
MQDISC Pseudocode

```
DEFINE CONN_HANDLE AS MQHCONN 1
DEFINE COMP_CODE AS MQLONG   2
DEFINE REASON_CODE AS MQLONG 3

CALL MQDISC                  5
  (CONN_HANDLE,             6
   COMP_CODE,             7
   REASON_CODE)          8

IF COMP_CODE NOT = MQCC_OK  11
  write an error message detailing the REASON_CODE  12
  terminate the program  13
  STOP                  14
END-IF                  15
```

Figure 2-29. MQDISC Pseudocode  SW3135.3

Notes:
Checkpoint

1. What are the three possible values of the Completion Code?

2. How can an application be sure that no other applications are getting messages from a queue at the same time as you?

3. What is the difference in syntax between an MQCONN and an MQCONNX?

Notes:

Write your answers here:

1.

2.

3.
Unit Summary

- Must access queue manager and queues before use
  - MQCONN, MQCONNX
  - MQOPEN
- Release resources when work is complete
  - MQCLOSE
  - MQDISC

Notes:

Now that we have gone through the housekeeping calls and become familiar with some of the basics, we will proceed with the actual manipulation of data in a messaging and queuing environment.
Unit 3. Major Call - MQPUT

What This Unit Is About

Major Call - MQPUT call. The parameters and introduction of usage of the message descriptor and the put message option structure will be covered.

What You Should Be Able to Do

After completing this unit, you should be able to:

• MQPUT to place a message to a local queue

How You Will Check Your Progress

Accountability:

• Classroom discussion
• Machine exercise

References

SC34-6064  WebSphere MQ Application Programming Guide
SC34-6062  WebSphere MQ Application Programming Reference
Unit Objectives

- Learn how to use queues to transfer data
- Be able to code the WebSphere MQ application MQPUT call

Notes:

Although the objectives look pretty simple, if you master this topic, you will be well-established to place messages with an WebSphere MQ application.

We will cover the MQPUT call that allows messages to be placed on queues.
3.1 MQPUT
**Notes:**

The first of the major calls we will look at is MQPUT. This call is used to place messages onto a queue that has been opened for output.

As you can see, the number of parameters passed on this call is greater than the housekeeping calls. This allows for a great deal of flexibility.

Some of the parameters will be familiar to you:

- **Hconn** - the connection handle
- **Hobj** - the object handle
- **CompCode** - the completion code
- **Reason** - the reason code

These were all used in the housekeeping calls.

In the next few charts, we will learn about the remaining parameters as well as understand some of the reason codes that might occur when an application issues an MQPUT.
The Message Descriptor

- StrucId
- Version
- MsgType
- Encoding
- CodedCharSetld
- Format
- Priority
- Persistence
- MsgId
- Correlld

**Notes:**

Remember that each message has a message descriptor. This is where it is defined/created.

The message descriptor (MQMD) encapsulates the properties of the message. It can be updated by the application and/or by the queue manager. In many cases, an application will simply allow default values (taken from the queue definitions, for instance) to be supplied by the queue manager when the MQPUT is issued.

As with all structures, the MQMD starts with a structure identifier (StrucId). The value is MD. This is followed by a version number which can be 1, 2 or 3.

Regardless of the version, the layout of the MQMD is identical for the first 324 bytes. We will focus on these common fields. The above list contains some of the fields. The Application Programming Reference manual contains a complete layout of the MQMD structure as well as a full description of each field and its associated values.

Every message that is on a queue has a message descriptor. It is important to understand each field and what the values mean.
• **Message Type** is used to help WebSphere MQ and applications know what a particular message is used for. There are four types that are defined in the WebSphere MQ product:
  - Datagram (the default)
  - Request
  - Reply
  - Report

It is possible to have application-defined message types as well.

The next three fields listed are all related to data conversion which will be covered in depth later in this section.

• **Encoding** contains the representation of numeric data in the message.

• **CodedCharSetId** contains the International Standards Organization code page number associated with the message.

• **Format** (blank unless filled in by the application) is used to help the queue manager know what to do with data that requires conversion.

• **Priority** can be used to control the sequence in which messages will be gotten from a queue.

• **Persistence** determines whether the message will be logged to enable its recovery in the case of a restart of the queue manager or a rebuilding of a damaged queue.

• **MsgId** and **CorrelId** can be used to retrieve specific messages from a queue.

As you can see, several of the fields mentioned, although set when an MQPUT is issued, impact the delivery of the message.
Put Message Options (MQPMO)

- Strucid
- Version
- Options
- Timeout
- Context
- KnownDestCount
- UnknownDestCount
- InvalidDestCount
- ResolvedQName
- ResolvedQMgrName

Notes:
The Put Message Options (MQPMO) structure controls how a message is put on a queue. There are currently two versions of the MQPMO, with the Version 2 structure containing extra fields which relate to distribution list processing. Here, for simplicity, we will look at the fields that are common to both versions of the MQPMO structure.

- **Strucid**: An identifier (PMO)
- **Version**: (1 or 2) regardless of the version, the layout of MQPMO is identical for the first 128 characters (through the field called ResolvedQMgrName).
- **Options**: The most commonly used field. It can contain a value that is a combination of options that have been added together in the application. For example:
  - MQPMO_SYNCPOINT or MQPMO_NO_SYNCPOINT
  - MQPMO_FAIL_IFQUIESCING
  - MQPMO_PASS_IDENTITYCONTEXT
- **Timeout**: Reserved (not used).
• **Context:** Contains the object handle of an input queue. Its use will discussed in the Security unit.

**KnownDestCount, UnknownDestCount** and **InvalidDestCount:** These fields are reserved in Version 1 of the MQPMO. They are set by the distribution list support which uses Version 2 of the structure.

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

As we go through the following discussions, you will see how your application can use these fields to affect the way a message is put on a queue.
MQPUT Reason Codes

- MQRC_NONE
- MQRC_HCONN_ERROR
- MQRC_HOBJ_ERROR
- MQRC_MD_ERROR
- MQRC_MSG_TOO_BIG_FOR_Q
- MQRC_NOT_OPEN_FOR_OUTPUT
- MQRC_OPTIONS_ERROR
- MQRC_PMO_ERROR
- MQRC_PUT_INHIBITED
- MQRC_Q_FULL

Notes:
The completion codes resulting from the MQPUT, as with all calls can be MQCC_OK (0), MQCC_WARNING (1) or MQCC FAILED (2). There can be a variety of reason codes. If the completion code is MQCC_OK, then the reason code will always be MQRC_NONE (0).

Each of the reason codes that might possibly be returned from an MQPUT call is listed in the Application Programming Reference manual in its discussion of MQPUT. There is also a complete section in that same manual that reviews all reason codes and includes descriptions of the various codes along with recommended actions.

The symbolic names listed are defined and supplied in a copybook, header, include or other type library file that is used in each particular supported language.
MQPUT Pseudocode (1 of 3)

<table>
<thead>
<tr>
<th>Line</th>
<th>Statement</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>DEFINE BLANKS AS CONSTANT ' '</td>
</tr>
<tr>
<td>2</td>
<td>DEFINE QMGR AS MQCHAR48</td>
</tr>
<tr>
<td>3</td>
<td>DEFINE CONN_HANDLE AS MQHCONN</td>
</tr>
<tr>
<td>4</td>
<td>DEFINE COMP_CODE AS MQLONG</td>
</tr>
<tr>
<td>5</td>
<td>DEFINE REASON_CODE AS MQLONG</td>
</tr>
<tr>
<td>6</td>
<td>DEFINE OBJ_DESC AS MQOD</td>
</tr>
<tr>
<td>7</td>
<td>DEFINE OPEN_OPTIONS AS MQLONG</td>
</tr>
<tr>
<td>8</td>
<td>DEFINE MYQUEUE_HANDLE AS MQHOBJ</td>
</tr>
<tr>
<td>9</td>
<td>DEFINE MESSAGE_DESCRIPTOR AS MQMD</td>
</tr>
<tr>
<td>10</td>
<td>DEFINE PUT_OPTIONS AS MQMO</td>
</tr>
<tr>
<td>11</td>
<td>DEFINE MESSAGE_LENGTH AS MQLONG</td>
</tr>
<tr>
<td>12</td>
<td>DEFINE MESSAGE AS CHAR100</td>
</tr>
<tr>
<td>13</td>
<td>CALL MQCONN(QMGR,</td>
</tr>
<tr>
<td>14</td>
<td>CONN_HANDLE,</td>
</tr>
<tr>
<td>15</td>
<td>COMP_CODE,</td>
</tr>
<tr>
<td>16</td>
<td>REASON_CODE)</td>
</tr>
<tr>
<td>17</td>
<td>CALL MQOPEN(CONN_HANDLE,</td>
</tr>
<tr>
<td>18</td>
<td>OBJ_DESC,</td>
</tr>
<tr>
<td>19</td>
<td>OPEN_OPTIONS,</td>
</tr>
<tr>
<td>20</td>
<td>MYQUEUE_HANDLE,</td>
</tr>
<tr>
<td>21</td>
<td>COMP_CODE,</td>
</tr>
<tr>
<td>22</td>
<td>REASON_CODE)</td>
</tr>
</tbody>
</table>

Notes:
MQPUT Pseudocode (2 of 3)

<table>
<thead>
<tr>
<th>Line</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>26</td>
<td>MESSAGE_DESCRIPTOR.StrucId = MQMD_STRUC_ID</td>
</tr>
<tr>
<td>27</td>
<td>MESSAGE_DESCRIPTOR.Version = MQMD_VERSION_1</td>
</tr>
<tr>
<td>28</td>
<td>MESSAGE_DESCRIPTOR.Report = MQRO_NONE</td>
</tr>
<tr>
<td>29</td>
<td>MESSAGE_DESCRIPTOR.MsgType = MQMT_DATAGRAM</td>
</tr>
<tr>
<td>30</td>
<td>MESSAGE_DESCRIPTOR.Expiry = MQEI_UNLIMITED</td>
</tr>
<tr>
<td>31</td>
<td>MESSAGE_DESCRIPTOR.Feedback = MQFB_NONE</td>
</tr>
<tr>
<td>32</td>
<td>MESSAGE_DESCRIPTOR.Encoding = MQENC_NATIVE</td>
</tr>
<tr>
<td>33</td>
<td>MESSAGE_DESCRIPTOR.CodedCharSetId = MQCCSI_Q_MGR</td>
</tr>
<tr>
<td>34</td>
<td>MESSAGE_DESCRIPTOR.Format = &quot;STKREQ&quot;</td>
</tr>
<tr>
<td>35</td>
<td>MESSAGE_DESCRIPTOR.Priority = MQPRI_PRIORITY_AS_Q_DEF</td>
</tr>
<tr>
<td>36</td>
<td>MESSAGE_DESCRIPTOR.Persistence = MQPER_PERSISTENCE_AS_Q_DEF</td>
</tr>
<tr>
<td>37</td>
<td>MESSAGE_DESCRIPTOR.MsgId = MQMI_NONE</td>
</tr>
<tr>
<td>38</td>
<td>MESSAGE_DESCRIPTOR.CorrelId = MQCI_NONE</td>
</tr>
<tr>
<td>39</td>
<td>MESSAGE_DESCRIPTOR.ReplyToQ = BLANKS</td>
</tr>
<tr>
<td>40</td>
<td>MESSAGE_DESCRIPTOR.ReplyToQMgr = BLANKS</td>
</tr>
</tbody>
</table>

Figure 3-7. MQPUT Pseudocode (2 of 3)

Notes:
You may wish to look up what the initial values for some of the above MQMD fields are in the Application Reference manual.
### MQPUT Pseudocode (3 of 3)

<table>
<thead>
<tr>
<th>Line</th>
<th>Pseudocode</th>
</tr>
</thead>
<tbody>
<tr>
<td>42</td>
<td>PUT_OPTIONS.StrucId = MQPMO_STRUC_ID</td>
</tr>
<tr>
<td>43</td>
<td>PUT_OPTIONS.Version = MQPMO_VERSION_1</td>
</tr>
<tr>
<td>44</td>
<td>PUT_OPTIONS.Options = MQPMO_NO_SYNCPOINT</td>
</tr>
<tr>
<td>45</td>
<td>MESSAGE_LENGTH = 11</td>
</tr>
<tr>
<td>46</td>
<td>MESSAGE = &quot;Hello World&quot;</td>
</tr>
<tr>
<td>47</td>
<td>CALL MQPUT(CONN_HANDLE,</td>
</tr>
<tr>
<td>48</td>
<td>MYQUEUE_HANDLE,</td>
</tr>
<tr>
<td>49</td>
<td>MESSAGE_DESCRIPTOR,</td>
</tr>
<tr>
<td>50</td>
<td>PUT_OPTIONS,</td>
</tr>
<tr>
<td>51</td>
<td>MESSAGE_LENGTH,</td>
</tr>
<tr>
<td>52</td>
<td>MESSAGE,</td>
</tr>
<tr>
<td>53</td>
<td>COMP_CODE,</td>
</tr>
<tr>
<td>54</td>
<td>REASON_CODE)</td>
</tr>
<tr>
<td>55</td>
<td>IF COMP_CODE NOT = MQCC_OK</td>
</tr>
<tr>
<td>56</td>
<td>write an error message detailing the REASON_CODE</td>
</tr>
<tr>
<td>57</td>
<td>terminate the program</td>
</tr>
<tr>
<td>58</td>
<td>STOP</td>
</tr>
<tr>
<td>59</td>
<td>END_IF</td>
</tr>
</tbody>
</table>

**Notes:**

Notice you filled in the Message Descriptor fields (lines 26-40) to add values to fill in the PUT_OPTIONS.Options field (line 44). But, you used the MESSAGE_DESCRIPTOR and PUT_OPTIONS structure in the MQPUT call (lines 52 and 53).
**Checkpoint**

1. Which PUT option forces the call to return if the queue manager is being stopped?

2. The MQPUT parameter that references the Message Descriptor - is it a field or a structure?

3. The MQPUT parameter that references the Put message options - is it a field or a structure?

**Notes:**

Write your answer here:

1. 
2. 
3. 
Unit Summary

- Use MQPUT to place messages on a WebSphere MQ queue
- Specify parameters to control behavior of the calls

Notes:
The MQI major calls will be the main focus of work you do with the MQI.

Remember with the MQPUT, many messages are to be placed on a queue after an independent MQOPEN. The application knows that many messages are going to the same pre-opened queue.
Exercise 2

Do Lab Exercise 2

Notes:
Unit 4. Opening Queues and Message Descriptor

What This Unit Is About

This unit starts by revising and extending the concept of remote queueing from an application perspective. It then explores the four message types, placing particular emphasis on requests, replies and reports and concludes with a look at the programmer's responsibilities with respect to data conversion. Then you will learn how to use MQMD fields.

What You Should Be Able to Do

After completing this unit, you should be able to:

- Understand Predefined and Dynamic queues
- Understand remote queues
- Understand reply to queues
- Discuss MQMD fields in detail

How You Will Check Your Progress

Accountability:

- Classroom discussion

References

SC34-6064  WebSphere MQ Application Programming Guide
SC34-6062  WebSphere MQ Application Programming Reference
Unit Objectives

- Describe the process of sending data to a remote queue
- Request appropriate reports when creating a message
- Understand the role of the Dead Letter Queue
- Write a program that can operate in a remote or local environment
- Discuss MQMD fields

Notes:

One of the strengths of WebSphere MQ is the ability to allow a program to be written so it will operate transparently in a mixed platform environment. Although the program need not know where it is running, it is important for programmers to understand how to write a program to operate in this fashion.

The notion of asynchronous processing enables an application to do its work, regardless of network availability. However, this mode of operation requires some planning and design if it is to be successful.
4.1 Opening Queues
Queue Independence

Notes:

There are four types of queue which an application may open:

- Local queues
- Remote queues (local definitions of remote queues; cannot be opened for input)
- Alias queues
- Model queues (used to create queues dynamically)

With the exception of model queues, the syntax of the MQOPEN call is the same, irrespective of the type of queue being opened. For output queues, this allows the application architect or administrator to move a queue from one queue manager to another without having to rewrite the program.

Name resolution is a queue manager responsibility.
Notes:

An ALIAS QUEUE is simply a definition. It allows a local or remote definition to be referred to by another name. An alias queue can have some different properties from the underlying queue it is pointing to. For instance:

- DEFINE QLOCAL(REALQ) GET(ENABLED) PUT(ENABLED)
- DEFINE QALIAS(MYNAME) TARGQ(REALQ) GET(DISABLED)

allows the programs accessing the queue called REALQ to GET and PUT messages. However, if a program opens the queue called MYNAME, only PUTs will be permitted. GETs are disabled - even though the same queue (REALQ) is actually being used by both programs.

It is important to note that the program thinks that the queue called MYNAME is a real queue, not simply a pointer to another queue.
Queue Name Resolution

MQOPEN . . . .ObjectName, ObjectQMgrName

<table>
<thead>
<tr>
<th>Step</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>No</td>
<td>Find an XMITQ with the same name as the queue manager specified. MQPUT messages will be placed on this XMITQ.</td>
</tr>
<tr>
<td>Yes</td>
<td>Local queue on the local queue manager.</td>
</tr>
<tr>
<td>No</td>
<td>Create queue from model definition and generate name. Queue manager is local.</td>
</tr>
<tr>
<td>Yes</td>
<td>Find target queue name on the local queue manager. It could be a local queue or a remote queue.</td>
</tr>
<tr>
<td>Yes</td>
<td>Resolve queue name, queue manager name and XMITQ from remote queue definition.</td>
</tr>
<tr>
<td>No</td>
<td>MQRC_UNKNOWN_OBJECT_NAME</td>
</tr>
</tbody>
</table>

**Notes:**

The chart indicates the steps followed by the queue manager for name resolution.

When a program opens an object, the Object Descriptor contains the object name and optionally the name of the queue manager.

In most cases, the ObjectQueueManagerName field in the Object Descriptor will contain blanks. If it does, or if it contains the name of the queue manager to which the application is connected, local definitions are searched to resolve the queue name.

If the queue manager name field contains the name of another queue manager, it is assumed that this queue manager is remote, and a transmission queue with the same name is sought in which MQPUT messages may be placed.
Model Queue

Notes:

When an administrator defines a model queue, the definition is simply a template. When the name of a model queue is specified in the Object Descriptor for an MQOPEN, a queue with the attributes of the model is dynamically created. The model itself has no other purpose. If the characteristics of the new queue were displayed, it would appear as a local queue.

Temporary Dynamic Queues last only until the execution of the program that created it ends (normally or abnormally) or until the creating program closes it. There is no way to keep a temporary dynamic queue beyond that point.

Temporary dynamic queues may not hold persistent messages.

Permanent Dynamic Queues are created in exactly the same way but they are not automatically deleted. They must be explicitly deleted (using a close option of delete or by the administrator with a delete command). Once created, there is nothing specific that WebSphere MQ does to keep track of dynamically created permanent dynamic queues.

Which type of dynamic queue is chosen is a matter of application design.
Dynamic Queue Names

Notes:

The Dynamic Queue Name field in the Object Descriptor is used to control the name of the created dynamic queue. When an asterisk appears in the last position, it is replaced in the generated name by a string which is guaranteed unique for the local queue manager. There are a number of choices:

- The default value (CSQ.* for z/OS, AMQ.* for other environments).
- A name, such as MYQUEUE, with no trailing *. Such a name may not be unique.
- A name, such as MYQUEUE.*. Queues whose names start with the common character string can be grouped for administrative and security purposes.
- An * alone (not recommended).

After the MQOPEN completes successfully, the name of the new queue is available in the Object Name field.
Dynamic Reply-To Queues Pseudocode

```java
DEFINE DYN_OBJ AS MQOD
DEFINE A_MQMD AS MQMD
CALL MQCONN(......,
DYN_OBJ.DynamicQName = 'MYQUEUE.**'
DYN_OBJ.ObjectName = 'MODEL1'
CALL MQOPEN(CONN_HANDLE,
    DYN_OBJ,
    OPEN_OPTIONS,
    MYQUEUE_HANDLE,
    COMP_CODE
    REASON_CODE)
IF COMP_CODE NOT = MQCC_OK
    /* error handling */
ELSE
    /* save name assigned to created dynamic queue for later use */
    /* typically in the MQMD reply to queue name field */
    A_MQMD.ReplyToQ = DYN_OBJ.ObjectName
END-IF
```

Figure 4-7. Dynamic Reply-To Queues Pseudocode

**Notes:**
4.2 The Elements of Remote Queuing
Notes:

When an application issues an MQOPEN and an MQPUT to a remote queue, this is what happens:

1. At MQOPEN, the queue manager performs queue name resolution for Q2, and notes that it is a remote queue, owned by QM2 and that messages for Q2 are to be stored for forwarding on transmission queue QM2. (It is an administrative convention for transmission queues to be given the same name as the queue manager to which they point.) The definition of the remote queue is made by the administrator, independently of the application. It is sometimes referred to as a local definition of a remote queue.

2. When the MQPUT is issued, the queue manager places the message on the transmission queue specified in the administrator’s definition of Q2. The application is not directly aware of this, and applications should never put messages explicitly on a transmission queue.
3. Messages wait on the transmission queue until the channel is started. The channel consists of two communicating WebSphere MQ components called Message Channel Agents (MCAs) and the underlying network connection.

4. When the message arrives at the target queue manager, the receiving MCA attempts to put the message on the target queue, Q2 in this case. If for any reason this is not possible, the message may be placed in the target queue manager’s Dead Letter Queue, depending on properties of the message established when it was created by the putting application.

The syntax of the remote put is exactly the same as the syntax of a local put. However, a reason code MQRC_NONE informs the application merely that the message has been placed safely on the appropriate transmission queue. If the application creating the message wishes to be informed of subsequent non-delivery of the message, which might occur when QM1 is no longer active, it must make use of report messages, which we will discuss later in this unit.
Remote Queuing (2 of 2)

Notes:

The destination queue (Q3 in the example above) doesn’t even have to be owned by an adjacent queue manager. Here, definitions made by the administrator at QM2 cause messages intended for QM3 to be forwarded via a suitable transmission queue. This configuration is known as multihopping.

The topology is more complicated, but it makes no difference to how the MQPUT is coded in the application running connected to QM1, although it might be wise to allow for a longer delay for responses.
The Dead Letter Queue

**Notes:**

All queue managers have the ability to place undeliverable messages on a local queue which has been assigned the role of the Dead Letter Queue for the queue manager.

When a message arrives from a remote queue manager and is not deliverable, the receiving queue manager inspects the Report field of the MQMD. An option specified by the creator of the message determines whether the undeliverable message is to be placed on the DLQ (MQRO_DEAD_LETTER_QUEUE, the default value) or discarded (MQRO_DISCARD_MSG).

If the message creator wishes to be notified of non-delivery, a further Report Option must be specified: this may be MQRO_EXCEPTION (report message consisting only of an MQMD), MQRO_WITH_DATA (MQMD + first 100 bytes of original application data) or MQRO_WITH_FULL_DATA + complete original application data). If one of these report options is specified, the queue manager builds a report message, and returns it to the ReplyToQ of the originating application. The Feedback field in the report message's MQMD identifies the reason for non-delivery.
## Undelivered Message

- **Reason**  
  - Why message was not delivered

- **DestQName, DestQMgrName**  
  - Where message was meant to go

- **PutApplType, PutApplName**  
  - Application that put the message on the Dead Letter Queue

- **PutDate, PutTime**  
  - When message was put on the Dead Letter Queue

- **Format, Encoding, CodedCharSetId**  
  - Format, encoding, codeset of the following application data

### Notes:

Messages placed on the dead letter queue contain a header, the dead letter header (MQDLH). The MQDLH contains the reason why the message was undeliverable, the name of the destination queue and destination queue manager as well as some other fields.

The original message and its complete message descriptor are not altered. If an application is authorized to put messages on the queue that has been identified as the dead letter queue, it must properly build a MQDLH and add it to the beginning of any messages that it places on the queue.

The format of the MQDLH is documented in *Application Programming Reference.*
Uses of the DLQ

- WebSphere MQ puts message on the DLQ:
  - MCA, if remote message can't be delivered
  - MCA, if message to be sent cannot be converted
  - Trigger monitor, if trigger message fails

Notes:

As we have already seen, the queue manager will put an message on the dead letter queue if a receiving Message Channel Agent is unable to deliver the message to its destination queue. But there are other instances in which a message will be placed on the dead letter queue.

If the trigger monitor is unable to start the triggered program for any reason, the trigger message from the initiation queue will be placed in the dead letter queue. And if a sending Message Channel Agent is defined to perform data conversion (see later in this unit), and the MCA is unable to convert a particular message, that message will be placed on the dead letter queue on the sending queue manager.

It is possible for a application program to place a message on the dead letter queue. If you choose to do so, make sure that you build an appropriate Dead Letter Header (MQDLH). However, due of the potential problems caused by a full dead letter queue, a better solution may be to create application dead letter queues.
4.3 MQMD Fields
Notes:
The persistence attribute of a message determines whether a message is recoverable after queue manager restarts and rebuilds of damaged queues.

Persistent messages are logged to enable recovery, and this logging has a performance impact. An example might be an instruction to buy a number of shares.

Non-persistent messages are not logged, and will not survive queue manager restarts or rebuilds of damaged queues. An example might be an inquiry on the price of a stock.

The MQMD Persistence field can be set by the application program:

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

When messages are put on a queue with persistence, at restart time, the queue manager will recover the messages by replaying its logs. At the same time, ALL messages that were put as non-persistent will be explicitly deleted at restart.

Therefore, if a message is critical and there is no simple way to recreate it, either the programmer should explicitly set the message descriptor persistence (MQMD_Persistence) to a value of 1 or, if the queue has been defined with a DEFPSIST(YES), the program can safely allow the default of persistence as defined for the queue to be used.
Priority

PUT applications message arrive randomly.

Notes:

The priority field in the Message Descriptor: Using this field, you can set the priority of a message when you put it on a queue. This value may be in the range 0 - 9, or -1, which causes the default priority attribute of the queue to be inherited.

The MsgDeliverySequence attribute of the queue: This attribute, set by the administrator, determines how the message is actually 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 set by the application. If this attribute is MQMDS_PRIORITY, the priority value in the Message Descriptor is honored when enqueuing the message.

A convention when constructing replies to use the priority value of the incoming request.
**Message Types**

**Datagram**

- Application → Datagram → Queue Manager → Datagram → Application
- send → receive

**Request/Reply**

- Application → Request → Queue Manager → Request → Application
- send request → receive reply
- Reply

- Application → Request → Queue Manager → Request → Application
- send request → receive reply

**Report**

- Application → Message → Queue Manager → XMITQ → Report → Queue Manager → DLQ
- send message

---

**Notes:**

WebSphere MQ provides four predefined messages types, identified by the Message Type field in the Message Descriptor.

- **Requests** When an application creates a message expecting a response from a target application, the message type should be set to request.

- **Replies** These are application specific responses from a target application.

- **Reports** Most frequently generated by a target queue manager as a result of a specific condition, such as non-delivery, occurring.

- **Datagrams** The default message type, expecting no response.

Applications can also define their own message types within the range of MQMT_APPL_FIRST and MQMT_APPL_LAST.
Notes:

In this example, the requesting application creates a message with a message type of request and specifies a local queue, Q2, as the reply to queue. The MQPUT will fail if the message type is request and no ReplyToQ is specified.

The responding application retrieves a message from a queue and MQPUT1s a reply to the reply to queue at the reply to queue manager in the incoming Message Descriptor.

Either the requesting application or another application can monitor Q2 for replies, depending on the application design. If another application is getting from the reply queue, the reply to queue is likely to be triggered.
Specifying a Remote Reply-to Queue

**Notes:**

Here, the ReplyToQ specified is a remote queue. If the ReplyToQMgr is left blank, as in this example, the local queue manager will attempt to resolve the name of the ReplyToQ and, if unable to, the call will fail. By not naming the queue manager, the application tells the local queue manager to search local definitions, and a local definition of the remote queue Q2 owned by queue manager QM2 is used.

The application processing the replies must be on QM2.
Notes:

It is possible for an application to explicitly set both the queue name and the queue manager name in the object descriptor prior to an MQPUT. These values could be hardcoded, but this example shows the most common situation where this technique is used. Both fields are initialized with values taken from the incoming Message Descriptor.

A subsequent MQOPEN and MQPUT, or an MQPUT1, will send a reply to the reply queue at the reply queue manager specified in the incoming request’s Message Descriptor. Because the queue manager name has been explicitly specified, any queue definitions held locally are ignored, and the queue manager will look for a transmission queue with the same name as the ObjectQMgrName (if it is not the local queue manager’s name), build a transmission header containing the addressing information and place the message on that transmission queue.

The reply queue thus accessed has often been dynamically created in the requesting application, in which case a definition for a predefined queue could not exist anyway.
Report Field Options

- Report requests with feedback generated by queue manager
  - MQRO_EXCEPTION
  - MQRO_EXPIRATION
  - MQRO_COA (confirm on arrival)
  - MQRO_COD (confirm on delivery)

- Report request with feedback generated by an application program
  - MQRO_PAN (positive acknowledgment)
  - MQRO_NAN (negative acknowledgment)

- Use of MQRO_ requires the reply-to-queue field to be filled

Notes:

- Exception report messages were described above.
- Expiry report messages indicate that an application attempted to retrieve a message that had reached its expiry threshold and that the message has been discarded.
- Confirmation of arrival (COA) report messages indicate that the message has reached its target queue.
- Confirmation of delivery (COD) report messages indicate that the message has been retrieved by a receiving application.
- Positive action notification (PAN) report message imply that a request has been successfully serviced.
- Negative action notification (NAN) report messages imply that a request has not been successfully serviced.

Report options may be combined: for example, if you set the delivery confirmation report option and one of the exception report message options, if the message fails to be delivered, you will receive an exception report message. However, if you select only the
delivery confirmation report message option and the message fails to be delivered, you will not get an exception report message.

For each report field option, the application creating the original message may choose to have none of the original data returned, the first 100 bytes returned or the complete application data.

The report option can even dictate the message actions as how to handle MSGID, CORRELID, or possible DLQ/DISCARD actions.

**Application Creation of Report Messages**

PAN and NAN messages are intended to be created by applications. How these reports are interpreted are part of the application design.

Report options ask for reports. Feedback provides the information as shown in the next graphic.

The corresponding report message type is clarified in the feedback field of the report messages MGMD.
Exception Reports

Notes:

Problems with Local Queues

When there is a problem with a local queue (queue full, does not exist, and so forth), the queue manager can notify the application immediately and synchronously by means of a Completion Code and Reason Code.

Problems with Remote Queues

With remote queues, a satisfactory CC/RC means merely that the message has been placed on the xmitq. If the receiving queue manager subsequently encounters a problem, the CC/RC mechanism can not be used: Time Independence means that the sending program may no longer be active. In this case, if the MQMD Report Options field has asked for an exception report in the case of non-delivery, the target queue manager will construct an exception report message and return it to the ReplytoQ specified in the MQMD.

Exception report options are MQRO_EXCEPTION, MQRO_EXCEPTION_WITH_DATA and MQRO_EXCEPTION_WITH_FULL_DATA.
Feedback Field Constant Values

Feedback:

- MQFB_NONE
- MQFB_EXPIRATION
- MQFB_COA
- MQFB_COD
- MQFB_PAN
- MQFB_NAN
- MQFB_QUIT
- MQFB_APPL_FIRST
- MQFB_APPL_LAST

Reason code:

- MQRC_PUT_INHIBITED
- MQRC_QUEUE_FULL
- MQRC_NOT_AUTHORIZED

Notes:

When sending a report message type, the feedback field Reports why.
Encoding

When floating point, decimal or integer numeric data is put into fields in the application data portion of a message, there is a problem that the data may be represented differently on different platforms.

In the above example, the PUTting platform is UNIX (that is, AIX) and the GETting environment is Intel (that is, Windows NT). Because of platform differences, a number created as 3 in UNIX will be interpreted as 768 on Windows NT.

This problem is avoided by the Encoding field in the MQMD. Using the default value, MQENC_NATIVE, will enable the queue manager to put the correct value into the MQMD_Encoding field based on the platform where the program is running as well as the language that the program is written in. The destination queue manager can inspect the Encoding field in the incoming MQMD and, if requested, convert the numeric fields as appropriate.
Notes:

Data can come in many languages as well as in ASCII or EBCDIC. So, when a message is put on a queue, it is important to identify what the coded character set is.

In our example, we may be going from US English EBCDIC (MVS/ESA) to a UK English ASCII system. The receiving system needs to know that the data 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 proper value to be supplied by the queue manager at MQPUT time. The actual value will be the International Standards Organization code page number for that particular language/platform.

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

- **Some formats are built-in**
  - Text strings
  - Message structures defined by WebSphere MQ
    - Trigger message
    - Event message
  - Conversion done automatically

- **Message Conversion Exit**
  - Format is defined by application, not WebSphere MQ
  - Message with built-in format fails to convert

**Notes:**

The Encoding and CodedCharSetId fields of an incoming message enable the receiving queue manager to convert the MQMD - the queue manager knows the location of the numeric fields, such as MsgType, and the character fields, such as ReplyToQ.

However, the receiving queue manager can have no idea of how the application data portion of the message has been designed; it sees only a stream of bytes.

If we require the application data portion of the message to be created as well, we have three options:

- Create a message where in the application data is all in character format, including numeric fields. If you set the Format field in the MQMD to MQFMT_STRING, application data will be converted on a character by character basis.
- Write a message conversion exit, and place its name in the Format field.
- Use an additional product such as WebSphere Business Integration Message Broker to perform data conversion.
What Applications Should Do

- Put messages with the following values in the **Encoding** and **CodedCharSetId** fields
  - MQENC_NATIVE - For native encoding
  - MQCCSI_Q_MGR - For the same CCSID as the queue manager

- Put all messages with a format name
  - MQFMT_STRING - For a message consisting entirely of characters

- Use the MQGMO_CONVERT option on the MQGET call
  - Check what is delivered by the call

- If necessary, use CONVERT(YES) at the sending end of a message channel

**Notes:**

The three MQMD fields related to data conversion enable any receiving queue manager to determine the data creation characteristics of the environment on which the message originated, and convert as appropriate.

There is no significant overhead in specifying the MQGMO_CONVERT option even when no conversion is required. It is worth doing this to allow for flexibility in the location of the queue.

Specifying a channel to convert at the sending end is necessary when the receiving queue manager is a back level one which does not support conversion.
Checkpoint

1. What are the four predefined Message Types?

2. If a message sent to a remote queue can't be delivered to the remote queue, what happens to it by default?

3. Name four report options.

4. Which three fields in the MQMD relate to data conversion?

5. What is the difference in syntax between an MQOPEN for a local queue and an MQOPEN for a remote queue?

Notes:

Please write your answers here.

1.

2.

3.

4.

5.
Unit Summary

- Transparent to application where queue is located
- Use replies and reports to check status
- Need to consider different data representations

Notes:

Applications can run transparently if properly designed. Programmers need to understand what platforms the program will run on, what the data within the message looks like, whether reports about the message and its progress are required, and what business replies are needed.
Unit 5. Major Calls - MQGET and MQPUT1

What This Unit Is About

Major Calls - Messaging and Queuing includes the MQGET and MQPUT1 calls.

What You Should Be Able to Do

After completing this unit, you should be able to:

• **MQGET** to remove a message from a local queue
• **MQPUT1** to put one message on a local queue

How You Will Check Your Progress

Accountability:

• Classroom discussion
• Machine exercise

References

SC34-6064  *WebSphere MQ Application Programming Guide*
SC34-6062  *WebSphere MQ Application Programming Reference*
Unit Objectives

- Learn how to use queues to transfer data
- Write simple WebSphere MQ applications using the major calls

Notes:

Although the objectives look pretty simple, if you master this topic, you will be well-established to deal with WebSphere MQ application development.

We will cover MQGET and MQPUT1 major calls that allow messages to be placed on and retrieved from queues.
5.1 MQGET
Notes:

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.

Again, the connection handle and object handle are the first two arguments passed.

The message descriptor is again both an input and output structure, as is the Get Message Options structure.

In the case of MQGET, the buffer is an output field. BufferLength tells the queue manager how long the buffer is into which the application data portion of the message is to be returned.

The queue manager uses the DataLength to return the actual length of the message data that was retrieved. In this way, the program can make use of a general purpose buffer that can contain small or large messages. It can know how big each message retrieved actually is by the DataLength value.
Get Message Options (MQGMO)

- StruclD
- Version
- Options
- WaitInterval
- Signal1
- Signal2
- ResolvedQName
- MatchOptions

Notes:

Like the MQPMO structure for Put Message Options, there is a MQGMO structure that allows for definition of options to be used when the MQGET is issued.

Also, as with all the structures in WebSphere MQ, MQGMO begins with an identifier ("GMO "), followed by a version number. There are currently three versions of the MQGMO. Version 1 is supported on all platforms; Version 2 contains information related to Message Groups and Message Segments, and Version 3 contains information on Message Tokens, supported only on z/OS. In this unit, we will focus on Version 1, as shown above.
**The Options field** contains the combination of options that an application specifies to control the behavior of the MQGET. Some of the more widely used options are:

- **MQGMO_WAIT** - wait until a message arrives on the queue.

  This is generally used in conjunction with the field in the MQGMO structure called WaitInterval, a value represented in milliseconds that specifies how long the wait should be for a message to arrive before issuing a failing return code.

- **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 will fit and discard the remainder.

  If not specified and a message is larger than the buffer length as declared in the MQGET, the call will fail.

- **MQGMO_CONVERT** - if the MQMD_ENCODING or the MQMD_CCSID on the incoming message differ from those of the environment where the MQGET is issued, use the value in the MQMD_FORMAT field 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 because, if an application issues an MQGET on an alias queue, the name of the base queue will be returned in the MQGMO_ResolvedQName field when the queue manager retrieves the message.

The MatchOptions allow the application to choose which fields in the MsgDesc parameter will be used to select the message returned by the MQGET call. The application sets the required options in this field, and then sets the corresponding fields in the MsgDesc parameter to the values required for those fields. Only messages that have those values in the MQMD for the message are candidates for retrieval using that MsgDesc parameter on the MQGET call.
**Buffer Length (1 of 2)**

--

**Figure 5-4. Buffer Length (1 of 2)**

**Notes:**

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

It is safest to ensure that the buffer length and buffer agree in the length. But, 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 will be placed into the buffer. It is important to remember that there data remaining from a previous get beyond the current buffer length.

If a message may be larger than the buffer length specified, it is possible to retrieve that portion of the message that will fit 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 is not specified, the MQGET will fail if a message larger than the buffer is encountered. However, the buffer will contain the portion of the message that will fit 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.
Notes:

In the visual above, numbers 1 through 4 represent the sequence of events.

1. Task 1 attempts to get the first message (Length 100). As it specified a buffer length of only 50, the MQGET fails.

2. Using the value returned by the queue manager in the data length parameter, Task 1 acquires a buffer of 100, using whatever mechanism the programming language provides.

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 once 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

- MQRC_NONE
- MQRC_CONNECTION_BROKEN
- MQRC_GET_INHIBITED
- MQRC_GMO_ERROR
- MQRC_HCONN_ERROR
- MQRC_HOBJ_ERROR
- MQRC_MD_ERROR
- MQRC_NO_MSG_AVAILABLE
- MQRC_NOT_CONVERTED
- MQRC_NOT_OPEN_FOR_INPUT
- MQRC_OPTIONS_ERROR
- MQRC_TRUNCATED_MSG_ACCEPTED
- MQRC_TRUNCATED_MSG_FAILED

Notes:
The list shown is a subset of many possible reasons that can be returned after the execution of an MQGET. A complete list of the reasons is contained in the Application Programming Reference manual in the section on MQGET within the chapter that describes each of the MQI calls.

MQRC_NONE will only be returned if the completion code is MQCC_OK (0). The remaining reasons could be a result of a completion code of MQCC_WARNING (1) or MQCC_FAILED (2).

- MQRC_CONNECTION_BROKEN - Connection to the queue manager has been lost
- MQRC_GET_INHIBITED - Gets have been disabled for the queue
- MQRC_GMO_ERROR - Get Message Options structure is not valid
- MQRC_HCONN_ERROR - Connection handle not valid
- MQRC_HOBJ_ERROR - Object handle not valid
- MQRC_MD_ERROR - Message descriptor not valid
- MQRC_NO_MSG_AVAILABLE - No message available
• MQRC_NOT_CONVERTED - Application message data not converted even though MQGMO_CONVERT option specified
• MQRC_NOT_OPEN_FOR_INPUT - Attempting to retrieve a message from a queue that was not opened for input
• MQRC_OPTIONS_ERROR - Options not valid or consistent
• MQRC_TRUNCATED_MSG_ACCEPTED - Truncated message returned (processing completed)
• MQRC_TRUNCATED_MSG_FAILED - Truncated message returned (processing not completed)

One of the most common reasons is MQRC_NO_MSG_AVAILABLE. If an application receives this reason, it may be because there are no messages left on the queue to be gotten, a valid occurrence. However, the same reason could be a result of forgetting to set up the MQGET properly. We will see more about this in the unit that reviews MsgId and CorrelId.

In the case of MQRC_NOT_CONVERTED, the message is returned to the application as there is little point in leaving it on the queue. The completion code will be MQCC_WARNING.
### MQGET Pseudocode (1 of 3)

<table>
<thead>
<tr>
<th>Line</th>
<th>Pseudocode</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>DEFINE BLANKS AS CONSTANT ' '</td>
</tr>
<tr>
<td>2</td>
<td>DEFINE CONN_HANDLE AS MQHCONN</td>
</tr>
<tr>
<td>3</td>
<td>DEFINE COMP_CODE AS MQLONG</td>
</tr>
<tr>
<td>4</td>
<td>DEFINE REASON_CODE AS MQLONG</td>
</tr>
<tr>
<td>5</td>
<td>DEFINE MESSAGE_DESCRIPTOR AS MQMD</td>
</tr>
<tr>
<td>6</td>
<td>DEFINE GET_OPTIONS AS MQGMO</td>
</tr>
<tr>
<td>7</td>
<td>DEFINE MESSAGE_LENGTH AS MQLONG</td>
</tr>
<tr>
<td>8</td>
<td>DEFINE MESSAGE AS CHAR (100)</td>
</tr>
<tr>
<td>9</td>
<td>DEFINE LENGTH_OF_MESSAGE AS MQLONG</td>
</tr>
<tr>
<td>10</td>
<td>MYQUEUE_HANDLE AS MQHOBJ</td>
</tr>
<tr>
<td>11</td>
<td>CALL MQCONN(QMGR,</td>
</tr>
<tr>
<td>12</td>
<td>CONN_HANDLE,</td>
</tr>
<tr>
<td>13</td>
<td>COMP_CODE,</td>
</tr>
<tr>
<td>14</td>
<td>REASON_CODE)</td>
</tr>
<tr>
<td>15</td>
<td>CALL MQOPEN(CONN_HANDLE,</td>
</tr>
<tr>
<td>16</td>
<td>OBJ_DESC,</td>
</tr>
<tr>
<td>17</td>
<td>OPEN_OPTIONS,</td>
</tr>
<tr>
<td>18</td>
<td>MYQUEUE_HANDLE,</td>
</tr>
<tr>
<td>19</td>
<td>COMP_CODE,</td>
</tr>
<tr>
<td>20</td>
<td>REASON_CODE)</td>
</tr>
<tr>
<td>21</td>
<td></td>
</tr>
<tr>
<td>22</td>
<td></td>
</tr>
<tr>
<td>23</td>
<td></td>
</tr>
</tbody>
</table>

Figure 5-7. MQGET Pseudocode (1 of 3)

**Notes:**
### MQGET Pseudocode (2 of 3)

<table>
<thead>
<tr>
<th>Line Number</th>
<th>Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>24</td>
<td>MESSAGE_DESCRIPTOR.StrucId=MQMD_STRUC_ID</td>
</tr>
<tr>
<td>25</td>
<td>MESSAGE_DESCRIPTOR.Version=MQMD_VERSION_1</td>
</tr>
<tr>
<td>26</td>
<td>GET_OPTIONS.StrucId = MQGMO_STRUC_ID</td>
</tr>
<tr>
<td>27</td>
<td>GET_OPTIONS.Version = MQGMO_VERSION_1</td>
</tr>
<tr>
<td>28</td>
<td>GET_OPTIONS.Options = MQGMO_NO_WAIT + MQGMO_NO_SYNCPOINT</td>
</tr>
<tr>
<td>29</td>
<td>MESSAGE_LENGTH =100</td>
</tr>
<tr>
<td>30</td>
<td>DO</td>
</tr>
<tr>
<td>31</td>
<td>MESSAGE_DESCRIPTOR.MsgId = MQMI_NONE</td>
</tr>
<tr>
<td>32</td>
<td>MESSAGE_DESCRIPTOR.CorrelId = MQCI_NONE</td>
</tr>
<tr>
<td>33</td>
<td></td>
</tr>
<tr>
<td>34</td>
<td></td>
</tr>
<tr>
<td>35</td>
<td></td>
</tr>
<tr>
<td>36</td>
<td></td>
</tr>
<tr>
<td>37</td>
<td></td>
</tr>
</tbody>
</table>

**Notes:**
**MQGET Pseudocode (3 of 3)**

```sql
call MQGET(conn_handle, 38
  message_queue_handle, 39
  message_descriptor, 40
  get_options, 41
  message_length, 42
  message, 43
  length_of_message, 44
  comp_code, 45
  reason_code) 46

until reason_code not = MQRC_NO_MSG_AVAILABLE 48

if comp_code not = MQCC_OK 49
  write an error message detailing the reason_code 51
  terminate the program 52
  stop 53
end-if 54
```

**Notes:**
Sending Replies to the Correct Reply-to Queue

**Notes:**

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 invoked if the responding application always expects request type messages.

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

If the queue that is receiving the responses is not on the local queue manager, the ReplyToQMGR is more significant. The queue manager looks for a transmission queue with the name of the ReplyToQMGR and that is where the message will be placed (with the proper transmission header containing routing information) for onward transmission by the channel.
RePLYING PSEUDOCODE

```plaintext
DEFINE IN_MSG DESC AS MQMD
DEFINE OUT_MSG DESC AS MQMD

CALL MQGET(....,....,IN_MSG_DESC,...)

IF IN_MSG_DESC.MsgType = MQMT_REQUEST
   formulate reply
   OUT_MSG_DESC.MsgType = MQMT_REPLY
   OBJ_DESC.ObjectName = IN_MSG_DESC.ReplyToQ
   OBJ_DESC.ObjectQMgrName = IN_MSG_DESC.ReplyToQMgr
   CALL MQPUT1(....,OBJ_DESC,OUT_MSG_DESC,...)
END-IF
```

Figure 5-11. RePLYING PSEUDOCODE

Notes:
5.2 MQPUT1
Notes:

As with the MQPUT, the MQPUT1 is used to place messages on queues. With MQPUT1, however, no associated MQOPEN and MQCLOSE calls are required as these two 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 overhead of multiple MQOPEN and MQPUT calls.

The MQPUT1 parameters are a combination of those used for MQOPEN (Object Descriptor), and those used for the MQPUT (Message Descriptor and Put Message Options), but not including 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. Note that MQPUT1 can not be used with a model queue.
Notes:

In order to successfully execute the MQPUT1, the user must have proper access that will enable the queue to be opened for output.

As with the MQOPEN, it is possible that the call can fail with MQRC_NOT AUTHORIZED.
MQPUT1 Pseudocode (1 of 3)

```
DEFINE DEFAULT_QMGR AS CONSTANT ' '
DEFINE QMGR AS MQCHAR48
DEFINE CONN_HANDLE AS MQHCNN
DEFINE COMP_CODE AS MQLONG
DEFINE REASON_CODE AS MQLONG
DEFINE MYQUEUE_HANDLE AS MQHOBJ
DEFINE MESSAGE_DESCRIPTOR AS MQMD
DEFINE PUT_OPTIONS AS MQPMO
DEFINE OBJ_DESC AS MQOD
DEFINE MESSAGE_LENGTH AS MQLONG
DEFINE MESSAGE AS MQCHAR100

QMGR = DEFAULT_QMGR
CALL MQCONN(QMGR,
            CONN_HANDLE,
            COMP_CODE,
            REASON_CODE)

OBJ_DESC.StrucId = MQOD_STRUCT_ID
OBJ_DESC.Version = MQOD_VERSION_1
OBJ_DESC.ObjectType = MQOT_Q
OBJ_DESC.ObjectName = "MYQUEUE"
OBJ_DESC.ObjectQMgrName = DEFAULT_QMGR
```

Notes:
### MQPUT1 Pseudocode (2 of 3)

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>24</td>
<td>MESSAGE_DESCRIPTOR.Strucid = MQMD_STRUC_ID</td>
</tr>
<tr>
<td>25</td>
<td>MESSAGE_DESCRIPTOR.Version = MQMD_VERSION_1</td>
</tr>
<tr>
<td>26</td>
<td>MESSAGE_DESCRIPTOR.Report = MQRO_NONE</td>
</tr>
<tr>
<td>27</td>
<td>MESSAGE_DESCRIPTOR.MsgType = MQMT_DATAGRAM</td>
</tr>
<tr>
<td>28</td>
<td>MESSAGE_DESCRIPTOR.Expiry = MQEI_UNLIMITED</td>
</tr>
<tr>
<td>29</td>
<td>MESSAGE_DESCRIPTOR.Feedback = MQFB_NONE</td>
</tr>
<tr>
<td>30</td>
<td>MESSAGE_DESCRIPTOR.Encoding = MQENC_NATIVE</td>
</tr>
<tr>
<td>31</td>
<td>MESSAGE_DESCRIPTOR.CodedCharSetId = MQCCSI_Q_MGR</td>
</tr>
<tr>
<td>32</td>
<td>MESSAGE_DESCRIPTOR.Format = &quot;STKREQ&quot;</td>
</tr>
<tr>
<td>33</td>
<td>MESSAGE_DESCRIPTOR.Priority = MQPRI_PRIORITY_AS_Q_DEF</td>
</tr>
<tr>
<td>34</td>
<td>MESSAGE_DESCRIPTOR.Persistence = MQPER_PERSISTENCE_AS_Q_DEF</td>
</tr>
<tr>
<td>35</td>
<td>MESSAGE_DESCRIPTOR.MsgId = MQMI_NONE</td>
</tr>
<tr>
<td>36</td>
<td>MESSAGE_DESCRIPTOR.CorrelId = MQCI_NONE</td>
</tr>
<tr>
<td>37</td>
<td>MESSAGE_DESCRIPTOR.ReplyToQ = BLANKS</td>
</tr>
<tr>
<td>38</td>
<td>MESSAGE_DESCRIPTOR.ReplyToQMgr = BLANKS</td>
</tr>
<tr>
<td>39</td>
<td></td>
</tr>
</tbody>
</table>

**Notes:**

Figure 5-15. MQPUT1 Pseudocode (2 of 3)

SW3135.3
### MQPUT1 Pseudocode (3 of 3)

<table>
<thead>
<tr>
<th>Line</th>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>40</td>
<td>PUT_OPTIONS.StrucId = MQPMO_STRUC_ID</td>
<td></td>
</tr>
<tr>
<td>41</td>
<td>PUT_OPTIONS.Version = MQPMO_VERSION_1</td>
<td></td>
</tr>
<tr>
<td>42</td>
<td>PUT_OPTIONS.Options = MQPMO_NO_SYNCPOINT</td>
<td></td>
</tr>
<tr>
<td>43</td>
<td>MESSAGE_LENGTH = 11</td>
<td></td>
</tr>
<tr>
<td>44</td>
<td>MESSAGE = &quot;Hello World&quot;</td>
<td></td>
</tr>
<tr>
<td>45</td>
<td>CALL MQPUT1(CONN_HANDLE,</td>
<td></td>
</tr>
<tr>
<td>46</td>
<td>OBJ_DESC,</td>
<td></td>
</tr>
<tr>
<td>47</td>
<td>MESSAGE_DESCRIPTOR,</td>
<td></td>
</tr>
<tr>
<td>48</td>
<td>PUT_OPTIONS,</td>
<td></td>
</tr>
<tr>
<td>49</td>
<td>MESSAGE_LENGTH,</td>
<td></td>
</tr>
<tr>
<td>50</td>
<td>MESSAGE,</td>
<td></td>
</tr>
<tr>
<td>51</td>
<td>COMP_CODE,</td>
<td></td>
</tr>
<tr>
<td>52</td>
<td>REASON_CODE)</td>
<td></td>
</tr>
<tr>
<td>53</td>
<td>IF COMP_CODE NOT = MQCC_OK</td>
<td></td>
</tr>
<tr>
<td>54</td>
<td>write an error message detailing the REASON_CODE</td>
<td></td>
</tr>
<tr>
<td>55</td>
<td>terminate the program</td>
<td></td>
</tr>
<tr>
<td>56</td>
<td>STOP</td>
<td></td>
</tr>
<tr>
<td>57</td>
<td>END_IF</td>
<td></td>
</tr>
</tbody>
</table>

**Notes:**

Notice that line 49 is using the object descriptor and not the MQHOBJ.
Checkpoint

1. Which option forces a call to return if the queue manager is being stopped?

2. What is the default action of the queue manager if you try to get a message that is too large for the application buffer?

3. Which MQPUT parameter is different in an MQPUT1 call?

Notes:

Write your answers here:

1.

2.

3.
Unit Summary

- Use MQPUT or MQPUT1 to place messages on a WebSphere MQ queue
- Use MQGET to retrieve messages from a WebSphere MQ queue
- Specify parameters to control behavior of the calls

Notes:

The MQI major calls will be the main focus of work you do with the MQI.

Remember the difference between the MQPUT (multiple messages to be placed on a queue) and MQPUT1 (a single message to be placed on a queue). MQPUT1 saves an explicit MQOPEN and MQCLOSE.

MQGET is the only call used to retrieve messages from a queue. You can also use MQGET with browse options to look at messages without removing them from the queue.
Exercise 3

Do Lab Exercise 3

Notes:
Unit 6. Controlling Message Retrieval

What This Unit Is About

This unit shows how you can use the MsgId and CorrelId fields in the MQMD to retrieve messages selectively, and how to await message arrival.

What You Should Be Able to Do

After completing this unit, you should be able to:

• Describe the problem of correlating various replies to a single message
• Write a program which waits for the appearance of a particular message on a queue

How You Will Check Your Progress

Accountability:

• Classroom discussion
• Machine exercises

References

SC34-6064  WebSphere MQ Application Programming Guide
SC34-6062  WebSphere MQ Application Programming Reference
Unit Objectives

- Controlling message delivery with Message ID and Correlation ID
- Programming to allow time for responses to arrive

Notes:

Now that the basic WebSphere MQ MQI calls have been covered and you have some experience with them, we will look at some of the ways to control message delivery.

- Use of message descriptor fields MsgId and CorrelId
- Ways to allow time for messages to arrive

Keep in mind the asynchronous nature of WebSphere MQ as we proceed.
6.1 MsgId and CorrelId
Notes:

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

- Several copies of this application may be running simultaneously
- The ReplyToQ is the same for all requesting applications

The requesting application MQGETs replies sent by the server:

- It must ensure that the reply it gets relates to the request it has just sent, ignoring any others

If an application simply issues MQGETs, we know the messages are 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 additional MQGETs would be issued just to examine each message.

Two fields in the message descriptor allow an application to selectively retrieve messages. The MsgId and Correlld fields can be used in our scenario to ensure that the reply message retrieved is the one that matches the request sent.
Request/Reply Queue Consideration

Common Reply Queue

- One reply queue for all service requests
- Which reply correlates to which request?
- Which reply belongs to which application?

Temporary Reply Queue

- One temporary reply queue for every service request
- What about persistent messages?

Notes:

It is possible to use a separate ReplyToQ for each separate requesting application. These queues are usual dynamic queues for easier administration. When dynamic reply queues are used the use of MsgId and CorrelId still might be necessary to correlate the replies, if the requesting application sends several request before receiving the replies.
**MsgId, CorrelId and Application Parallelism**

![Diagram](image)

**Notes:**

In this example, we might be sending out requests to different applications for information (perhaps bank account balance and a credit check for a loan application). Again, we need to be able to collect all the responses that are related to one particular request from the many replies on the ReplyToQ. We can't be sure about the order of the replies either. One time we may get the account balance information ahead of the credit check information. The next time, the replies might be reversed.

By issuing the MQGET using MsgId or CorrelId or both, the correct messages can be retrieved while others are left intact.

So, the application that issues the MQPUT of the initial message must be sure to set up some unique values in either or both fields. A program can put any desired value into either or both fields. In the case of the CorrelId, whatever value is in the field at the time of the MQPUT, is left by the queue manager. In the case of MsgId, the content of the field at MQPUT time will determine whether the queue manager assigns a unique value and places it in the MsgId field.
In all implementations, if an application sets the MsgId field to nulls (low values), the queue manager will then assign a unique value to the MsgId. Any other value assigned by the application will result in no action taken by the queue manager; in this way an application can specify a MsgId explicitly.

On the V5 distributed products (AIX, DOS client, HP-UX, OS/2, Sun Solaris, Windows client and Windows) and AS/400 a special Put Message Options option is supported to make the task of setting these fields simpler. By specifying MQPMO_NEW_MSG_ID and/or MQPMO_NEW_CORREL_ID, the application can ask the queue manager to put a unique value in the appropriate field. This would override any values that the application may place in these fields. It is the only way to have the queue manager place a unique value in the CorrelId field. These options are not supported for z/OS.
Retrieving Every Message

Notes:

The application controls which message it retrieves with MQGET. By setting BOTH MsgId and CorrelId in the MQMD to nulls (low values) before the MQGET, the next available message on the queue will be retrieved. However, once the MQGET is completed, the values in the MsgId and CorrelId will be initialized to those of the message just retrieved. If not continuously reset to nulls (low values), the application may receive a failing completion code with the reason set to MQRC_NO_MSG_AVAILABLE even though messages remain on the queue.

In the example shown, we see the use of symbolic constants to set MsgId and CorrelId to nulls (low values). Note the first case sets the MsgId and CorrelId before going into the MQGET loop while the second case has the setting of these fields within the loop.

Extra Options on Distributed Platforms Version 5.0 and z/OS Version 5.3

On these platforms, there is an additional field available in the Version 2 Get Message Option structure called MatchOptions. It is possible to specify MQMO_NONE and avoid having to reset the MsgId and CorrelId. This will result in retrieval on the next available message regardless of the value in MsgId and CorrelId.
Notes:

Let's examine the examples; in each case fields in the MQMD are being initialized prior to the MQGET:

1. Asks for the first available message regardless of the values of MsgId and CorrelId.
2. Requests a message that has a CorrelId of 3, regardless of the value of the MsgId field.
3. Will succeed if it finds a message with MsgId of 4, regardless of the value in CorrelId.
4. Unless a message with MsgId of 4 AND CorrelId of 2 is found, this call would not succeed.

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

Note: MsgId and CorrelId are defined as MQBYTE24 and are not converted.
Be aware that queues with many messages could cause a performance impact if an application chooses to use MQGET with MsgId and CorrelId. In general, queues that hold less than a hundred or so messages, will not cause any problems. However, if a queue has thousands of messages, the queue manager will actually scan the queue looking for a match. On z/OS, it is possible for the WebSphere MQ administrator to define either MsgId or CorrelId as an index. For example, if IndexType on a queue is MsgId, an index of all MsgIds on the queue is maintained and message retrieval by MsgId will be faster even if there are a large number of messages on the queue. Using a match of CorrelId will result in slower processing since it is not possible to specify IndexType of both.

Extra Options on Distributed Platforms Version 5.0 and z/OS Version 5.3

It is possible to specify MQMO_MATCH_MSG_ID and/or MQMO_MATCH_CORREL_ID in the MatchOptions field and avoid having to reset the MsgId and CorrelId. This means that NOT specifying MQMO_MATCH_MSG_ID will result in retrieval on the next available message regardless of the value in MsgId. The same holds true if the MQMO_MATCH_CORREL_ID is NOT specified. In other words, if no match options are specified, the MsgId and CorrelId field will be IGNORED on the MQGET, essentially behaving as if they were set to MQMI_NONE and MQCI_NONE, respectively. This makes applications running on these platforms a little easier to code.

The default on these platforms is MQMO_MATCH_MSG_ID and MQMO_MATCH_CORREL_ID. This means that no program changes are necessary for programs that are migrated from earlier releases. If the program was looking for a match, it will continue to do so. However, new programs will have less to consider. They will not have to clear MsgId and CorrelId before each MQGET if they do not want to match. They merely set the MatchOptions to MQMO_NONE.
**Msgid, Correlid, Reports and Replies**

**Notes:**

This visual relates the concepts of MsgId, CorrelId, reports, replies and selective MQGETs.

1. The requesting application creates a new MQMD, setting the MsgType to request, the reply to queue to Q2, the MsgId to MQMI_NONE and asking for the target queue manager to generate COA and COD reports.

2. Responding to MQMI_NONE in the MQMD MsgId field, the queue manager creates a unique MsgId for the new message - represented as 99 in this example and returns this value to the program in the MQMD.

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’s MsgId is copied to the report’s Correlid.

4. The replying application’s 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 Msgld to the outgoing Correlld.

6. There are now three messages on Q2: two reports generated by the replying application's queue manager, and one reply created by the replying application.

7. In order to retrieve only the replies and reports which relate to request 99, the requesting application sets the MQMD Msgld field to MQMI_NONE and the MQMD Correlld field to 99.

8. Subsequent MQGETs will 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.
Notes:

Our sample application is a loan request application from a bank.

1. Program 1 gathers the customer information required for the loan request, and puts a message on both Q1 and Q2, each with the same MsgId and Correlld.

2. Program 2 takes the input message, processes it and sends one message to Q3 in the bank and another message to Q4 at the credit bureau. Before placing the two messages on the output queues, Program 2 uses a convention that is common in WebSphere MQ:
   - Move MsgId from the input message to Correlld of the output message
   - Create a new and unique MsgId

3. Program 3, within the bank, gathers information about the customer’s history with the bank. Program 4, at the credit bureau, gathers information about the customer’s overall credit history. Both Programs 3 and 4 perform the same WebSphere MQ functions. Each program gets the message from its input queue, regardless of MsgId and Correlld,
and after processing, puts the resulting output message to Q5 without changing the Correlld value.

4. Program 5 retrieves a message from Q1 and saves the MsgId. Program 5 then moves that MsgId to the Correlld, gets the two messages from Q5 that match the CorrelId, processes the data, and places a message on Q6 that either confirms or denies the loan.

We are making some broad assumptions that all messages will arrive in time to complete the process. We will explore options in a moment to allow for the more probable case that messages might arrive at different times.

The above mentioned convention where Program 2 moves the incoming MsgId to Correlld and generates a new MsgId is the convention used when the queue manager generates report messages. So any application that expects to process reports from the queue manager must consider this.
## MSGID Pseudocode Program 1

<table>
<thead>
<tr>
<th>Line</th>
<th>Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>DEFINE MSG_DESC AS MQMD</td>
</tr>
<tr>
<td>2</td>
<td>DEFINE OBJ_DESC AS MQOD</td>
</tr>
<tr>
<td>3</td>
<td>Process</td>
</tr>
<tr>
<td>4</td>
<td>OBJ_DESC.ObjectName = &quot;Q1&quot;</td>
</tr>
<tr>
<td>5</td>
<td>MSG_DESC.MsgId = MQMI_NONE</td>
</tr>
<tr>
<td>6</td>
<td>MSG_DESC.CorrelId = MQCI_NONE</td>
</tr>
<tr>
<td>7</td>
<td>CALL MQPUT1(......, OBJ_DESC, MSG_DESC, ...)</td>
</tr>
<tr>
<td>8</td>
<td>Process</td>
</tr>
<tr>
<td>9</td>
<td>OBJ_DESC.ObjectName = &quot;Q2&quot;</td>
</tr>
<tr>
<td>10</td>
<td>MSG_DESC.CorrelId = MQCI_NONE</td>
</tr>
<tr>
<td>11</td>
<td>CALL MQPUT1(......, OBJ_DESC, MSG_DESC, ...)</td>
</tr>
</tbody>
</table>

### Notes:

- **Figure 6-9.** MSGID Pseudocode Program 1

- **SW3135.3**
MSGID Pseudocode Program 2 (1 of 2)

```
DEFINE IN_MSG_DESC AS MQMD
DEFINE OUT_MSG_DESC AS MQMD
DEFINE OBJ_DESC AS MQOD

CALL MQCONN(......,

CALL MQOPEN(......,

IN_MSG_DESC.MsgId = MQMI_NONE
IN_MSG_DESC.CorrelId = MQCI_NONE
CALL MQGET(......,

Process

OBJ_DESC.ObjectName = "Q3"

OUT_MSG_DESC.MsgId = MQMI_NONE
OUT_MSG_DESC.CorrelId = IN_MSG_DESC.MsgId
```

Notes:
MSGID Pseudocode Program 2 (2 of 2)

CALL MQPUT1(.......,
              OBJ_DESC,
              OUT_MSG_DESC,
              ...)

Process

OBJ_DESC.ObjectName = "Q4"

OUT_MSG_DESC.MsgId = MQMI_NONE

CALL MQPUT1(.......,
              OBJ_DESC,
              OUT_MSG_DESC,
              ...)

CALL MQCLOSE(.......,
              ...)

CALL MQDISC(.......,
              ...)

Notes:
### MSGID Pseudocode Programs 3 and 4

<table>
<thead>
<tr>
<th>Line</th>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>DEFINE IN(MSG_DESC AS MQMD</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>DEFINE OUT(MSG_DESC AS MQMD</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>DEFINE OBJ_DESC AS MQOD</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>CALL MQCONN(......,</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>CALL MQOPEN(......, for Q3 or Q4</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>IN_MSG_DESC.MsqId = MQMI_NONE</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>IN_MSG_DESC.CorrelId = MQCI_NONE</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>CALL MQGET(......,</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>OBJ_DESC.ObjectName = &quot;Q5&quot;</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>OUT_MSG_DESC.MsqId = MQMI_NONE</td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>OUT_MSG_DESC.CorrelId = IN_MSG_DESC.CorrelId</td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>CALL MQPUT1(......,</td>
<td></td>
</tr>
<tr>
<td>13</td>
<td>OBJ_DESC,</td>
<td></td>
</tr>
<tr>
<td>14</td>
<td>OUT_MSG_DESC,</td>
<td></td>
</tr>
<tr>
<td>15</td>
<td>...)</td>
<td></td>
</tr>
<tr>
<td>16</td>
<td>CALL MQCLOSE(......,</td>
<td></td>
</tr>
<tr>
<td>17</td>
<td>CALL MQDISC(......,</td>
<td></td>
</tr>
</tbody>
</table>

**Figure 6-12.** MSGID Pseudocode Programs 3 and 4

**Notes:**
### MSGID Pseudocode Program 5 (1 of 2)

```plaintext
DEFINE IN_MSG_DESC_1 AS MQMD
DEFINE IN_MSG_DESC_2 AS MQMD
DEFINE OUT_MSG_DESC AS MQMD
DEFINE OBJ_DESC AS MQOD

CALL MQCONN(......,

CALL MQOPEN(......, /* for Q1 */

IN_MSG_DESC_1.MsgId = MQMI_NONE
IN_MSG_DESC_1.Msgld = MQCI_NONE
CALL MQGET(......,

CALL MQCLOSE(......,

CALL MQOPEN(......, /* for Q5 */

IN_MSG_DESC_2.MsgId = MQMI_NONE
IN_MSG_DESC 2.CorrelId = IN_MSG_DESC_1.MsgId
CALL MQGET(......,

IN_MSG_DESC_2.MsgId = MQMINONE
CALL MQGET(......,
```

Figure 6-13. MSGID Pseudocode Program 5 (1 of 2)

**Notes:**
### MSGID Pseudocode Program 5 (2 of 2)

```c
CALL MQCLOSE(......,  
  Process  
  OBJ_DESC.ObjectName = "Q6"  
  OUT_MSG_DESC.MsgId = MQMI_NONE  
  OUT_MSG_DESC.CorrelId = MQCI_NONE  
  CALL MQPUT1(......,  
    OBJ_DESC,  
    OUT_MSG_DESC,  
    ...)  
  CALL MQDISC(......,  
```

**Notes:**
Notes:

The graphic above is the solution and has the UM1 in the relative positions after the programs have been executed.

We provide our text about the loan request application from a bank again as a convenience.

1. Program 1 gathers the customer information required for the loan request, and puts a message on both Q1 and Q2, each with the same MsgId and CorrelId.

2. Program 2 takes the input message, processes it and sends one message to Q3 in the bank and another message to Q4 at the credit bureau. Before placing the two messages on the output queues, Program 2 uses a convention that is common in WebSphere MQ:
   - Move MsgId from the input message to CorrelId of the output message
   - Create a new and unique MsgId

3. Program 3, within the bank, gathers information about the customer's history with the bank. Program 4, at the credit bureau, gathers information about the customer's overall...
credit history. Both Programs 3 and 4 perform the same WebSphere MQ functions. Each program gets the message from its input queue, regardless of MsgId and CorrelId, and after processing, puts the resulting output message to Q5 without changing the CorrelId value.

4. Program 5 retrieves a message from Q1 and saves the MsgId. Program 5 then moves that MsgId to the CorrelId, gets the two messages from Q5 that match the CorrelId, processes the data, and places a message on Q6 that either confirms or denies the loan.

We are making some broad assumptions that all messages will arrive in time to complete the process. We will explore options in a moment to allow for the more probable case that messages might arrive at different times.

The above mentioned convention where Program 2 moves the incoming MsgId to CorrelId and generates a new MsgId is the convention used when the queue manager generates report messages. So any application that expects to process reports from the queue manager must consider this.
6.2 Browse
Notes:

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. There is no way to browse backward on a queue.

In order 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 as shown in cases one and two, the application merely moves along through the messages in the queue, retrieving 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 this Browse Cursor that monitors which message is currently being pointed to by the application. The Browse Cursor is not something that can be manipulated by the program except by issuing MQGET using 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 strongly advised that a Wait Interval be specified.

Since the browse is unidirectional, it is possible that 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 have been 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 wishes to remove from the queue, a further MQGET must be issued using the option MQGMO_MSG_UNDER_CURSOR. This will result in a destructive removal of the message pointed to by the Browse Cursor.
**Browsing the Same Message**

![Diagram showing the sequence of events for browsing the same message.](image)

**Notes:**

The numbers in the visual above represent the sequence of events.

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

In fact, two applications could 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 will fail with MQRC_NO_MSG_UNDER_CURSOR. In effect, the cursor is pointing to an empty slot in the queue.

This problem could be avoided by using another option that can be combined with the MQGMO_BROWSE... - the MQGMO_LOCK. This will make the message being browsed invisible and unavailable to any other application that may also be browsing or getting messages from the queue. There is an MQGMO_UNLOCK to allow for an explicit unlock of a previously locked message (must use same handle). The message will also be unlocked by a successful MQCLOSE using the same handle.
Browsing Pseudocode (1 of 2)

```plaintext
DEFINE COMP_CODE AS MQLONG
DEFINE REASON_CODE AS MQLONG
DEFINE GET_OPTIONS AS MQGMO

CALL MQCONN(......,

CALL MQOPEN(......,

GET_OPTIONS.Options = MQGMO_BROWSE_FIRST + MQGMO_LOCK

CALL MQGET(......,

DO WHILE COMP_CODE = MQCC_OK

IF we need to process this message

GET_OPTIONS.Options = MQGMO_MSG_UNDER_CURSOR

CALL MQGET(......,

IF COMP_CODE NOT = MQCC_OK
```

Notes:
### Browsing Pseudocode (2 of 2)

<table>
<thead>
<tr>
<th>Line</th>
<th>Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>23</td>
<td>IF REASON_CODE NOT = MQRC_NO_MSG_UNDER_CURSOR</td>
</tr>
<tr>
<td>24</td>
<td>/* investigate what has gone wrong */</td>
</tr>
<tr>
<td>25</td>
<td>END-IF</td>
</tr>
<tr>
<td>26</td>
<td>ELSE</td>
</tr>
<tr>
<td>27</td>
<td>/* process the message */</td>
</tr>
<tr>
<td>28</td>
<td>END-IF</td>
</tr>
<tr>
<td>29</td>
<td>END-IF</td>
</tr>
<tr>
<td>30</td>
<td>GET_OPTIONS.Options = MQGMO_BROWSE_NEXT + MQGMO_LOCK</td>
</tr>
<tr>
<td>31</td>
<td>CALL MQGET(......,</td>
</tr>
<tr>
<td>32</td>
<td>END-DO</td>
</tr>
<tr>
<td>33</td>
<td>/*find out why we terminated the loop and */</td>
</tr>
<tr>
<td>34</td>
<td>/*take the appropriate action if required */</td>
</tr>
</tbody>
</table>

### Notes:

---

Figure 6-19. Browsing Pseudocode (2 of 2)
6.3 Waiting for Replies
Notes:

Because WebSphere 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 has been put. In these situations if the application is designed to be synchronous, the MQGMO_WAIT option can be useful.

If an application uses MQGMO_WAIT, it is recommended that the MQGMO_FAIL_IFQUIESCING option is used as well. This allows the call to be terminated if the queue manager is quiescing, returning MQRC_Q_MGRQUIESCING.

The wait can also terminate if conditions regarding the queue's ability to deliver messages changes. For instance, if the WebSphere MQ administrator changes the Get attribute for the queue from allowed to inhibited, the call would then fail with MQRC_GETINHIBITED.
WAIT with WaitInterval

Notes:

Within the Get Message Option structure is a field called WaitInterval to be 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 allow the program to establish 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.

Note that the default wait interval is MQWI_UNLIMITED (forever). This should be used only with extreme discretion, and always with MQGMO_FAIL_IF_QUIESCING.
### WAIT with WaitInterval Pseudocode (1 of 2)

<table>
<thead>
<tr>
<th>Line</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>DEFINE COMP_CODE AS MQLONG</td>
</tr>
<tr>
<td>2</td>
<td>DEFINE REASON_CODE AS MQLONG</td>
</tr>
<tr>
<td>3</td>
<td>DEFINE MESSAGE_DESCRIPTOR AS MQMD</td>
</tr>
<tr>
<td>4</td>
<td>DEFINE GET_OPTIONS AS MQGMO</td>
</tr>
<tr>
<td>5</td>
<td>CALL MQCONN(......,</td>
</tr>
<tr>
<td>6</td>
<td>CALL MQOPEN(......,</td>
</tr>
<tr>
<td>7</td>
<td>MESSAGE_DESCRIPTOR.MsgId = SomeValue</td>
</tr>
<tr>
<td>8</td>
<td>MESSAGE_DESCRIPTOR.CorrelId=SomeOtherValue</td>
</tr>
<tr>
<td>9</td>
<td>GET_OPTIONS.Options = MQGMO_WAIT + MQGMO_FAIL_IF_QUIESCING</td>
</tr>
<tr>
<td>10</td>
<td>GET_OPTIONS.WaitInterval = 300000</td>
</tr>
</tbody>
</table>

**Notes:**
WAIT with WaitInterval Pseudocode (2 of 2)

CALL MQGET(...., 17
IF COMP_CODE NOT = MQCC_OK 18
IF REASON_CODE=MQRC_NO_MSGAVAILABLE 19
 /*we were woken up but there was no message*/ 20
 /*available - timeout expired*/ 21
END-IF 22
IF REASON_CODE=MQRC_GETINHIBITED 23
 /*we were woken up but we are not now*/ 24
 /*allowed to get a message*/ 25
END-IF 26
ELSE 27
 /*we get the message OK, now process it*/ 28
END-IF 29

Figure 6-23. WAIT with WaitInterval Pseudocode (2 of 2)

Notes:
Another Example of WAIT

Notes:
If multiple programs are processing a queue for input, it is possible that a message that has satisfied a wait could be removed by another program that is, perhaps, just getting any available message from the queue. It is unpredictable which application will actually retrieve the message from a queue in this situation: it depends on the relative operating system priorities of the competing tasks.

Applications should therefore always be designed to allow for the possibility of receiving MQRC_NO_MSG_AVAILABLE, even if the wait was set up with no WaitInterval.
Notes:

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

An example is an z/OS IMS environment. An IMS system cannot be terminated 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 there was no way to interrupt the wait, the IMS system could not be terminated normally.
Notes:

It is an option on the MQGET on some WebSphere MQ implementations to allow the operating system to notify (or signal) the program when an expected message arrives on the queue. You can not use the MQGMO_SET_SIGNAL option in conjunction with the MQGMO_WAIT option. However, the WaitInterval is permitted and, in the case of IMS, is recommended to be used with the SET SIGNAL option.

The advantage of the MQGMO_SET_SIGNAL is that the application thread is freed to do other processing, relying on the operating system to notify the program when a message arrives.

A program can have outstanding signals set on multiple queue handles simultaneously. When the operating system notifies the application that a message has arrived for an outstanding signal request, the program must issue another MQGET to actually retrieve the message.
The notification of arrival is operating system specific: for example, on z/OS, a completion code is returned to the Event Control Block (ECB) whose address is identified in the Signal1 field of the Get Message Options structure.

The completion code can notify the program that:

1. That the message has arrived on the queue.
2. The WaitInterval expired and no message has arrived.
3. The signal has been cancelled (for example, Get Inhibited has been set for the queue).
**Notes:**

In our example, the program uses the MQGMO_SET_SIGNAL as one of its Get Message Options options. The Signal1 field of the Get Message Options structure contains the address of the ECB. The program may also set a value in WaitInterval to indicate how long the Set Signal should be active. The MQGET is then issued.
Notes:

The result of the MQGET using the SET_SIGNAL option can vary.

- If a message is available, it is returned to the program.
- If no message is available, the call completes with MQCC_WARNING and MQRC_SIGNAL_REQUEST_ACCEPTED

If the WaitInterval was set, the SET_SIGNAL will be in effect until a message arrives or until the WaitInterval expires. If no WaitInterval is specified, the set remains in effect until the SET_SIGNAL is satisfied.
Notes:

It is possible that, should the MQGET with SET_SIGNAL be accepted, and the ECB later be posted, a reason of MQRC_NO_MSGAVAILABLE can be encountered. Another program may retrieve the message between the time the ECB is posted and the time that the second MQGET is issued.

The completion code mentioned earlier that is returned as part of the signal will contain one of the following values:

- MQEC_MSG_ARRIVED
- MQEC_WAIT_INTERVAL_EXPIRED
- MQEC_WAIT_CANCELED
- MQEC_Q_MGR QUIESCING
- MQEC_CONNECTION QUIESCING

Obviously, before assuming a message has arrived, the ECB should be examined to determine what completion code was returned.
## SIGNAL Pseudocode

```plaintext
DEFINE COMP_CODE AS MQLONG  
DEFINE REASON_CODE AS MQLONG  
DEFINE MESSAGE_DESCRIPTOR AS MQMD  
DEFINE GET_OPTIONS AS MQGMO  
DEFINE MY_ECB AS ECB  

CALL MQCONN(......,  
CALL MQOPEN(......,  
MESSAGE_DESCRIPTOR.MsgId = SomeValue  
MESSAGE_DESCRIPTOR.CorrelId = SomeOtherValue  
GET_OPTIONS.Options = MQGMO_SET_SIGNAL + MQGMO_FAIL_IF_QUIESCING  
GET_OPTIONS.Signal1 = ADDRESS of MY_ECB  
GET_OPTIONS.WaitInterval = 300000  
CALL MQGET(......,  
IF REASON_CODE=MQRC_SIGNAL_REQUEST_ACCEPTED  
  Do some processing  
  Wait on ADDRESS of MY_ECB  
  MQGET new message
```

---

### Notes:

Figure 6-30. SIGNAL Pseudocode

SW3135.3
**Expiry**

*Notes:*

The Expiry value is the message lifetime. This is a period of time expressed in tenths of a second, set by the application that puts the message. The message becomes eligible to be discarded if it has not been removed from the destination queue before this period of time elapses.

In the current implementations, the message is discarded when a browse or nonbrowse MQGET call occurs that would have returned the message had it not already expired.

The default value is MQEI_UNLIMITED for an unlimited lifetime.

The expiry value is usually used in conjunction with the wait interval. You should set the expiry value to that of the wait interval as you are not willing to wait any longer then that time.
Checkpoint - Building Replies and Reports

What This Exercise Is About

This exercise is to familiarize students with considerations for messaging in a remote environment. Students will answer some questions about building a reply message and a report message based on the input they receive.

What You Should Be Able to Do

1. Answer the questions regarding the setup for the reply to a specified queue.
2. Complete similar responses to send a report message to the same queue.

Introduction

You will answer the questions while using the supplied messages. The content of the message you send back is not important. It is more important that you find the information you need to know where to send your messages as well as what fields to manipulate to send back the proper information.

The messages you will work with that are on your queue are captures of the information that is displayed by amqsbcg.
Exercise Instructions

REPLY

1. Using the first message (the text asks for a reply), consider what is needed to ensure that the reply is properly sent.

2. What value will you put in MsgId?

3. What will you put in CorrelId?

4. What will you put in Report?

5. What will the MsgType be?

6. Will you put anything in ReplyToQ?

7. What will you put in the ObjectName of the Object Descriptor?

8. What other fields will you change?

   1. SETTING UP A REPLY

   ****Message descriptor****

   StrucId : 'MD '  Version : 2
   Report : 1  MsgType : 1
   Expiry : -1  Feedback : 0
   Encoding : 17  CodedCharSetId : 437
   Format : 'MQSTR '
   Priority : 0  Persistence : 0
  _MsgId_ : X'414D51204D51303520202020202020202095D6C83413100000'
   CorrelId : X'0000000000000000000000000000000000000000000000000000000000000000'
   BackoutCount : 0
   ReplyToQ : 'AMQ.1998012317430538'
   ReplyToQMgr : 'MQ05'

   ** Identity Context
   UserIdentifier : 'mqm '
   AccountingToken : X'0131000000000000000000000000000000000000000000000000000000000000'
   ApplIdentityData :

   ** Origin Context
   PutApplType : '11'
   PutApplName : ':\mgm\TOOLS\mq05\request.exe'
   PutDate : '20030812'  PutTime : '17432682'
   ApplOriginData :

   GroupId : X'0000000000000000000000000000000000000000000000000000000000000000'
   MsgSeqNumber : '1'
   Offset : '0'
   MsgFlags : '0'
   OriginalLength : '48'
***** Message *****

length - 48 bytes

00000000: 506C 6561 7365 2073 656E 6420 6D65 2061 'Please send me a'

REPORT

__ 9. Using the second message (the text asks for a report), consider what you need to do
to ensure that the report is delivered.

__ 10. What value will you put in MsgId?
__ 11. What will you put in CorrelId?
__ 12. What will you put in Report?
__ 13. What will the MsgType be?
__ 14. Will you put anything in ReplyToQ?
__ 15. What will you put in the ObjectName of the Object Descriptor?
__ 16. What other fields will you change?

2. SETTING UP A REPORT

****Message descriptor****

StrucId : 'MD '  Version : 2
Report : 1    MsgType : 1
Expiry : -1   Feedback : 0
Encoding : 17  CodedCharSetId : 437
Format : 'MQSTR '
Priority : 0    Persistence : 0
MsgId : X'414D51204D513035202020202020202095D6L83423100000'
CorrelId : X'0000000000000000000000000000000000000000000000000000000000000000'
BackoutCount : 0
ReplyToQ : 'AMQ.1998012317430538 '
ReplyToQMgr : 'MQ05 '
** Identity Context
UserIdentifier : 'mqm     '
AccountingToken : X'0131000000000000000000000000000000000000000000000000000000000000'
ApplIdentityData : '
** Origin Context
PutApplType    : '11'
PutApplName    : ':\mqm\TOOLS\mq05\request.exe'
PutDate  : '20030812'    PutTime  : '17472948'
ApplOriginData : ''

GroupId : X'000000000000000000000000000000000000000000000000000'
MsgSeqNumber  : '1'
Offset        : '0'
MsgFlags      : '0'
OriginalLength : '48'

****   Message      ****

length - 48 bytes

00000000:  506C 6561 7365 2073 656E 6420 6D65 2070 'Please send me a'
Checkpoint

1. Which two fields in the MQMD allow responses to be related to requests?

2. How does the application ask the queue manager to generate a new unique MsgId when creating a message?

3. What two things must an application set when establishing an MQGET with a wait?

Notes:
Please write your answers here.
1.
2.
3.
Unit Summary

- MsgId and CorrelId allow control over message retrieval
- Use MessageId/CorrelId to relate responses to specific requests
- Waiting for replies enables processing to be automated

Notes:

Now that we have learned about MsgId and CorrelId, it becomes clear that an application programmer needs to know how to manipulate these fields both when putting messages on a queue and when retrieving them.

MsgId and CorrelId allow retrieval of specific messages as well as providing the capability of relating messages that may be part of one processing unit (for example, a hotel, plane and car confirmation for one individual’s trip).

Finally, because of the asynchronous nature of WebSphere MQ, we now know that we can not always expect our replies to be available just because the application has reached the point where it is ready to process them.

Use of the MQGMO_WAIT and MQGMO_SET_SIGNAL options along with the WaitInterval allow greater control over when messages are retrieved.
Exercise 4

Do Lab Exercise 4

Notes:
Unit 7. MQINQ and MQSET

What This Unit Is About

This unit focuses on the MQINQ and MQSET calls that allow you to programmatically obtain information about a WebSphere MQ object, such as a queue, queue manager, process or namelist and to set the attributes of a queue.

What You Should Be Able to Do

After completing this unit, you should be able to:

• Write a program that inquires about the attributes of a WebSphere MQ object
• Write a program that changes the attributes of a WebSphere MQ queue

How You Will Check Your Progress

Accountability:

• Classroom discussion
• Machine exercise

References

SC34-6064  WebSphere MQ Application Programming Guide
SC34-6062  WebSphere MQ Application Programming Reference
Unit Objectives

- Describe the format of the MQINQ call
- Describe which attributes can be inquired
- Explain the function of the MQSET call
- Describe which attributes can be changed with MQSET

Notes:

Attributes are the properties that define the characteristics of a WebSphere MQ object. They affect the way that an object is processed by a queue manager. The attributes of each type of WebSphere MQ object are described in detail in the Application Programming Reference manual.

Some attributes are set when the object is defined, and can be changed only by using the WebSphere MQ commands; an example of such an attribute is the default priority for messages put on a queue. Other attributes are affected by the operation of the queue manager and may change over time; an example is the current depth of a queue.

You can inquire about the current values of all these attributes using the MQINQ call. The MQI also provides an MQSET call with which you can change some queue attributes.
7.1 MQINQ and MQSET
Why MQINQ and MQSET?

- To query the maximum message length for a queue to set up the input buffer size before the MQGET
- To query the number of names in a namelist
- To query the application name that implements a process, for accounting purposes
- To obtain backout-requeue queue relevant information for poison messages
- To change a queue with TRIGTYPE=DEPTH from NOTRIGGER to TRIGGER

Notes:

MQINQ allows you to interrogate all of the attributes of any queue, process, queue manager or namelist. From within a program, you can use this call to discover such things as a queue’s maximum message length, the application pointed to by a process, the name of the queue manager’s dead letter queue, or the list of attributes for a namelist. MQSET allows you to change attributes, but only the attributes of a queue. The attributes that you can set are primarily associated with triggering.

Both the MQINQ and the MQSET calls use arrays of selectors to identify those attributes you want to interrogate or set. There is a selector for each attribute you can work with. The selector name has a prefix, determined by the nature of the attribute:

- MQCA_ These selectors refer to attributes that contain character data (for example, the name of a queue).
- MQIA_ These selectors refer to attributes that contain either numeric values (such as CurrentQueueDepth, the number of messages on a queue) or a constant value (such as SyncPoint, whether or not the queue manager supports syncpoints).
Before you use the MQINQ or MQSET calls your application must be connected to the queue manager, and you must use the MQOPEN call to open the object for setting or inquiring about attributes.
Notes:
The MQINQ call can be used to find out the settings of attributes of WebSphere MQ objects. This includes queues, queue managers, processes and namelists (z/OS and distributed Version 5 only). The call returns an array of integers and a set of character strings that contains the requested attributes. The object needs to be opened for Inquiry before the MQINQ is used.

Reviewing the parameters, it is obvious that the MQINQ (and MQSET) require more than any of the other calls. The common parameters used on other calls include the connection handle (Hconn) and the object handle (Hobj) as the first two parameters. The last two, as with all calls in WebSphere MQ, are the completion code (CompCode) and reason code (Reason).

By breaking the other parameters into groups, it is fairly easy to understand them:

- SelectorCount and Selectors
- IntAttrCount and IntAttrs
- CharAttrLength and CharAttrs
SelectorCount is the total number of attributes that are included in the MQINQ. It is simply a count of how many attributes you are asking about. The Selectors is an array (or list) of those attributes. Each attribute has a symbolic name assigned which can be used in place of the numeric value assigned to that attribute. All of the attributes are listed by object in the discussion of MQINQ in *IBM WebSphere MQ Application Programming Reference*.

IntAttrCount is the total number of the selectors that are represented as integers. These include things like MQIA_CURRENT_Q_DEPTH, MQIA_Q_TYPE, or MQIA_MAX_Q_DEPTH. The IntAttrs is an array (or list) that will be used to return the requested integer attributes.

CharAttrLength is the total length of all the character attributes that have been requested. CharAttrs is the name of a buffer area where all of the requested character attributes will be returned as one long string up to the length specified in CharAttrLength.

It is apparent that the character attributes will require a bit more work to set up. If the character attributes MQCA_Q_NAME and MQCA_Q_DESC were desired, the lengths of these two attributes (48 and 64, respectively) would be added together to set the value in CharAttrLength (112).

It is not necessary to know the length of each of these fields since the lengths also have symbolic names (listed with the attribute names in the Application Programming Reference). In our example, the value of CharAttrLength can be obtained by adding MQ_Q_NAME_LENGTH and MQ_Q_DESC_LENGTH. Of course, you must ensure that you have a buffer that will accommodate the longest string of characters you would expect to have returned.

Responses are returned in the exact order requested. Assume that the Selectors are:

1. MQIA_CURRENT_Q_DEPTH
2. MQCA_Q_NAME
3. MQIA_MAX_Q_DEPTH
4. MQCA_Q_DESC

A buffer of at least 112 characters has been set up for characters and an array has been set up for integer attributes. The program supplies the value of 2 in the IntAttrCount and 112 in the CharAttrLength. On return from the call, the value in the first occurrence of the integer array will be that of the current depth (MQIA_CURRENT_Q_DEPTH), followed by maximum depth (MQIA_MAX_Q_DEPTH). The buffer of character strings will contain the queue name (MQCA_Q_NAME) in the first 48 characters, immediately followed by the 64-character queue description (MQCA_Q_DESC). The program would need to be set up to break up the character string into the various pieces.
Figure 7-4. MQINQ Pseudocode (1 of 2)

```
DEFINE CONN_HANDLE AS MQHCONN
DEFINE OBJ_HANDLE AS MQHOBJ
DEFINE COMP_CODE AS MQLONG
DEFINE REASON_CODE AS MQLONG
DEFINE SELECTORS AS ARRAY 5 MQLONG
DEFINE SELECTOR_COUNT AS MQLONG
DEFINE INT_COUNT AS MQLONG
DEFINE INTS AS ARRAY 3 MQLONG
DEFINE CHAR_COUNT AS MQLONG
DEFINE CHARS AS CHAR2000

CALL MQCONN(......,
CALL MQOPEN(......,
SELECTOR_COUNT = 4
SELECTORS(1) = MQCA_BACKOUT_REQ_Q_NAME
SELECTORS(2) = MQIA_BACKOUT_THRESHOLD
SELECTORS(3) = MQCA_CREATION_DATE
SELECTORS(4) = MQIA_INHIBIT_GET
```

Notes:
### MQINQ Pseudocode (2 of 2)

```
INT_COUNT = 2

CHAR_COUNT = MQ_Q_NAME_LENGTH + MQ_CREATION_DATE_LENGTH

MQINQ(CONN_HANDLE,
    OBJ_HANDLE,
    SELECTOR_COUNT,
    SELECTORS,
    INT_COUNT,
    INTS,
    CHAR_COUNT,
    CHARs,
    COMP_CODE,
    REASON_CODE)
```

---

**Notes:**
Format of MQSET

The format of the MQSET is the same as that of MQINQ. The difference is in the direction of the flow of some of the data. Fields that were output fields on MQINQ (IntAttrs and CharAttrs) are now input to the call.

So, the MQSET works exactly like the MQINQ except that the buffers must be set up by the application rather than filled in by the queue manager.

A more important difference is that the MQSET is not permitted for any object except a queue. In addition, only a small subset of attributes can be changed. The attribute that indicates if PUTs are permitted (MQIA_INHIBIT_PUT) is settable for all queue types. The MQIA_GET_INHIBITED attribute can only be set for local and alias queues. The following attributes for a local queue that are associated with triggering are permitted:

- MQCA_TRIGGER_DATA
- MQIA_TRIGGER_CONTROL
- MQIA_TRIGGER_DEPTH
- MQIA_TRIGGER_MSG_PRIORITY
- MQIA_TRIGGER_TYPE
And MQIA_DIST_LISTS is permitted for those platforms which support distribution lists.

The queue must be opened using the MQOO_SET as one of the open options. If any of the attributes fail to be set (perhaps one of the requested attributes is not permitted with MQSET), then none of the requested attributes are set and the call fails. If the call is successful, the changes take effect immediately.
### MQSET Pseudocode (1 of 2)

<table>
<thead>
<tr>
<th>Line</th>
<th>Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>DEFINE CONN_HANDLE AS MQHCONN</td>
</tr>
<tr>
<td>2</td>
<td>DEFINE OBJ_HANDLE AS MQOBJ</td>
</tr>
<tr>
<td>3</td>
<td>DEFINE COMP_CODE AS MQLONG</td>
</tr>
<tr>
<td>4</td>
<td>DEFINE REASON_CODE AS MQLONG</td>
</tr>
<tr>
<td>5</td>
<td>DEFINE SELECTORS AS ARRAY 5 MQLONG</td>
</tr>
<tr>
<td>6</td>
<td>DEFINE SECTOR_COUNT AS MQLONG</td>
</tr>
<tr>
<td>7</td>
<td>DEFINE INT_COUNT AS MQLONG</td>
</tr>
<tr>
<td>8</td>
<td>DEFINE INTS AS ARRAY 3 MQLONG</td>
</tr>
<tr>
<td>9</td>
<td>DEFINE CHAR_COUNT AS MQLONG</td>
</tr>
<tr>
<td>10</td>
<td>DEFINE CHAR AS CHAR2000</td>
</tr>
<tr>
<td>11</td>
<td>CALL MQCONN(......,</td>
</tr>
<tr>
<td>12</td>
<td>CALL MQOPEN(......,</td>
</tr>
<tr>
<td>13</td>
<td>SELECTOR_COUNT = 2</td>
</tr>
<tr>
<td>14</td>
<td>SELECTORS(1) = MQIA_TRIGGER_TYPE</td>
</tr>
<tr>
<td>15</td>
<td>SELECTORS(2) = MQIA_TRIGGER_DEPTH</td>
</tr>
</tbody>
</table>

**Notes:**
MQSET Pseudocode (2 of 2)

<table>
<thead>
<tr>
<th>Line</th>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>21</td>
<td>INTS(1) = MQTT_DEPTH</td>
<td></td>
</tr>
<tr>
<td>22</td>
<td>INTS(2) = 10</td>
<td></td>
</tr>
<tr>
<td>23</td>
<td>INT_COUNT = 2</td>
<td></td>
</tr>
<tr>
<td>24</td>
<td>CHAR_COUNT = 0</td>
<td></td>
</tr>
<tr>
<td>25</td>
<td>MQSET(CONN_HANDLE,</td>
<td></td>
</tr>
<tr>
<td>26</td>
<td>OBJ_HANDLE,</td>
<td></td>
</tr>
<tr>
<td>27</td>
<td>SELECTOR_COUNT,</td>
<td></td>
</tr>
<tr>
<td>28</td>
<td>SELECTORS,</td>
<td></td>
</tr>
<tr>
<td>29</td>
<td>INT_COUNT,</td>
<td></td>
</tr>
<tr>
<td>30</td>
<td>INTS,</td>
<td></td>
</tr>
<tr>
<td>31</td>
<td>CHARS_COUNT,</td>
<td></td>
</tr>
<tr>
<td>32</td>
<td>CHARS,</td>
<td></td>
</tr>
<tr>
<td>33</td>
<td>COMP_CODE,</td>
<td></td>
</tr>
<tr>
<td>34</td>
<td>REASON_CODE)</td>
<td></td>
</tr>
</tbody>
</table>

Figure 7-8. MQSET Pseudocode (2 of 2)

Notes:
Checkpoint

1. How do the names of character attribute constants begin?

2. Name two common uses for the MQSET call.

Notes:
Please write your answers here.
1.
2.
Unit Summary

- MQINQ can list attributes of queues, queue managers, processes and namelists
- MQSET can change some attributes of queues

Notes:

MQINQ and MQSET are not commonly used calls, but can prove to be useful for gathering information about the WebSphere MQ objects that your program uses.
Exercise 5

Do Lab Exercise 5

Notes:
Unit 8. Transaction Support and Triggering

What This Unit Is About

Transaction Support and Triggering focuses on the syncpoint control calls (MQBEGIN, MQCMIT, and MQBACK) as well as a discussion of triggering and its effect on application design.

What You Should Be Able to Do

After completing this unit, you should be able to:

• Describe the meaning of logical units of work and understand how syncpointing is implemented in WebSphere MQ
• Understand how applications can be started by WebSphere MQ when messages arrive on its queue

How You Will Check Your Progress

Accountability:

• Classroom discussion
• Machine exercise
• Optional classroom exercise

References

SC34-6064  WebSphere MQ Application Programming Guide
SC34-6062  WebSphere MQ Application Programming Reference
SC34-6275  WebSphere MQ Extended Transactional Clients

http://www.ibm.com/software/integration/wmq
IBM’s WebSphere MQ Web Page
Unit Objectives

- Understand the WebSphere MQ transaction related facilities
- Understand how WebSphere MQ clients can participate in transactions
- Review triggering and understand how to implement it in a WebSphere MQ application

Notes:

WebSphere MQ has some capabilities that allow it to perform unit of work processing. A WebSphere MQ queue manager can act as its own resource manager, work in conjunction with a transaction manager, and on some platforms, provide global unit of work coordination of its own. Also WebSphere MQ clients can participate in units of work.

WebSphere MQ also has the ability to operate in an event-driven mode, by using the triggering ability of queues. Instead of writing programs that must constantly poll for messages, WebSphere MQ applications may be automatically started by the arrival of a message on a queue.
8.1 Syncpoint Control Calls
Notes:

In the situation above, the application would like to have both queue updates and the database update occur, or none of the updates. This is sometimes referred to as the transactionality of the application. In WebSphere MQ, the transactionality of MQ resources is ensured by getting and putting in 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 be retrieved by other programs.

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 enables a program to ensure 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 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 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. The unit of work is then generally coordinated by a transaction manager. CICS, ENCINA and TUXEDO are examples of transaction managers.
Notes:

A local unit of work involves only updates to WebSphere MQ resources (queues). A global unit of work includes the updates to resources belonging to other resource managers. The Version 5 queue managers (AIX, HP-UX, OS/2, Sun Solaris and Windows) support queue manager coordination of global units of work in some cases.

To start the global unit of work, a special call - MQBEGIN is used. The only parameters 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 MQCMIT call is used. MQBACK is used to back out all updates. These calls will be reviewed later in more detail. Presently, certain database resource managers can have their updates coordinated by WebSphere (DB2, Sybase, and Oracle).

External syncpoint coordination occurs when a syncpoint coordinator other than WebSphere MQ has been selected; for example, CICS, Encina, or Tuxedo. In this situation, WebSphere MQ register 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 MQCMIT, MQBACK, and MQBEGIN may not be issued. Calls to these functions fail with the reason code MQRC_ENVIRONMENT_ERROR.
Syncpoint

- Option on each GET or PUT
  - NO_SYNCPOINT - message is added or removed immediately
  - SYNCPOINT - PUT or GET becomes visible when transaction is committed

- Default is platform-dependent

Figure 8-4. Syncpoint

Notes:

Not all MQGETs and MQPUTs within a program need to 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, respectively.

With no syncpoint, the messages retrieved from a queue are removed from the queue immediately (a destructive read) and messages put on an output queue are immediately available to be retrieved by other programs.

Using syncpoint delays deletion of messages that are retrieved from input queues and delays delivery of messages that have been placed on output queues until the unit of work is committed.

Finally, remember that it is best to explicitly specify the desired syncpoint option since the default varies depending on the platform where the program executes.
**Notes:**

In order for a message to be part of a unit of work, the Put Message Options option of MQPMO_SYNCPOINT must be used. The opposite of this is MQPMO_NO_SYNCPOINT. It is recommended that the desired option be explicitly set since the default differs by platform.

Assuming 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, messages can continue to be placed on the queue using syncpoint. They will count toward the number of messages on the queue (they are part of the physical count) and the current depth will reflect these messages but, any attempt to use MQGET to retrieve them will fail with MQRC_NO_MSG_AVAILABLE.

Only after the commit is issued will the messages be available to other applications.
MQGET within Syncpoint Control

Notes:
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 MQGETs. Once again, the physical number of messages on a queue include those that have been gotten in syncpoint. The messages are not physically removed until the commit is completed (delayed deletion).

If a rollback occurs, the message is simply made available again and the delete flag is turned off.
Notes:
The MQBEGIN call begins a unit of work that is coordinated by the queue manager, and that may involve external resource managers.

The parameters again are very 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.

This call is supported in the following environments: AIX, HP-UX, OS/2, OS/400, Solaris, Linux, and Windows.
**Notes:**

Once a program is satisfied with all processing that has been done within a unit of work, the WebSphere MQ call MQCMIT can be used to process the completion of the unit of work, physically deleting any messages that were retrieved from input queues and making messages that were put to output queues available to other applications.

This call can not be used if the queue manager is running in an environment where an external coordinator is used (CICS, IMS, ENCINA, and so forth). Any attempt to use MQCMIT, MQBACK or MQBEGIN will result in a failure with MQRC_ENVIRONMENT_ERROR.

All MQPUTs and MQGETs that have been done in syncpoint (and any participating updates for other resource managers that occurred after the MQBEGIN call) will be considered part of the unit of work and be committed. If a global unit of work is being committed (including non-WebSphere MQ resources), then a two-phase commit process is used. This means that 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 will the second phase (the actual commit) occur. Otherwise, all the involved resources will be rolled back.
Notes:
The MQBACK call works in the reverse of the MQCMIT call. All resources involved in the unit of work are put back to the way things were before the unit of work began. Messages retrieved are no longer marked for deletion and are available to be gotten again. Messages that have been put are removed from the output queues. In the case of a global unit of work (identified by the MQBEGIN), the MQBACK results in all updates done by those other resource managers being undone as well.

The parameters again are very 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, any messages that had been retrieved using an MQGET in syncpoint will have the BackoutCount incremented by one.
## Syncpoint Pseudocode

<table>
<thead>
<tr>
<th>Line</th>
<th>Code</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>DEFINE GET_OPTIONS AS MQGMO</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>DEFINE PUT_OPTIONS AS MQPMO</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>CALL MQCONN(.......,)</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>CALL MQOPEN(.......,)</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>GET_OPTIONS.Options = MQGMO_NO_WAIT + MQGMO_SYNCPOINT +</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>MQGMO_FAIL_IF_QUIESCING</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>CALL MQGET(.......,)</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>PUT_OPTIONS.Options = MQPMO_SYNCPOINT +</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>MQPMO_FAIL_IF_QUIESCING</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>CALL MQPUT1(.......,)</td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>CALL MQCMIT(.......,)</td>
<td></td>
</tr>
<tr>
<td>12</td>
<td></td>
<td></td>
</tr>
<tr>
<td>13</td>
<td></td>
<td></td>
</tr>
<tr>
<td>14</td>
<td></td>
<td></td>
</tr>
<tr>
<td>15</td>
<td></td>
<td></td>
</tr>
<tr>
<td>16</td>
<td></td>
<td></td>
</tr>
<tr>
<td>17</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### Notes:

Figure 8-10. Syncpoint Pseudocode

SW3135.3
Poisoned Messages

Notes:

If a message is retrieved from a queue in syncpoint and some data in that message contains some invalid data that causes a backout, the message that contains the bad data is now, once again, available on the input queue. This could 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 is incremented by one. It is now possible to check that BackoutCount when the next MQGET is issued and to perform some type of error handling routine if it is greater than zero.

There are also some attributes associated with the queue definition. On queue managers other than AS/400, the BOTHRESH (BackoutThreshold) can be set to some value. There is also a BOQNAME (BackoutRequeueQName) attribute. The application could inquire of these attributes (using MQINQ) and then, compare the value in the BackoutCount of the message descriptor. If greater than the threshold, the application could re-route 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 UOW, including DB updates

Original message available in new transaction

MQPUT reply message
SYNCPOINT

Notes:

On z/OS only, it is possible to avoid rolling back a message that has poisoned data. If an input queue holds the potential for containing messages with bad data, use the Get Message Options option of MQGMO_MARK_SKIP_BACKOUT. Only one message per logical unit of work can have this option specified.

If the program determines there is a need to do a rollback, the message that was retrieved using the MQGMO_MARK_SKIP_BACKOUT will not be made available on the input queue and have the delete flag turned off. Rather, all other resources involved in the unit of work will be rolled back leaving the poisoned message as the sole participant in its very 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 and making it available for exception processing on the error queue.
Remote Updates

Notes:
When WebSphere MQ applications are distributed across more than one platform, Time Independence means that there is no such thing as a traditional distributed unit of work.

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 philosophy must be one of an optimist; the message will be successfully delivered and the expected response will be returned. In cases when this does not happen, the application will need to include some exception handling. This is called compensating transactions. For example, I offer the last seat on a plane to my customer. I then send the confirm over to the airline’s system only to be informed the seat is already gone. I will need to offer an alternative to my customer to compensate for the one I thought we might get.

What is the MINIMUM number of units of work needed to send a message and receive a response if ALL work was done using unit of work processing (assuming the above picture)?
Coordination Choices

- AIX, HP-UX, Sun Solaris
  - TXSeries
  - Tuxedo
  - WebSphere Application Server
- Linux
  - WebSphere Application Server
- Windows
  - WebSphere Application Server
  - TXSeries
  - Tuxedo
  - MTS/COM+
- z/OS
  - CICS
  - IMS
  - RRS
- iSeries
  - WebSphere Application Server
- Compaq NSK
  - NonStop TM/MP (TMF)
- AT&T GIS, SINIX, DC/OSx
  - Tuxedo

Notes:
The currently cooperating XA-compliant transaction coordination products that have been tested and are supported with WebSphere MQ are listed here. However, for the very latest information, you should check the WebSphere MQ Web site:
## Rollback Pseudocode

```
DEFINE GET_OPTIONS AS MQGM
DEFINE PUT_OPTIONS AS MQPMO
DEFINE ERROR_OBJDESC AS MQOD
CALL MQCONN(......,

CALL MQOPEN(......,

CALL MQINQ ( . . . /* BACKOUT_THRESHOLD BACKOUT_REQ_Q_NAME */
GET_OPTIONS.Options = MQGMO_NO_WAIT +
  MQGMO_SYNCPOINT + MQGMO_FAIL_IF_QUIESCING
CALL MQGET(......,
IF MQMD.BackoutCount > SAVED_BACKOUT_THRESHOLD
  /* handle message as error and commit */
  PUT_OPTIONS.Options = MQPMO_SYNCPOINT + MQPMO_FAIL_IF_QUIESCING
  ERROR_OBJDESC.OBJECTNAME = SAVED_BACKOUT_REQ_Q_NAME
  CALL MQPUT1(......,
  CALL MQCMIT ( . . . ,
ELSE
  CONTINUE . . .
IF REASON NOT = MQRC_NONE
  CALL MQBACK ( . . . ,
ELSE
  CALL MQCMIT ( . . . ,
```

### Notes:

For all platforms including z/OS and where Mark Skip Backout does not help (abends), notice the use of MQINQ to obtain back out threshold and back out requeue queue.
Notes:

A client application can participate in a unit of work that is managed by a queue manager to which it is connected. Within the unit of work, the client application can put messages to, and get messages from, the queues that are owned by that queue manager. The client application can then use the MQCMIT call to commit the unit of work or the MQBACK call to back out the unit of work. However, within the same unit of work, the client application cannot update the resources of another resource manager, the tables of a DB2 database, for example. Using a WebSphere MQ extended transactional client removes this restriction.

A WebSphere MQ extended transactional client is a WebSphere MQ client with some additional function. This function allows a client application, within the same unit of work:

- To put messages to, and get messages from, queues that are owned by the queue manager to which it is connected
- To update the resources of a resource manager other than a WebSphere MQ queue manager
This unit of work must be managed by an external transaction manager that is running on the same system as the client application. The unit of work cannot be managed by the queue manager to which the client application is connected. This means that the queue manager can act only as a resource manager, not as a transaction manager. It also means that the client application can commit or back out the unit of work using only the application programming interface (API) provided by the external transaction manager. The client application cannot, therefore, use the MQI calls, MQBEGIN, MQCMIT, and MQBACK.
Extended Transaction Managers

<table>
<thead>
<tr>
<th>Extended transactional client Platform</th>
<th>Supported external transaction managers</th>
</tr>
</thead>
<tbody>
<tr>
<td>IBM AIX, Sun Solaris, HP-UX</td>
<td>• Tuxedo</td>
</tr>
<tr>
<td></td>
<td>• TXSeries</td>
</tr>
<tr>
<td></td>
<td>• WebSphere Application Server</td>
</tr>
<tr>
<td>Windows NT/2000/XP</td>
<td>• Tuxedo</td>
</tr>
<tr>
<td></td>
<td>• TXSeries</td>
</tr>
<tr>
<td></td>
<td>• WebSphere Application Server</td>
</tr>
<tr>
<td></td>
<td>• MTS/COM+</td>
</tr>
<tr>
<td>Linux (on Intel and on z/OS)</td>
<td>• Websphere Application Server</td>
</tr>
</tbody>
</table>

Notes:

WebSphere MQ extended transactional clients are available for the platforms listed. A client application that is using an extended transactional client can connect to a queue manager of the following WebSphere MQ Version 5.3 products only:

- WebSphere MQ for AIX
- WebSphere MQ for HP-UX
- WebSphere MQ for iSeries
- WebSphere MQ for Linux for Intel
- WebSphere MQ for Linux for zSeries
- WebSphere MQ for Solaris
- WebSphere MQ for Windows
8.2 Triggering
**Notes:**

In the visual, what happens above the dotted line is application code. Below is the responsibility of the queue manager and a supplied utility called a Trigger Monitor.

1. Program A MQPUTs a message to the application queue.
2. The queue manager recognizes that the application queue is defined as a triggered queue and references the process specified in the queue definition.
3. 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 be started), the queue manager builds a special message called a trigger message...
4. ... and places it on the initiation queue.
5. A constantly running application called a trigger monitor retrieves the trigger message, obtains the name of the application to be started from within that message...
6. ... and starts the program to be triggered (Program B in this case), passing it the name of the application queue.
Trigger Types

- **First**
  - TriggerMsgPriority

- **Every**
  - TriggerMsgPriority

- **Depth**
  - TriggerDepth
  - TriggerMsgPriority

**Notes:**

It is possible to set up a queue for triggering when the first message arrives (TRIGTYPE=FIRST), when some fixed number of messages arrives (TRIGTYPE=DEPTH, TRIGDPTH=3), or whenever a new message arrives (TRIGTYPE=EVERY). The conditions for triggering are set up as part of the queue definition, so the WebSphere MQ administrator will need to be involved in any design that requires triggering.
**Trigger Type Depth**

**Notes:**

When the trigger conditions are satisfied on the triggered queue, the queue manager creates a trigger message and disables triggering on that queue using the TriggerControl attribute. The triggered application must re-enable triggering itself by using the MQSET call.

To avoid a triggering yo-yo effect, MQGET should have a WAITINTERVAL to see if more arrive before turning trigger on and exiting.
**Notes:**

When a trigger monitor retrieves a trigger message (MQTM) from an initiation queue, the trigger monitor may need to pass some or all of the information in the trigger message to the application that is started by the trigger monitor. Information that may be needed by the started application includes QName, TriggerData, and UserData. The trigger monitor application can pass the MQTM structure directly to the started application.

By passing the QName to the triggered program, you can design a generic program capable of being triggered by different queues. 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.

There are three different trigger message formats, MQTM, MQTMC and MQTMC2. All contain the fields displayed on the visual. 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 there is one additional field,
QMgrName. MQTMC is only used by the AS/400. MQTM and MQTMC2 can be used by any platform.
## Triggered Application Pseudocode

```plaintext
DEFINE OBJ_DESC AS MQOD
DEFINE TRIG AS MQTMC2
DEFINE CONN_HANDLE AS MQHCONN
DEFINE OBJ_HANDLE AS MQHOBJ
COPY INPUTPARAMETER TO TRIG
QMNAME = TRIG.QMgrName
QNAME = TRIG.QName
CALL MQCONN(QMNAME, CONN_HANDLE, ...)
OBJ_DESC.ObjectName = QNAME
CALL MQOPEN(CONN_HANDLE, OBJ_DESC, ..., OBJ_HANDLE, ...)
CALL MQGET(CONN_HANDLE, OBJ_HANDLE, ...)
```

**Notes:**

---

Figure 8-22. Triggered Application Pseudocode

© Copyright IBM Corp. 1997, 2004

Unit 8. Transaction Support and Triggering

Course materials may not be reproduced in whole or in part without the prior written permission of IBM.
**Triggering and Syncpoint**

![Diagram of Triggering and Syncpoint]

**Notes:**

If a message is physically placed on a queue, it has the potential to satisfy trigger conditions associated with the queue. For example, if a message is put in syncpoint and is the first message placed on the queue, if TRIGTYPE=FIRST and triggering is enabled for the queue, a trigger message will be built by the queue manager and placed on the appropriate initiation queue. However, the trigger message will also be put in syncpoint. This can delay the start of an application to process the messages which may be what is desired. But, consider a queue that has a mix of messages put in syncpoint and without syncpoint. The subsequent messages put without syncpoint will not be processed until the trigger message is made available to be received by the trigger monitor.

When any syncpoint occurs, whether a commit or rollback, the message in the initiation queue will be committed and made available to the trigger monitor. This may result in an application starting and finding no messages available (if they were all rolled back).
Checkpoint

1. What happens to a message destructively gotten in syncpoint if the program issues an MQBACK call?

2. How can you tell how many times a message has already been backed out?

3. What are the three types of triggering?

Notes:
Please write your answers here.
1.
2.
3.
Unit Summary

- Use MQGMO_, MQPMO_, SYNCPOINT or NO_SYNCPOINT to control message deletion and delivery
- Use MQCMIT and MQBACK if no transaction manager available
- Use triggering to create event-driven programs

Notes:

We have discussed how to use the MQGMO_SYNCPOINT, MQPMO_SYNCPOINT, MQGMO_NO_SYNCPOINT and MQPMO_NO_SYNCPOINT options and what effect each has on message deletion and delivery.

The MQCMIT and MQBACK calls are available for use when no transaction manager is available. On some platforms, the MQBEGIN call allows the inclusion of certain database updates in a global unit of work that is coordinated by WebSphere MQ.

Finally, the ability to trigger a program based on message arrival on a queue lets you design event-driven applications. And by using the contents of the trigger message in the triggered program, you can create generic programs capable of being triggered by different queues.
Exercise 6

Do Lab Exercise 6

Notes:
Unit 9. MQI Security

What This Unit Is About

This unit focuses on the security capabilities that have been built into the MQI calls. The unit covers the context fields of the message descriptor, as well as alternate user authority.

What You Should Be Able to Do

After completing this unit, you should be able to:

• Understand the context fields of the message descriptor
• Save the context fields in a program and then pass those fields on to another message
• Use alternate user authority to allow one user to send a message on behalf of another

How You Will Check Your Progress

Accountability:

• Classroom discussion
• Machine exercise

References

SC34-6064  WebSphere MQ Application Programming Guide
SC34-6062  WebSphere MQ Application Programming Reference
Unit Objectives

- Understand the context fields of the message descriptor
- Save the context fields in a program and then pass those fields on to another message
- Use alternate user authority to allow one user to send a message on behalf of another

Notes:

Programmers may deal with WebSphere MQ security through two capabilities - the context fields of the message descriptor and the alternate user authority. In this unit, we will discuss those capabilities and how they can alter a user's ability to access WebSphere MQ objects.
9.1 MQI Security
Message Context

Notes:

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

- Check the authority of the sending application,
- Perform some accounting functions (that is, charge back) using the initiator's information,
- Keep an audit trail of messages that have been 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: the Identity data and the Origin data.
Context

DEFAULT CONTEXT

Program

Queue manager creates context information

PASS_ALL_CONTEXT

Program

Queue manager moves all context information

PASS_IDENTITY_CONTEXT

Program

Queue manager moves Identity context information and creates origin context

Notes:

Typically, application programs will not explicitly update the context fields. The default behavior will be that the queue manager sets all of the fields. Usually an application will not be authorized to update the context fields.

In the case of an application that is simply passing a message along, it is possible (if authorized) to pass context information from an input message to an output message.

In some cases, only the Identity data portion of the context information is passed or updated. Again, an application and user must be authorized to do this.

Normally, business applications will not concern themselves with the context fields; the queue manager will update them as we shall next examine.
Default Context

- UserIdentifier
- AccountingToken
  - MVS CICS
    - Unit of work identifier (UEPUOWDS)
  - IMS
    - PSB name + IMS recovery token
  - OS/400
    - 1 byte length + job accounting code
  - UNIX
    - User number
- ApplIdentityData
  - Blank
- PutApplType
  - CICS, UNIX, and so forth
- PutApplName
- PutDate
  - YYYYMMDD
- PutTime
  - HHMMSTH
- ApplOriginData
  - Blank

Notes:

The eight fields that make up context are further divided into two parts. When an application issues an MQPUT, if no actions have been taken to update the context fields, the queue manager, by default, will update each of the fields as shown.

It is important to note that UserIdentifier is 12 characters. In some environments, the userid can be larger (for example, the ADMINISTRATOR userid in Windows NT). Only the first 12 characters of a large userid will be stored in the UserIdentifier field. A Windows NT security identifier (SID) is stored in the AccountingToken field when a message is created under WebSphere MQ for Windows NT. The SID can be used to supplement the UserIdentifier field and to establish the credentials of a user.

The PutApplName field is 28 characters long. If the path and application name are longer, the last 28 characters is used. Note that the PutTime is in GMT.
No Context

- Option on MQPUT
  - MQPMO_NO_CONTEXT
  - The queue manager clears all the context fields

- Can be tested after MQGET
  - Is PutApplType = MQAT_NO_CONTEXT?

- Default and No Context require no more authority than to put a message on the target queue

Notes:

If the PutMsgOption.Option of MQPMO_NO_CONTEXT is specified, any data in any of the context fields is cleared by the queue manager at MQPUT time. Since the queue manager does not have to determine values to be filled in, this may be slightly faster.

No special authority is required to use No Context.

The only time the PutApplType is blank will be if the message that is retrieved was put with a No Context specification.
Notes:

In the visual, various users (A1, A2, A3) initiate a business transaction by putting a message on QUEUE1. The server program B gets the messages from QUEUE1, enriches them with data from the departmental data base and passes them on to QUEUE 2. Program D gets the messages from QUEUE 2 and updates the corporate data base accordingly.

If D wishes to verify the requestor’s ID before updating the corporate data base, the context information in the QUEUE 2 messages is of no use - it will show that each message has been created by the user under whose ID program B is running. The solution is for B to pass context information from QUEUE 1 to QUEUE2.

This is done by using an Open Option on QUEUE1 of Save All Context, an Open Option on QUEUE2 of Pass Identity Context and the Pass Identity Context option on the MQPUT to QUEUE2. The UserIdentifier, AccountingToken and ApplIdentityData will be passed from the input message to the output message. However, because the request was Pass Identity Context, not Pass All Context, the last five fields (the Origin data) will be updated by the queue manager.
Context Handling

**Input Queue**

- **Default**
  - ID: ORIGIN
  - QMgr: PROGRAM

- **Pass Identity**
  - ID: ORIGIN
  - QMgr: PROGRAM

- **PASS ALL**
  - ID: ORIGIN
  - QMgr: PROGRAM

- **SET Identity**
  - ID: ORIGIN
  - QMgr: PROGRAM

- **SET ALL**
  - ID: ORIGIN
  - QMgr: PROGRAM

**Output Queue**

- ID: ORIGIN

**Notes:**

Putting programs can stipulate how the context fields should be filled in. Let's review what the possible options might be:

- **MQPMO_DEFAULT_CONTEXT** - The queue manager is to supply both the Identity context and the Origin context data.

- **MQPMO_PASS_IDENTITY_CONTEXT** - The queue manager is to supply the Origin context but the Identity context is to be copied from the input message. This 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_IDENTITYCONTEXT.

- **MQPMO_PASS_ALL_CONTEXT** - All eight of the context fields are to be passed from the input to the output message. Again, the proper Open Options are required for the input and output queues.
• MQPMO_SET_IDENTITY_CONTEXT - The queue manager is to supply the Origin context but the application is to set the Identity context. The queue must be opened for output using the additional Open Option MQOO_SET_IDENTITY_CONTEXT.

• MQPMO_SET_ALL_CONTEXT - The program will control the content of all context fields.

Except for the first, the options reviewed will only succeed if the user and application attempting them have the appropriate authority.

A question - If you want to pass Identity context from a message that your application retrieves from QUEUE1 BUT, you have three queues opened for input (QUEUE1, QUEUEA and QUEUBE), how do you tell the queue manager which context information to pass?
Programs That Set Context

- Two options that require additional authority
  - MQPMO_SET_INDENTITY_CONTEXT
  - MQPMO_SET_ALLCONTEXT

- Used only by special programs
  - Message channel agents
  - System utilities
  - And so forth

Notes:
As stated earlier, in addition to using the Open Option MQOO_OUTPUT, if a program is to set any of the context fields, it will be necessary to add in the appropriate Open Option as shown above.

In most cases, business applications will not use the SET_CONTEXT options.
Pass Context Pseudocode (1 of 2)

<table>
<thead>
<tr>
<th>Define</th>
<th>Description</th>
<th>Line</th>
</tr>
</thead>
<tbody>
<tr>
<td>OPEN_OPTIONS</td>
<td>MQLONG</td>
<td>1</td>
</tr>
<tr>
<td>MY_IN_HANDLE</td>
<td>MQOBJ</td>
<td>2</td>
</tr>
<tr>
<td>MY_OUT_HANDLE</td>
<td>MQOBJ</td>
<td>3</td>
</tr>
<tr>
<td>MY_INPUT_QUEUE</td>
<td>MQOD</td>
<td>4</td>
</tr>
<tr>
<td>MY_OUTPUT_QUEUE</td>
<td>MQOD</td>
<td>5</td>
</tr>
<tr>
<td>MESSAGE_DESCRIPTOR</td>
<td>MQMD</td>
<td>6</td>
</tr>
<tr>
<td>PUT_OPTIONS</td>
<td>MQPMO</td>
<td>7</td>
</tr>
<tr>
<td>CALL MQCONN(QMGR,</td>
<td></td>
<td>9</td>
</tr>
<tr>
<td>CONN_HANDLE,</td>
<td></td>
<td>10</td>
</tr>
<tr>
<td>COMP_CODE,</td>
<td></td>
<td>11</td>
</tr>
<tr>
<td>REASON_CODE)</td>
<td></td>
<td>12</td>
</tr>
<tr>
<td>OPEN OPTIONS=MQOO_SAVE_ALL_CONTEXT +</td>
<td></td>
<td>14</td>
</tr>
<tr>
<td>MQOO_INPUT_AS_Q_DEF</td>
<td></td>
<td>15</td>
</tr>
<tr>
<td>MY_INPUT_QUEUE.ObjectName='INP1'</td>
<td></td>
<td>17</td>
</tr>
</tbody>
</table>

**Notes:**
### Pass Context Pseudocode (2 of 2)

```c
CALL MQOPEN(CONN_HANDLE,
    MY_INPUT_QUEUE,
    OPEN_OPTIONS,
    MY_IN_HANDLE,
    COMP_CODE,
    REASON_CODE)
OPEN_OPTIONS=MQOO_PASS_ALL_CONTEXT +
    MQOO_OUTPUT
MY_OUTPUT_QUEUE.ObjectName='OUT1'
CALL MQOPEN(CONN_HANDLE,
    MY_OUTPUT_QUEUE,
    OPEN_OPTIONS,
    MY_OUT_HANDLE,
    COMP_CODE,
    REASON_CODE)
MQGET(......,
    PUT_OPTIONS.Options=MQPMO_PASS_ALL_CONTEXT
    PUT_OPTIONS.Context=MY_IN_HANDLE
CALL MQPUT(......,
```

**Notes:**
Alternate User Authority

**Notes:**

The above scenario is the same as the one we saw earlier when discussing passing context, but in this case we do not rely on application code in program D to decide whether to update the corporate data base. Instead we get the queue manager to prevent program B opening Queue 2 on behalf of users we do not trust. In this way the request gets no further.

AlternateUserAuthority enables program B to get a message from the input queue (QUEUE1) and move the UserIdentifier from the context information of the input message to a field in the object descriptor called MQOD_AlternateUserld. If an explicit MQOPEN is to be done, then the MQOO_ALTERNATE_USER_AUTHORITY is included in the Open Options or, if an MQPUT1 is to be used, the more likely choice, MQPMO_ALTERNATE_USER_AUTHORITY is included in the MQPMO_Options.

When the MQOPEN or MQPUT1 is executed, the authority of B is checked to see if it is authorized to request alternate user authority AND the authority of the userid specified in the MQOD_AlternateUserld is checked to see if it is authorized to open the queue for output.
Figure 9-12. Alternate UserId Pseudocode (1 of 2)

Notes:
Alternate UserId Pseudocode (2 of 2)

CALL MQGET(......, 21
22
MY_OUTPUT_QUEUE.AlternateUserId = 23
MESSAGE_DESCRIPTOR.UserIdentifier 24
OPEN_OPTIONS=MQOO_ALTERNATE_USER_AUTHORITY + 26
MQOO_OUTPUT 27
MY_OUTPUT_QUEUE.ObjectName='OUT1' 29
30
CALL MQOPEN(CONN_HANDLE, 31
32
MY_OUTPUT_QUEUE, 33
OPEN_OPTIONS, 34
MY_OUT_HANDLE, 35
COMP_CODE, 36
REASON_CODE)
Checkpoint

1. Name four of the context fields.

2. Which time zone is used in context information?

3. What field is used for the Alternate UserId open request?

Notes:
Please write your answers here.
1.
2.
3.
Unit Summary

- Context fields are contained within every message and can be used for security
- Alternate user authority can be used to allow one user to send messages on behalf of another user

Notes:

The authorization for users to perform these tasks must be set up by the WebSphere MQ administrator. He or she can give individual users or groups of users the authority to connect to the queue manager, and open various queues, processes, or namelists. If the application design requires the ability to pass security information or to pass messages on behalf of another user, you should work with your WebSphere MQ administrator to be sure that the appropriate user authorizations will be set.
Exercise 7

Do Lab Exercise 7

Notes:
Unit 10. Message Groups and Segmentation

What This Unit Is About

Message groups are regular WebSphere MQ messages that are logically grouped together. This function has been available on the distributed platforms since Version 5.0, and on z/OS since Version 5.3. Message segmentation is the ability to take very large WebSphere MQ messages and break them into smaller physical messages. This function is available only on the distributed platforms.

What You Should Be Able to Do

After completing this unit, you should be able to:

• Write a program that logically groups messages together
• Write a program that allows the queue manager to physically segment large messages
• Write a program that performs the physical segmentation

How You Will Check Your Progress

Accountability:

• Classroom discussion
• Machine exercise

References

SC34-6064  WebSphere MQ Application Programming Guide
SC34-6062  WebSphere MQ Application Programming Reference
Unit Objectives

- Understand how to group messages together logically
- Understand how to deal with very large messages by breaking them into smaller physical segments

Notes:

Before Version 5 of WebSphere MQ, grouping messages together logically was a difficult task. You had to add to your application code, and use fields such as the MsgId and CorrelId to tie the messages together.

And if you wanted to deal in messages larger than the maximum message length of a queue or queue manager, once again you had to write the application code to perform that function.

In Version 5, everything changed.
10.1 Message Groups and Message Segmentation
Physical and Logical Messages, Segments and Groups

Figure 10-2. Physical and Logical Messages, Segments and Groups

Notes:

Messages can occur in groups. This enables ordering of messages as well as segmentation of large messages within the same group.

Message segments will be discussed 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, make 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 actually 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, consisting of one or more physical messages that have some relationship. We will explore the possible relationships as we proceed.
Why?

- **Message groups**
  - To ensure ordering on retrieval where not already guaranteed
    - Avoid the use of MsgID and CorrelID
  - To allow an application to group related messages together
    - When messages must be processed by the same or particular server instance
    - Avoid the use of MsgID and CorrelID

- **Message segmentation**
  - To handle large messages
    - Too large for the queue manager or the queue
    - Too large for the application buffer

*Notes:*

By using the capabilities associated with message groups, a program can make sure that it retrieves messages in the proper order without having to make use of some combination of MsgId and CorrelId. Remember that we have said that, unless explicitly requested, MsgId and CorrelId are not used as part of the message retrieval process for the Version 5 products. Remember that the default setting is MQMO_MATCH_MSG_ID and MQMO_MATCH_CORREL_ID.

Also, it provides an alternative means to allow grouping of related messages, again without using MsgId and CorrelId.

As for segments, they are simply a means to handle messages that are too large for a program or queue manager (including an intermediate queue manager through which a message passes). Most queue managers (including z/OS) allow for messages up to 100 MB.
Message Groups

Figure 10-4. Message Groups

Notes:

If using Version 2 of the message descriptor structure (or the MQMDE as described in the Application Programming Reference), there are some fields that are used when working with message groups and segments. The first of these added 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 will have the same GroupId. The next new field is the MQMD_MsgSeqNumber; although the GroupId will be the same, each logical message will have a different MsgSeqNumber (the numbering starts at 1).

If a message is not part of a group, the GroupId will be all nulls while MsgSeqNumber will be set to 1.

Our example shows three logical messages in a single group called XYZ. Each has a different MsgSeqNumber (1, 2 and 3).
Grouping Logical Messages

**Notes:**

The recommended 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 new Put Message Options option - MQPMO_LOGICAL_ORDER. Version 2 of the message descriptor contains another new field called MsgFlags. By specifying MQPMO_LOGICAL_ORDER and setting MQMD.MsgFlags to either MQMF_MSG_IN_GROUP or MQMF_LAST_MSG_IN_GROUP, the queue manager will generate a new GroupId for the starting message of each new group, and will then keep that GroupId and assign a new MsgSeqNumber for each new logical message within the group.

The queue manager will know to create a new GroupId, if the last that it placed contained the MQMD.MsgFlag of MQMF_LAST_MSG_IN_GROUP. Since message groups imply that more than one message will be placed on an output queue, this automatic handling by the queue manager is only supported for MQPUT, not MQPUT1.

If the application wishes to use message groups with MQPUT1 calls or simply wishes to control the setting up of the message group, it would not use the
MQPMO_LOGICAL_ORDER option. Then, GroupId is handled in a manner similar to the way MsgId setting is handled on non-Version 5 platforms. If the application sets the GroupId to MQGI_NONE (nulls), a unique GroupId will be generated. It would be the application's responsibility to 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 desired.

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 will fail with MQRC_INCONSISTENT_PERSISTENCE. If MQPMO_LOGICAL_ORDER is not specified, it will be the responsibility of the application to determine that persistence be kept consistent, no check will be handled by the queue manager. This means that some messages in a group could be lost if a channel that permits fast messaging fails or a queue manager stops for any reason and the messages within the group have a mixture of persistent and non-persistent.
Retrieving Logical Messages (1 of 2)

- **MQGMO_VERSION_2**
- **MQGMO_ALL_MSGS_AVAILABLE**
  - Messages available for retrieval only when all messages in a group have arrived
- **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**
  - The application must select the correct GroupId, MsgSeqNumber
  - Can be used to restart in the middle of a group after a failure
    - Can then switch to MQGMO_LOGICAL_ORDER
  - To be used for applications forwarding messages
- **z/OS requires INDXTYPE of GROUPID**

Figure 10-6. Retrieving Logical Messages (1 of 2)

**Notes:**

There is a Version 2 of the Get Message Options structure. It must be used to allow the proper retrieval of logical messages within a group.

By specifying MQGMO_ALL_MSGS_AVAILABLE options, the program can prevent retrieval of messages belonging to an incomplete group. By specifying MQGMO_LOGICAL_ORDER, messages in groups (and segments in logical messages) are returned in order. This order may be different from the physical order of messages on a queue.

If both MQGMO_ALL_MSGS_AVAILABLE and MQGMO_LOGICAL_ORDER are specified, the former will only have effect when there is no current group or logical message.

As with the MQPUT, the MQGET of messages in groups can be controlled by the application. However, this is not the recommended approach except in order to restart a group after a system or queue manager failure. The queue manager will then retain the group and segment information, and subsequent calls, using the same queue handle, can revert to using MQGMO_LOGICAL_ORDER.
Likewise, messages that are simply forwarding physical messages should NOT use the MQGMO_LOGICAL_ORDER option or the GroupId of the original message can be corrupted.

WebSphere MQ for z/OS now supports message grouping for shared and non-shared queues. This function is enabled by the introduction of a new INDXTYPE of GROUPID for local queues. 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

Notes:

The MatchOptions field of the Version 2 Get Message Options structure allows a program to control which group it retrieves.

If an application wishes to retrieve a particular group, it can use the MQGMO_LOGICAL_ORDER option (as long as no current logical message is being processed) in conjunction 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 MessageSequenceNumber. As with the other MatchOptions, this is in addition to any other matches that may apply. The MQMO_MATCH_MSG_SEQ_NUM is not valid if combined with the MQGMO_LOGICAL_ORDER.
Finally, an application can determine if it has processed the final message in a group by checking the MQGMO.GroupStatus field in the Get Message Options structure after a message is retrieved. The MQGMO.GroupStatus field would contain a value represented by the symbolic MQGS_LAST_MSG_IN_GROUP.

If not, and a group was being processed, the value would be MQGS_MSG_IN_GROUP.
Spanning Units of Work

- **Putting application**
  - Some of the group may 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 has been successfully put
  - MQPMO_LOGICAL_ORDER cannot be used on the first PUT of the restart application

- **Getting application**
  - STATUS queue again
  - First GET of the restart application must match on GroupId and MsgSeqNumber

**Notes:**

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, they need not be put within the same unit of work. This permits spreading a message group or logical message that has many physical messages into smaller units of work.

- Conversely, if the first physical message has specified 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 will fail with MQRC_INCONSISTENT_UOW. The conditions 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 keep track of the status information associated with a group or logical messages that span units of work. By keeping track of the GroupId and the MsgSeqNumber on a special status queue (using syncpoint options for the MQPUT and MQGET), an accurate picture of what has been completed can be kept.

If a restart is done, this is the information that can be used, as discussed earlier, without specifying the Get or Put Message Options option for logical order to essentially restart proper 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
    - So every segment starts with a MQMD
- **A message can be segmented and reassembled**
  - By the queue manager
  - By an application

**Notes:**

Message segments have been defined at a high level in the previous topic. We now need to explore how to use them in a program. Note that these discussions will make an assumption that all MQPUT and MQGET work will operate within a unit of work. This is strongly recommended to be a standard practice in order to avoid any possibility of incomplete groups being present 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’s 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.
Segmentation by the Queue Manager

- **MQMF_SEGMENTATION_ALLOWED in MsgFlag of MQMD**
  - Version 2

- The queue manager does the following:
  - User-defined formats are split on 16-byte boundaries
  - The split never occurs in the middle of an MQSeries header
  - MQRO_..._WITH_DATA modified to MQRO_... in MQMD if the segment does not contain any of the first 100 bytes of the user message
  - No assumptions should be made on the way the queue manager splits a message

- Persistent message segmentation can be performed only within a unit of work
  - The queue manager creates a UOW if no user-defined UOW already exists
  - MQRC_UOW_NOT_AVAILABLE if a user-defined UOW exists and MQPMO_NO_SYNCPOINT is specified

**Notes:**

If the MsgFlag field in the Version 2 message descriptor includes the value represented as MQMF_SEGMENTATION_ALLOWED, the queue manager will handle building a segmented message. If the queue manager recognizes that the message will be too big for its MaxMsgLength (a queue manager attribute only supported on Version 5 queue managers) or for a queue's MaxMsgLength, segmentation will be done.

If the message descriptor included the MQRO_..._WITH_DATA, it will be 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.

Note the restrictions regarding unit of work processing and recall the strong recommendation that unit of work processing be the standard. For persistent messages, unit of work processing is required; if the queue manager determines that there is no application-defined unit of work active, the queue manager will create a unit of work that will only be active for the duration of the call.
Reassembly by the Queue Manager

- **MQGMO_VERSION_2**
- **MQGMO_COMPLETE_MESSAGE**
  - Only complete logical messages can be retrieved
  - If the segments have different MQMD values specified, values from the FIRST segment are returned
    - If different CodedCharSetId or encoding, MQCC_WARNING is returned and a partial message is retrieved
- Persistent message reassembly can be performed only within a unit of work
  - The queue manager creates a UOW if no user-defined UOW already exists
  - MQRC_UOW_NOT_AVAILABLE if a user-defined UOW exists and MQGMO_NO_SYNCPOINT is specified

**Notes:**

The getting application needs to include a Get Message Options option of MQGMO_COMPLETE_MESSAGE. If the queue manager determines that the message is segmented, it will retrieve all the segments and return the complete message in the program buffer.

Data conversion is done after the message is placed in the program's buffer so this should not change with message segments. However, data conversion requested by a sender channel will fail for a message segment since the exit will 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 there is no user-defined unit of work, the queue manager will create a unit of work for the duration of the call. It will NOT automatically create a unit of work for non-persistent messages. Note again the strong recommendation that all messages that have the potential for segmentation use unit-of-work processing.
Segmentation by the Application

Notes:

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. Data conversion must be performed by sender channels and the putting program needs to split the message on specific boundaries to enable successful conversion of the segments.

The application can include the Put Message Options option of MQPMO_LOGICAL_ORDER. The same caution applies to using unit-of-work processing. As each segment is built and the MQPUT is executed, the application needs to make sure the MsgFlags field in the message descriptor includes either MQMF_SEGMENT or MQMF_LAST_SEGMENT. The queue manager will assign and maintain the GroupId, MsgSeqNumber and Offset.
If the application does not specify the MQPMO_LOGICAL_ORDER, then the program is responsible for ensuring a new GroupId is assigned as discussed in the coverage of Message groups, as well as assigning proper MsgSeqNumbers and Offsets.

Putting and getting messages that span units of work is permitted as discussed in the Message Groups topic.
Reassembly by the Application (1 of 2)

- MQGMO_VERSION_2
- MQGMO_ALL_SEGMENTSAVAILABLE
  - Messages available for retrieval only when all segments have arrived
  - It is forced by MQGMO_ALL_MSGS_AVAILABLE
- 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
  - The 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
  - To be used for applications forwarding messages

Notes:

The use of the Version 2 Get Message Options structure is required. It is possible to specify the MQGMO_COMPLETE_MESSAGE option and the MQMF_SEGMENTATION_ALLOWED MsgFlag to enable queue manager retrieval of a message that was segmented by an application.

If the application chooses to retrieve the segments individually, most likely because of a buffer size constraint, then there is the capability to reassemble the message under application control.

If the program specifies MQGMO_ALL_MSGS_AVAILABLE or MQGMO_ALL_SEGMENTS_AVAILABLE as one of the Get Message Options options, processing will not occur if all message segments are not available. It makes sense to include the MQGMO_WAIT option to allow time for all segments to arrive (with a WaitInterval).

Once the first message is retrieved, use the MQGMO_LOGICAL_ORDER option to ensure all remaining segments for this logical message are processed.
Not specifying MQGMO_LOGICAL_ORDER can be used for recovery purposes as we previously discussed in our coverage of Message Groups. The only additional requirement is the need to keep track of the Offset of the segment as well as the GroupId and MsgSeqNumber.

Finally, any intermediate applications that are simply passing the data should not use the MQGMO_LOGICAL_ORDER option to ensure the Offset field is not corrupted.
Reassembly by the Application (2 of 2)

- **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
  - Only if MQGMO_LOGICAL_ORDER is not specified

- **SegmentStatus field in MQGMO**
  - Returned on the MQGET
  - MQSS_NOT_A_SEGMENT
  - MQSS_SEGMENT
  - MQSS_LAST_SEGMENT

**Notes:**

It is possible to control which segment is retrieved by a program as well as have total control over message retrieval by a program. The MatchOptions field of the Version 2 Get Message Options structure allows for this.

The following is very much like the process to retrieve a complete message group. Now, we want to learn how to retrieve a complete logical message.

If an application wishes to retrieve a particular logical message, it can begin retrieval of messages from the GroupId (using MatchOption MQMO_MATCH_GROUP_ID), the logical message (using MQMO_MATCH_MSG_SEQ_NUM) starting with MQMD.Offset set to zero, and including MQMO_MATCH_OFFSET in the MatchOptions of the Get Message options structure. The MQMO_MATCH_MSG_SEQ_NUM is not valid if combined with the MQGMO_LOGICAL_ORDER.
Finally, an application can determine if it has processed the final segment of a logical message by checking another field in the Get Message Options structure after a message is retrieved. The MQGMO.SegmentStatus field would contain a value represented by the symbolic MQSS_LAST_SEGMENT. If not, and a group was being processed, the value would be MQSS_SEGMENT. The GroupStatus field could be checked to see if all logical messages within the group had been processed.
**Notes:**

The sample message group will have three logical messages. Logical message 1 has three segments, logical message 2 has only one and logical message 3 has two segments.

The value in the Put Message Options structure’s 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 need only make sure that the MsgFlags field is properly set.

Once 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 nothing regarding segment, there is no need for the queue manager or application to be concerned 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 the MsgFlags field is properly set up.

The queue manager is running under a user-defined unit of work so none of this is actually available on the output queue until the application issues the MQCMIT.
Notes:

The application is going to reassemble the messages that it retrieves. So, the options for the Get Message Options structure include syncpointing, logical order and the combination of wait and all messages available which means that no segments will be retrieved until they all arrive.

The loop will execute 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, there is a user-defined unit of work so the queue manager will not do any unit of work of its own.
Segmented Messages and Reports

- A report message can be generated for EACH of the segments
- MQMD fields copied to queue manager generated reports
  - Groupld, MsgSeqNumber, Offset
  - MsgFlags
  - OriginalLength
    - Length 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 generated by a queue manager or MCA that does not support segmentation will not contain the segment information
  - Use MQRO_..._WITH_(FULL_)DATA

Notes:

Report messages require special processing by the queue manager.

If the MQGMO_ALL_SEGMENTS_AVAILABLE option is specified, the queue manager checks to see if 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 type of report messages present so there may be a mixture of report types.

By specifying MQGMO_COMPLETE_MSG, the queue manager checks to see if all the report messages of that report type relating to all the different segments are present. If so, it then retrieves the message. For this to be possible, either the report messages must be generated by a queue manager or MCA that supports segmentation or the originating application needs to request at least 100 bytes of message data (using MQRO_..._WITH_DATA or MQRO_..._WITH_FULL_DATA). If less than the full amount of application data is present for a segment, the missing bytes are replaced by nulls in the report message that is sent back.
Checkpoint

1. What advantages do Message Groups provide?

2. Where is the Group ID specified?

3. Which option would you use if you wanted to ensure that your application got messages only from complete groups?

Notes:
Please write your answers here.

1.
2.
3.
Unit Summary

- Message groups are used to tie together related messages
- Message segmentation allows a queue manager or application program to deal with very large messages

Notes:

WebSphere MQ message groups serve two purposes -- to tie messages together logically and to allow messages to be sent and received in the exact same logical order. Though this facility will not be used frequently, it is much easier to group messages together in this way, rather than using application code.

Message segmentation is also an uncommon need for most programs, but when you need the ability, this is a simple way to do it.
Exercise 8

Do Lab Exercise 8

Notes:
Unit 11.  Distribution Lists

What This Unit Is About

This unit focuses on another of the capabilities available on the distributed platforms since Version 5 - the ability to send a single message to a list of queues. This capability can simplify programming when the application requires that the same messages be sent to a group of users, for example, sending price list updates to a group of retail stores. Distribution Lists are on available on z/OS.

What You Should Be Able to Do

After completing this unit, you should be able to:

- Describe the use of distribution lists
- List the platforms that support distribution lists
- Implement a distribution list in a program

How You Will Check Your Progress

Accountability:

- Classroom discussion
- Machine exercise

References

SC34-6064  WebSphere MQ Application Programming Guide
SC34-6062  WebSphere MQ Application Programming Reference
Unit Objectives

- Describe the use of distribution lists
- List the platforms that support distribution lists
- Implement a distribution list in a program

Notes:

Distribution lists allow you to put a message to multiple destinations in a single MQPUT or MQPUT1 call. Multiple queues can be opened using a single MQOPEN and a message can then be put to each of those queues using a single MQPUT. Some generic information from the MQI structures used for this process can be superseded by specific information relating to the individual destinations included in the distribution list.

This ability to send a common message to a list of queues was added to WebSphere MQ when the distributed Version 5 queue managers were released. This function is still limited to those queue managers.
11.1 Distribution Lists
What Is a Distribution List?

Notes:

Prior to availability of the distributed Version 5 products, there was no true distribution list support in WebSphere MQ. Distribution lists allow you to put a message to multiple destinations in a single MQPUT or MQPUT1 call. Multiple queues can be opened using a single MQOPEN and then a single message can be put to each of those queues using the single MQPUT.

In our above example, a single open has been issued using a list of queues built by the application. One object handle is returned. So, the MQPUT can place the message on each of the queues using that single object handle.

Also, to keep message flows across networks at a minimum, the example demonstrates that one message on a transmission queue can be delivered to more than one remote queue.
Opening a Distribution List (1 of 3)

**Notes:**

Version 2 of the object descriptor (MQOD) contains fields that are used for distribution lists. An important addition is the Object Descriptor RecsPresent field. This is the number of Object Records (MQORs) provided by the application. If this field is greater than zero, it indicates that a distribution list is being opened. We will look at the Object Record structure in a moment.

The other two fields are mutually exclusive. Only one of the two may have a value greater than zero and, if using distribution lists, one MUST be greater than zero. The two fields are ObjectRecOffset and ObjectRecPtr:

- **ObjectRecOffset** is the offset of the first MQOR from the start of the Object Descriptor. This approach lends itself to programming languages that do not support the pointer data type or which implement the pointer data type in a way that is not portable between environments (an example is COBOL).

- The **ObjectRecPtr** allows the use of this pointer to the start of the array of MQORs. This method is recommended for programming languages that support the pointer data type in a portable manner (an example is C).
So the MQOD is set up to somehow point to an array of Object Records and the RecsPresent tells how many MQORs there are.

The application fills in the MQOR which consists of two fields - ObjectName and ObjectQMgrName. The ObjectName must be the name of a queue name and must NOT be the name of a model queue. The ObjectQMgrName behaves just as the ObjectQMgrName in the MQOD. Note that these MQODs are built by the application. Distribution lists are not a new WebSphere MQ object to be defined by the WebSphere MQ administrator.

So the MQOPEN is issued once the Object descriptor is set up and the Object records are built.
Opening a Distribution List (2 of 3)

Each destination queue is opened separately

Notes:
Each destination queue is opened separately, meaning some could succeed and others could fail if they are not valid. If all queues open successfully, the completion code returned from the MQOPEN will be MQCC_OK. If all the queues failed to open, it will be MQCC_FAILED. However, if some queues are successfully opened and some fail to open, the completion code will be MQCC_WARNING. In the last two cases, the reason code will be MQRC_MULTIPLE_REASONS. This brings us to the last fields in the Object Descriptor as well as another structure.
Notes:

Some other new fields in the Object Descriptor (MQOD) are useful for determining how many:

- Local queues were opened successfully (KnownDestCount).
- Remote queues opened successfully (UnknownDestCount).
- Queues failed to open successfully (InvalidDestCount).

The RecsPresent field is useful to tell us how many instances there are of a structure called Response Record (MQRR). The MQRR contains a completion code and reason code. They are used to receive the results of an open or put operation for a single destination queue. By providing an array of these MQRRs, it is possible to determine the completion codes and reason codes for each of the queues in the distribution list. The MQRRs would be set if the reason returned from the MQOPEN, MQPUT or MQPUT1 is MQRC_MULTIPLE_REASONS.

As with the MQOR array, the Object Descriptor has two ways to address the MQRR array. The ResponseRecOffset to define the offset of the start of the MQRR array or ResponseRecPtr to point to the address of the start of the MQRR array.
Adding a Message to a Distribution List (1 of 3)

Structure MQPMO
- MQPMO_VERSION_2
- PutMsgRecFields
- RecsPresent
- PutMsgRecOffset OR PutMsgRecPtr

Structure MQPMR
- Used to specify certain message properties for each destination queue individually.
  - MsgId, CorrelId, GroupId
  - Feedback
  - AccountingToken
- Fields present specified using
  - PutMsgRecFields
- Defaults taken from MsgDesc

MQPUT (Hconn, Hobj, MsgDesc, PutMsgOpts, BufferLength, Buffer, CompCode, Reason)

Notes:
Within Version 2 of the Put Message Options structure, there are several fields that are used in relation to distribution lists. We will start by looking at PutMsgRecFields, RecsPresent, PutMsgRecOffset and PutMsgRecPtr. Some of these may sound familiar because the Object Descriptor has similar fields. A brief look at their meanings follows:

- PutMsgRecFields includes flags indicating which fields from a new structure (the Put Message Record) are present. One or more of the flags can be specified:
  - MQPMRF_MSG_ID - Message-identifier field
  - MQPMRF_CORREL_ID - Correlation-identifier field
  - MQPMRF_GROUP_ID - Group-identifier field
  - MQPMRF_FEEDBACK - Feedback field
  - MQPMRF_ACCOUNTING_TOKEN - Accounting-token field
  - MQPMRF_NONE - No put-message record fields are present

- RecsPresent is the number of put message records or response records present (in this case, it is the former).
• PutMsgRecOffset is the offset of the first put message record from the beginning of the Put Message Options structure.

• PutMsgRecPtr is the address of the first put message record.

The rules regarding PutMsgRecOffset and PutMsgRecPtr are like those of the similar fields in the Object descriptor. One or the other can have a value but not both. However, since put message records are optional, it is possible for both to have a value of zero.

The Put Message Record structure (MQPMR) can contain up to five fields:

• MsgId - The message identifier to be used for the message sent to the queue whose name was specified by the corresponding element in the array of MQOR structures provided on the MQOPEN or MQPUT1 call. For those MQPMRs that do not contain a MsgId, the value from the MQMD MsgId field is used. If that value is MQMI_NONE, a unique MsgId is generated for each of those destinations (no two destinations will have the same MsgId). If MQPMO_NEW_MSG_ID is specified, a new MsgId will be generated for each of the destinations on the distribution list, regardless of whether they have MQPMR records.

• CorrelId - Correlation identifier is used for the message sent to the queue whose name was specified by the corresponding element in the array of the MQOR structures provided on MQOPEN or MQPUT1. This MQMD CorrelId is used if this field is not present or there are fewer MQPMRs than destinations. If MQPMO_NEW_CORREL_ID is specified, a SINGLE new correlation identifier is generated and used for all of the destinations in the distribution list.

• GroupId - This is the group identifier for the message sent to the queue whose name is in the corresponding element in the array of MQOR structures provided on the MQOPEN or MQPUT1. As with MsgId and CorrelId, the MQMD GroupId will be used under those conditions.

• Feedback - Used for the message sent to the queue whose name was specified by the corresponding element in the array of MQOR structures from the MQOPEN or MQPUT1. If not present, the MQMD value is used.

• AccountingToken - Again, used for the message sent to the corresponding queue in the MQOR array from the MQOPEN or MQPUT1. If not present, the MQMD value is used.
Adding a Message to a Distribution List (2 of 3)

**Notes:**

As described in the MQOPEN, the same applies for the MQPUT. The MQCC_OK will only be issued if ALL the puts succeeded, the MQCC_FAILED only if ALL puts failed and the MQCC_WARNING is some succeeded and some failed. If MQCC_WARNING or MQCC_FAILED is returned, the reason will be MQRC_MULTIPLE_REASONS.

The MQRR (response record) for each queue would contain the completion code and reason for the put to that queue. Since a separate MQPUT is issued to each queue, it is possible to keep track of each one separately.
Adding a Message to a Distribution List (3 of 3)

**Notes:**

Some other fields in the Version 2 Put Message Options structure (MQPMO) allow the application to determine how many:

- Messages were sent to local queues successfully (KnownDestCount)
- Messages were sent to remote queues successfully (UnknownDestCount)
- Messages could not be sent successfully (InvalidDestCount)

The RecsPresent field is useful to tell us how many instances of a structure called Response Record (MQRR) there are. The MQRR contains a completion code and reason code. They are used to receive the results of an open or put operation for a single destination queue. By providing an array of these MQRRs, it is possible to determine the completion codes and reason codes for each of the queues in the distribution list. The MQRRs would be set if the reason returned from the MQOPEN, MQPUT or MQPUT1 is MQRC_MULTIPLE_REASONS.

As with the MQPMR array, the Put Message Options structure has two ways to address the MQRR array. The ResponseRecOffset to define the offset of the start of the MQRR array or ResponseRecPtr to point to the address of the start of the MQRR array.
Closing a Distribution List

**Notes:**

When closing a distribution list, any close options other than MQCO_NONE will result in MQRC_OPTION_NOT_VALID_FOR_TYPE or MQRC_OPTIONS_ERROR.

Only the completion and reason codes from the MQCLOSE are available for diagnostic purposes. Even though a failure may occur, the queue manager will continue processing, and will attempt to close the remaining queues in the distribution list.
Remote Queues and Distribution Lists (1 of 3)

**Notes:**

When an application puts a message to a distribution list and some or all of the destinations are remote, the queue manager prefixes the message with two headers, the Transmission Header (MQXQH) and the Distribution Header (MQDH), and places the message on the appropriate transmission queues. If more than one transmission queue is involved, the MQDH structures in those messages identify different subsets of destinations.

Applications should not attempt to build MQDHs just as they normally would not build MQXQHs. The queue manager handles this on behalf of the application.

The layout of the MQDH header is interesting:

- **StrucId** - Contains the value DH
- **Version** - Contains constant value of 1
- **StrucLength** - The number of bytes from the start of the MQDH to the start of the message data (this number includes any MQORs and MQPMRs)
- **Encoding** - For encoding of the message data
- **CodedCharSetId** - Coded character set identifier of the message data
- **Format** - The format name of the message data

* This is transparent to the programmer.
• Flags - Can be none or MQDFH_NEW_MSG_ID to enable generation of a new message identifier for each destination in the distribution list
• PutMsgrecFields - Flags indicating which MQPMR fields are present
• RecsPresent - Number of MQORs present
• ObjectRecOffset - Offset of first MQOR from beginning of MQDH
• PutMsgRecOffset - Offset of first MQPMR from start of MQDH
Notes:

The DISTL(YES) shown in the above picture is actually an attribute on the transmission queue that is set by the MCA to inform the local queue manager whether the queue manager at the other end supports distribution lists. The attribute is set by the sending MCA whenever it connects with the receiving MCA on the adjacent remote queue manager.

If DISTL is YES, distribution list messages can be stored on the transmission queue and sent to the other side of the channel in that form, reducing the amount of processing and traffic needed to send messages to multiple destinations.

If DISTL is NO and the application puts a distribution list message, and there are multiple destinations using a single transmission queue, the queue manager will split the distribution list message and place the individual messages on the transmission queue instead. This increases the amount of processing to send the message to multiple destinations but ensures that the message will be handled correctly.

Although transparent to the application, it is worth knowing this might happen in case of a need to look at the messages on a transmission queue in the course of debugging.
Remote Queues and Distribution Lists (3 of 3)

- Distlists XmitQ attribute
  - DISTL=YES defines if distribution lists supported by partner
  - Set (overridden) by the sending MCA at bind time
  - Don't specify it unless partner capability is known
    - Can be useful with SERVER time MCA
  - Sending MCA will handle mistakes (wrong YES setting)
    - Unpack into individual messages and put them back to XmitQ

- Priority and Persistence
  - Different distribution list messages if different settings for different destinations

- If a distribution list message too big for the XmitQ, it is split into smaller ones

Notes:

Although most of this has been covered, note the item on Priority and Persistence. The individual queues each can have their own default settings. It is possible that some definitions would have DEFPRTY(4) while others may have DEFPRTY(2), and still others have DEFPRTY set to some other integer from zero through nine. If the application allows the Priority field in the MQMD to be MQPRI_PRIORITY_AS_Q_DEF, the different messages will need to have the appropriate priorities set.

The same is true for DEFPSIST. It can be set to YES or NO. If the application allows the default (MQPER_PERSISTENCE_AS_Q_DEF) in the Persistence field of the MQMD, the resulting value can be different for different destination queues.

Since the programmer controls whether the values in these fields use defaults or are explicitly set, you should be aware of the possibility of mixed results.
### Distribution List Pseudocode (1 of 2)

<table>
<thead>
<tr>
<th>Define/Declare</th>
<th>Line</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>DEFINE CONN_HANDLE AS MQCONN</code></td>
<td>1</td>
</tr>
<tr>
<td><code>DEFINE COMP_CODE AS MQLONG</code></td>
<td>2</td>
</tr>
<tr>
<td><code>DEFINE REASON_CODE AS MQLONG</code></td>
<td>3</td>
</tr>
<tr>
<td><code>DEFINE OD_DISTL AS MQOD</code></td>
<td>4</td>
</tr>
<tr>
<td><code>DEFINE DISTLIST AS ARRAY 2 MQPOR</code></td>
<td>5</td>
</tr>
<tr>
<td><code>DEFINE OPEN_OPTIONS AS MQLONG</code></td>
<td>6</td>
</tr>
<tr>
<td><code>DEFINE MY_OUT_HANDLE AS MQLONG</code></td>
<td>7</td>
</tr>
<tr>
<td><code>DEFINE MESSAGE_DESCRIPTOR AS MQMD</code></td>
<td>8</td>
</tr>
<tr>
<td><code>DEFINE PUT_OPTIONS AS MQPMO</code></td>
<td>9</td>
</tr>
<tr>
<td><code>DEFINE MESSAGE_LENGTH AS MQLONG</code></td>
<td>10</td>
</tr>
<tr>
<td><code>DEFINE MESSAGE AS MQCHAR1000</code></td>
<td>11</td>
</tr>
<tr>
<td><code>DEFINE RESPONSE_RECORD AS ARRAY 2 MQRR</code></td>
<td>12</td>
</tr>
<tr>
<td><code>MQCONN(....)</code></td>
<td>13</td>
</tr>
<tr>
<td><code>OD_DISTL.VERSION = MQOD_VERSION_2</code></td>
<td>14</td>
</tr>
<tr>
<td><code>OD_DISTL.RECSPRESENT = 2</code></td>
<td>15</td>
</tr>
<tr>
<td><code>OD_DISTL.OBJECTRECPtr = DISTLIST</code></td>
<td>16</td>
</tr>
<tr>
<td><code>OD_DISTL.RESPONSERECPtr = RESPONSE_RECORD</code></td>
<td>17</td>
</tr>
<tr>
<td><code>DISTLIST(1).OBJECTNAME = QUEUE1</code></td>
<td>18</td>
</tr>
<tr>
<td><code>DISTLIST(1).OBJECTQMGRNAME = QM1</code></td>
<td>19</td>
</tr>
<tr>
<td><code>DISTLIST(2).OBJECTNAME = QUEUE2</code></td>
<td>20</td>
</tr>
<tr>
<td><code>DISTLIST(2).OBJECTQMGRNAME = QM2</code></td>
<td>21</td>
</tr>
<tr>
<td><code>OPEN OPTIONS = MQOO OUTPUT</code></td>
<td>22</td>
</tr>
</tbody>
</table>

**Notes:**
Distribution List Pseudocode (2 of 2)

```python
MQOPEN(CONN_HANDLE, 27
  OD_DISTL, 28
  OPEN_OPTIONS, 29
  MY_OUT_HANDLE, 30
  COMP_CODE, 31
  REASON_CODE) 32

IF REASON_CODE = MQRC_MULTIPLE_REASONS 34
  Check RESPONSE_RECORD 35

PUT_OPTIONS.VERSION = MQPMO_VERSION_2 37
PUT_OPTIONS.RECSPRESENT = 2 38
PUT_OPTIONS.RESPONSERECPR = RESPONSE_RECORD 39

MQPUT(CONN_HANDLE, 41
  MY_OUT_HANDLE, 42
  MESSAGE_DESCRIPTOR, 43
  PUT_OPTIONS, 44
  MESSAGE_LENGTH, 45
  MESSAGE, 46
  COMP_CODE, 47
  REASON_CODE) 48

IF REASON_CODE = MQRC_MULTIPLE_REASONS 50
  Check RESPONSE_RECORD 51
```

Figure 11-14. Distribution List Pseudocode (2 of 2)  SW3135.3

Notes:
Checkpoint

1. Which field in Object Descriptor indicates whether a distribution list is being opened?

2. Which Reason Code is returned when some queues in the distribution list have opened successfully and some have not?

Notes:
Please write your answers here.

1.
2.
Unit Summary

• Distribution lists provide a simple and easy method for broadcasting messages to multiple destinations

• Only the sender of the message must be a Version 5 queue manager -- the receivers can be at any level

Notes:
Exercise 9

Do Lab Exercise 9

Notes:
Appendix A. Checkpoint Solutions

Unit 1: Overview

1. What is the name of the header which accompanies every message?
   The Message Descriptor or MQMD

2. How many MQI calls are there?
   13

3. Name four structures which are commonly used in MQI programs.
   MQMD, MQOD, MQPMO, MQGMO

Unit 2: Major Calls - Housekeeping

1. What are the three possible values of the Completion Code?
   MQCC_OK, MQCC_WARNING, MQCC_FAILED

2. How can an application be sure that no other applications are getting messages from a queue at the same time as you?
   Use the open option MQOO-INPUT_EXCLUSIVE

3. What's the difference in syntax between an MQCONN and an MQCONNx?
   The ConnectOpts parameter is inserted between the Name and Hconn parameters.

Unit 3: Major Call - MQPUT

1. Which PUT option forces the call to return if the queue manager is being stopped?
   MQPMO_FAIL_IFQUIESCING

2. The MQPUT parameter that references the Message Descriptor - is it a field or a structure?
   A structure.

3. The MQPUT parameter that references the Put message options - is it a field or a structure?
   A structure.

Unit 4 - Opening Queues and Message Descriptor

1. What are the four predefined Message Types?
   Request, report, reply, datagram.
2. If a message sent to a remote queue can't be delivered to the remote queue, what happens to it by default?

   It will be placed on the DLQ.

3. Name four report options.

   Among them are: COA, COD, expiration, exception, PAN, NAN.

4. Which three fields in the MQMD relate to data conversion?

   Encoding, CCSID, format.

5. What is the difference in syntax between an MQOPEN for a local queue and an MQOPEN for a remote queue?

   There is none.

Unit 5: Major Calls - MQGET and MQPUT1

1. Which option forces a call to return if the queue manager is being stopped?

   MQGMO-FAIL_IF_QUIESCING

2. What is the default action of the queue manager if you try to get a message that is too large for the application buffer?

   The message remains on the queue, as much as will fit is copied into the application buffer, the full application data length is returned in the data length parameter and the reason code MQRC_TRUNCATED_MSG_FAILED is returned to the application.

3. Which MQPUT parameter is different in an MQPUT1 call?

   The object handle is replaced by the object descriptor.

Unit 6: Controlling Message Retrieval

1. Which two fields in the MQMD allow responses to be related to requests?

   MsgId and CorrelId

2. How does the application ask the queue manager to generate a new unique MsgId when creating a message?

   Set the msgid in the application MQMD to MQMI_NONE or set the MQPMO_NEW_MSG_ID option in MQPMO options.

3. What two things must an application set when establishing an MQGET with a wait?

   Wait option, wait interval.
Unit 7: MQINQ and MQSET

1. How do the names of character attribute constants begin?
   MQCA_

2. Name two common uses for the MQSET call.
   Manipulating triggering attributes, manipulating PUT and GET inhibited.

Unit 8: Transaction Support and Triggering

1. What happens to a message destructively gotten in syncpoint if the program issues an MQBACK call?
   It is reinstated in same position on the queue.

2. How can you tell how many times a message has already been backed out?
   Look at the value in the MQMD backout count field.

3. What are the three types of triggering?
   FIRST, DEPTH, EVERY

Unit 9: MQI Security

1. Name four of the context fields.
   User Identifier, Accounting Token, Appl Identity Data, Put Appl Type, Put Appl Name, Put Date, Put Time, Appl Origin Data.

2. Which time zone is used in context information?
   GMT

3. What field is used for the Alternate UserId open request?
   The Object Descriptor AlternateUserId.

Unit 10: Message Groups and Segmentation

1. What advantages do Message Groups provide?
   Guaranteed sequence of retrieval, associating messages in a logical set.

2. Where is the Group ID specified?
   In the message descriptor.

3. Which option would you use if you wanted to ensure that your application got messages only from complete groups?
   MQGMO_ALL_MSGS_AVAILABLE
Unit 11: Distribution Lists

1. Which field in Object Descriptor indicates whether a distribution list is being opened?
   RecsPresent

2. Which Reason Code is returned when some queues in the distribution list have opened successfully and some have not?
   MQRC_MULTIPLE_REASONS
Appendix B. Housekeeping Calls Solution

Housekeeping Calls Paper Exercises C Solutions

MQCONN (QManager, &Hcon, &CompCode, &CReason);

MQOPEN (Hcon, &odG, O_options, &Hin, &CompCode, &Reason);

MQOPEN (Hcon, &odP, O_options, &Hout, &CompCode, &Reason)
Housekeeping Calls Paper Exercises COBOL Solutions

CALL 'MQCONN' USING QM-NAME, HCONN, COMPLETION-CODE, CON-REASON.

CALL 'MQOPEN' USING HCONN, OBJECT-DESCRIPTOR, OPTIONS, GQ-HANDLE, OPEN-CODE, REASON.

CALL 'MQOPEN' USING HCONN, OBJECT-DESCRIPTOR, OPTIONS, PQ-HANDLE, OPEN-CODE, REASON.
Appendix C. Reference Messages
Reference Messages

- Allows large objects to be transferred from one node to another:
  - The object does not need to be stored on any WebSphere MQ queues
  - A message exit is needed at both ends of a channel
  - The reference message references the object through the MQRMH header
    - This is the only message which needs to be put on an WebSphere MQ queue

- MQRMH specifies information like:
  - Source object name
  - Object type, Object instance ID
  - Source environment data
  - Destination environment data
  - Destination name
  - Data length

Notes:

The idea of being able to transfer data using the benefits of WebSphere MQ ease of programming and assured delivery is appealing. However, with a very large piece of data, this could be a problem. The Reference Message allows WebSphere MQ to transfer large chunks of data without having to use queue space of an equally large size. This is particularly appealing when the data already exists in some other form, for example, mail applications.

By using the Reference Message Header structure (MQRMH), the message that does go on WebSphere MQ queues is only as large as the header.

The effort of writing the message exits on either side is required and the receiving application will have to allow for the handling of the actual data but system overhead can be reduced by not having the actual data stored on queues on either side.

The MQRMH contains fields that relate to the bulk data, somewhat as the message descriptor (MQMD) contains information about the message. The actual message content is the MQRMH so fields in the MQMD are describing the header.
Notes:
Assume an application is transferring large amounts of bulk data using WebSphere MQ and reference messages.

1. A program builds a message that contains an MQRMH and issues the MQPUT, delivering the message to the transmission queue (with the normal transmission header attached).

2. The message channel agent on the sending side retrieves the message.

3. Using an attribute of the channel definition that names its message exit, the MCA will call that exit.

4. The exit finds the data being pointed to by the MQRMH and attaches it to the message.

5. The message containing the MQRMH and the data is returned to the MCA to be put on the channel.

6. These large messages travel across the channel to the receiving MCA.

7. The receiving MCA invokes the message exit identified as one of its attributes.
8. This exit essentially strips off the data and stores it according to the way the exit is written, leaving the MQRMH.

9. The exit returns this MQRMH message to the MCA.

10. The MCA delivers the MQRMH message to the destination queue.

11. The application that retrieves the MQRMH message will be designed to use information from the MQRMH to know how to go about retrieving the actual data.
Notes:

First, let's consider some of the fields in a bit of detail from the MQRMH structure. The order listed above is not that of their occurrence in the structure but is more listed regarding their relationships. So, we will follow the list above. For the exact structure layout, refer to the Application Programming Reference.

- Starting with SrcNameLength and SrcNameOffset:
  - **SrcNameLength** is the length of the source object name. If the field is zero, there is no SrcObjectName and the SrcNameOffset is ignored.
  - **SrcNameOffset** specifies the offset of the source object name from the MQRMH. The name can be specified by the creator of the reference message, if known. If not known, it is one of the tasks left to the user-written message exit.

- **SrcEnvLength** and SrcEnvOffset are next:
  - **SrcEnvLength** specifies the length of source environment data; if zero, there is no source environment data and SrcEnvOffset is ignored.
- **SrcEnvOffset** specifies the start of the source environment data from the start of the MQRMH structure. It can be specified if known by the creator of the reference message. An example of source environment data might be the directory path of the object containing the bulk data in OS/2. If not known by the creator, the user-written exit is responsible for any needed environment information.

- **ObjectTyype** and **ObjectInstanceId** follow:
  - **ObjectType** is a name that can be used by the message exit to recognize types of reference messages it supports.
  - **ObjectInstanceId** can be specified to identify a specific instance of an object. If not needed, it should be set to MQOII_NONE.

- **StrucLength** defines the total length of the MQRMH structure, including strings at the end of fixed fields, but not bulk data.

- **DestNameLength** and **DestNameOffset** next:
  - **DestNameLength** is the length of the destination object name; if set to zero, there is no destination object name and DestNameOffset is ignored.
  - **DestNameOffset** specifies the offset of the destination object name from the start of the MQRMH. If known, it can be supplied by the reference message creator; otherwise, the user-written exit must identify the object to be created or modified.

- **Finally, DestEnvLength** and **DestEnvOffset**:
  - **DestEnvLength** is the length of the destination environment data. If zero, there is no destination environment data and DestEnvOffset is ignored.
  - **DestEnvOffset** specifies the offset of the destination environment data from the start of the MQRMH. Similar to the SrcEnvOffset, this serves the same purpose for where the bulk data is to be stored.
**Transmission to the Remote Node**

Figure C-4. Transmission to the Remote Node

**Notes:**

Some additional fields in the MQRMH are used during this stage:

- **DataLogicalLength** specifies the length of the bulk data referenced by the MQRMH structure. If the bulk data is present in the message, it starts immediately after the MQRMH structure. The length of the entire message, minus the offset within the message of the end of the MQRMH gives the length of the bulk data.

- **DataLogicalOffset** specifies the low offset of the bulk data from the start of the object of which the bulk data is part. This is also known as the logical offset. The physical offset is the value in StrucLength. To allow large objects to flow, the logical offset is divided into two fields and the actual logical offset is the sum of these two fields. DataLogicalOffset represents the first field and is the remainder obtained when the logical offset is divided by 1,000,000,000.

- **DataLogicalOffset2** is the second of the two fields used to determine the actual logical offset. It specifies the high offset of the bulk data from the start of the object of which the bulk data is a part. The value represents the result obtained when the logical offset is
divided by $1,000,000,000$. It is the number of complete multiplies of $1,000,000,000$ that exist in the logical offset.

- **MQRMH_LAST** is actually a value that can be used in the MQRMH Flags field. If the Flags field is set to MQRMH_LAST, it indicates that the reference message contains or represents the last part of the object. If MQRMH_NOT_LAST, then the reverse is true. Note the MaxMsgLength at number 6. There is a MaxMsgLength associated with each side of the channel. The exit can only return a single message to the MCA for each message it has passed. This means the putting application may need to issue multiple MQPUTs to cause one entire object to be passed. Each message must identify the logical length and offset of the object that is to be appended to it. However, it may not be possible to know the total size of the object or the maximum size of the channel. In this case, the sending MCA can be designed so that the putting application puts a single message and the exit itself puts the next message on the transmission queue when it has passed as much data as possible.
Receiving Side

7. The receiving MCA invokes the message exit

8. The message exit creates the object in the appropriate way

9. The message exit returns the reference message WITHOUT the object data

10. The MCA puts the reference message to the destination queue

11. The reference message is retrieved by an application which knows that the object has been created at this node

**Notes:**

As depicted, there is little done on the receiving side that is not dependent on user-written code. The message exit will create or update the object as appropriate and the user application will use the reference message to work with the object.
Several Considerations

- The object creation is not synchronized with the channel UOW
  - If a batch is backed out, the same portion of the object will arrive later
  - Receiving end may need a log, for example, an WebSphere MQ queue
- Reference messages cannot be segmented
- Reference message can be sent to distribution lists
  - MQDH and so forth, structures precede MQRMH in XmitQ
- Object data is not normally converted
  - Not normally present in the reference message
  - Converted if present and conversion requested
    - Format, Encoding and CodedCharSetId MQMD fields describe MQRMH
    - Equivalent fields in MQRMH describe the object data

Notes:

Channels transfer messages by issuing MQGETs in syncpoint from the transmission queue and MQPUTs in syncpoint on the destination queue. There may be one or more messages transferred before a batch is committed. If some parts of a batch are delivered as reference messages (containing portions of an object), it is possible that the bulk data portion can be delivered again in a new batch.

We will discuss distribution lists in a moment; for now, just know that reference messages can use them.

Data conversion depends on whether the actual data is in the reference message or not. If so, conversion can be achieved using the fields contained in the MQRMH.
Bibliography

Manuals:
SC34-6064 WebSphere MQ Application Programming Guide
SC34-6062 WebSphere MQ Application Programming Reference
GC34-6058 WebSphere MQ Clients
SC34-6055 WebSphere MQ Script (MQSC) Command Reference
SC34-6067 WebSphere MQ Using C++
SC34-6066 WebSphere MQ Using Java
GC34-6073 WebSphere MQ for Windows Quick Beginnings
SC34-6079 WebSphere MQ Security
SC34-5059-03 WebSphere MQ Intercommunication

Technical Bulletins:
SG24-6508 MQSeries Programming Patterns
GC34-5269 MQSeries Publish/Subscribe User’s Guide
SC34-6275 WebSphere MQ Extended Transactional Clients

Web URLs:
  IBM WebSphere MQ

supported.html
  Supported software for IBM WebSphere MQ