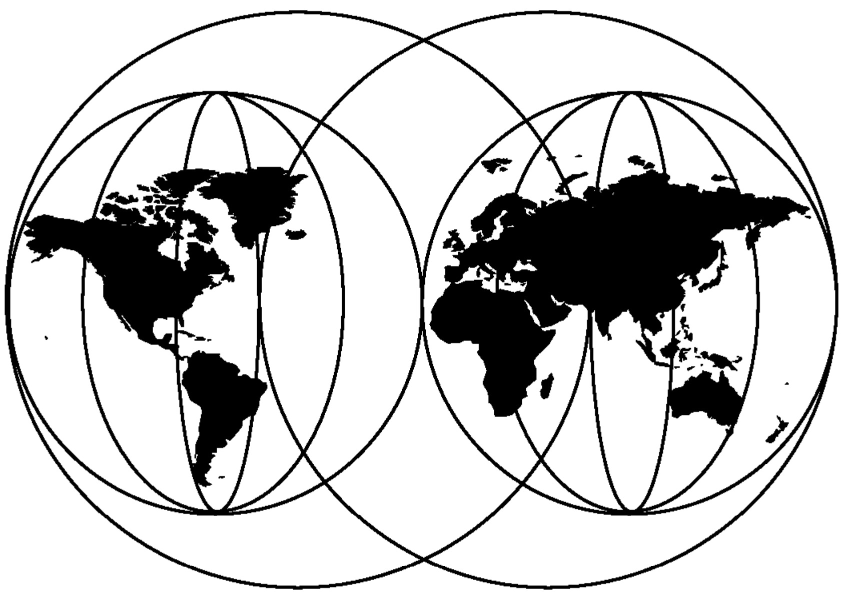
MQSeries Version 5 Programming Examples

Dieter Wackerow, Robert Gage



International Technical Support Organization

http://www.redbooks.ibm.com

This book was printed at 240 dpi (dots per inch). The final production redbook with the RED cover will be printed at 1200 dpi and will provide superior graphics resolution. Please see "How to Get ITSO Redbooks" at the back of this book for ordering instructions.

SG24-5214-00

SG24-5214-00

International Technical Support Organization

MQSeries Version 5 Programming Examples

October 1998

Take Note!

Before using this information and the product it supports, be sure to read the general information in Appendix I, "Special Notices" on page 265.

First Edition (October 1998)

This edition applies to the following products:

- MQSeries for AIX Version 5
- MQSeries for OS/2 Warp Version 5
- MQSeries for Windows NT Version 5

Comments may be addressed to:
IBM Corporation, International Technical Support Organization
Dept. HZ8 Building 678
P.O. Box 12195
Research Triangle Park, NC 27709-2195

Contents

Figures

Tables

Preface

This redbook helps you to design and develop application programs that use the features of MQSeries Version 5. MQSeries Version 5 is available for five platforms, OS/2, AIX, Windows NT, HP-UX and Sun Solaris. Some of the functions are also available on the AS/400.

This redbook outlines the new features of MQSeries Version 5. It is based on class exercises for an ITSO workshop. Several practical examples are presented to demonstrate how to:

- Segment large messages of up to 100MB.
- Group messages for better performance.
- Send messages to multiple destinations using a distribution list.
- Coordinate queueing functions and database updates using a two-face commit.
- Perform remote administration in a Windows NT workgroup.
- Transfer files to other systems using reference messages.
- Improve performance using fastpath bindings.
- Write multi-threaded programs.

The first chapter contains an overview of the functions released with MQSeries Version 5. The other chapters are dedicated to specific functions. They include programming hints and examples. This redbook comes with a diskette that contains the source code of all examples.

The Team That Wrote This Redbook

This redbook was produced by a team of specialists from around the world working at the System Management and Networking ITSO Center, Raleigh.

Dieter Wackerow is the MQSeries expert at the Systems Management and Networking ITSO Center, Raleigh. His areas of expertise include application design and development, performance evaluations, capacity planning, and modelling of computer systems and networks. He also wrote a simulator for banking hardware and software. He teaches classes and writes on performance issues, application development and about MQSeries.

Robert Gage is a Sr. Marketing Support Representative with IBM Software Group, Worldwide Sales and Technical Support/MQSeries.

Thanks to the following people for their invaluable contributions to this project:

David Armitage, IBM Australia

Satish Babu, IBM India

Hany Adel Kaiser, IBM Egypt

Jill Lennon, IBM USA

Marc Luong, IBM France

Geert Van de Putte, IBM Belgium

Matthias Schuette, IBM Germany

Jimmy Tsai, IBM Taiwan

Steve Bolam
Andy Hickson
Brian Homewood
James Wilkinson, IBM Hursley, England

Comments Welcome

Your comments are important to us!

We want our redbooks to be as helpful as possible. Please send us your comments about this or other redbooks in one of the following ways:

- Fax the evaluation form found in "ITSO Redbook Evaluation" on page 277 to the fax number shown on the form.
- Use the electronic evaluation form found on the Redbooks Web sites:

For Internet users	`http://www.redbooks.ibm.com/`
For IBM Intranet users	`http://w3.itso.ibm.com/`

- Send us a note at the following address:

  `redbook@us.ibm.com`

Chapter 1. About MQSeries Version 5

MQSeries Version 5 is interoperable with current and previous releases of the MQSeries product on all platforms. The Version 5 queue managers will operate on the Version 2 level for connections to both Version 1 and Version 2 queue managers.

The following MQSeries Version 5 products have been available since September 1997:

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

1.1 Important New Functions of MQSeries Version 5

MQSeries Version 5 adds a number of customer requested functions which will simplify MQSeries installation, application design, system administration and problem determination.

1.1.1 Database Resource Manager

This is one of most important new features of MQSeries Version 5. The queue manager is able to coordinate database updates and messaging activity. With Version 5, a customer application which includes both MQSeries and SQL activity uses a new API of MQSeries, the MQBEGIN verb, to register a logical unit of work (LUW). MQSeries and SQL activity can be committed or backed out atomically, using the MQCMIT or MQBACK verbs.

Restart resynchronization between MQSeries and coordinated Relational Data Base Management (RDBM) is provided by MQSeries Database Message Resource Coordination (DMRC).

Examples are described, in detail, in Chapter 2, “Transaction Coordination” on page 7.

1.1.2 Transaction Coordinator

MQSeries is designed to work well in conjunction with transaction monitors, such as CICCS, Encina, etc. Its advantages include scalability, performance, resource sharing, system administration and system management.

1.1.3 Distribution Lists

The enhanced message distribution carries more business information, while minimizing use of the network. Multicast customer applications require the ability to send information to multiple destinations. MQSeries also provides smart message distribution. This minimizes the amount of network trafic required to distribute a copy of a single message to multiple users whose queues reside on a single node.

One MQSeries MQPUT can now be used to send copies of a single message to multiple destinations with assured delivery to each destination.

Distribution lists allow you to put a message to multiple destinations in a single MQPUT or MQPPUT1. 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 destination list. When an MQOPEN call is issued, generic information is taken from the *Object Descriptor* (MQOD). If you specify MQOD_VERSION_2 in the *Version field* and when a message is put on the queues (MQPUT), generic information is taken from the *Put Message Options* structure (MQPMO) and the *Message Descriptor* (MQMD). The specific information is given in the form of *Put Message Records* (MQPMR). The *Response Records* (MQRR) can receive a completion code and reason code specific to each destination queue.

Examples are described, in detail, in Chapter 7, “Distribution Lists” on page 141.

1.1.4 Handling Large Messages

With MQSeries Version 5, the maximum message length is 100 MB, up from 4 MB. Non-Version 5 queue managers, such as queue managers running in MVS/ESA and AS/400 systems are still limited to 4 MB.

1.1.4.1 Message Segmentation and Message Groups

Messages may now be built or retrieved in segments. This is known as ″partial put/get″. This allows application programs to deal with messages larger than could be stored in a single buffer. It can also be used to group multiple records into a single MQSeries message.

Examples are described, in detail, in Chapter 3, “Message Segmentation” on page 65 and Chapter 4, “Message Groups” on page 89.

1.1.4.2 Reference Message

This is a new feature of the API MQPUT. The message is actually a logical pointer to external data such as a file, graphical images or other stored data. MQSeries will move the referenced data in its assured manner, store it at the receiving site and make the reference available to the target process in the corresponding new form of the API MQGET.

This work is done by a new IBM-provided MQSeries messages exit. The program fetches the messages data indicated in the reference message header and sends it to the remote queue manager. Usually, the remote queue manager will have the corresponding message exit installed which writes the incoming message to a file or optionally to a queue.

An example is described in Chapter 6, “Reference Message” on page 119.

1.1.5 SPX Support

IPX and SPX are the proprietary native protocols used on Novell LANs. In MQSeries Version 5 SPX is a supported transport protocol for the following platforms and clients: OS/2, Windows NT, DOS, Windows 3.1 and Windows 95.

1.2 Enhancement of Existing Functions

Here are some important enhancements to existing functions:

1.2.1 Performance

MQSeries Version 5 can transmit messages eight times faster than the previous version.

Applications using fastpath bindings (MQCONNX) run faster than standard bindings. Chapter 8, “FastPath Bindings” on page 159 provides an example that lets you measure the performance inprovement.

1.2.2 Change in Triggering Rules

With MQSeries Version 5 the process definition object for channels has been eliminated. You do NOT need to create a process definition object; the tranmission queue definition is used instead. When a trigger event occurs, the transmission queue definition contains information about the application that processes the message which caused the event. Again, when the queue manager generates the trigger message, it extracts this information and places it in the trigger message.

1.2.3 Data Conversion and Exits

Several new code pages and languages are now supported. Channel exits can now also be chained. An example is shown in Chapter 6, "Reference Message" on page 119.

1.2.4 Channels

New channel attributes have been added. You can define "fast" channels, a heartbeat interval, a batch interval and you can chain channel exits.

1.3 Enhancement of Product Installation and Administration

Installation of the product and its administration is now easier.

1.3.1 Product Installation

During MQSeries installation, default MQSeries objects are automatically created. The separate execution of MQSeries commands is no longer required. The MQSC syntax and user interaction are improved for definition of MQSeries resources and issuing commands.

1.3.2 Product Administration

Administration of MQSeries channels and auto-definition of server-connection and receiver channels is now supported.

Dynamic definition is provided for receiver and MQI server channels. Definition is optionally eliminated for processes associated with triggered channels. Process definitions are still allowed, if desired.

MQI Server channel status is now available. This applies to MQSeries clients. Channel status is preserved over a restart. Channels which have failed or were in retry prior to a shutdown appear in that mode following restart.

One of the new features of MQSeries Version 5 is channel auto-definition. This means that you don't have to define a receiver channel. Try this out by deleting your receiver channels. Change the queue manager attribute CHAD to enable this feature. Do not forget that your existing sender channel has a status record with a message sequence number. Before you test this out, you should reset the sender channels. In a runmqsc session, type: `alter qmgr chad(enable)`. This enables the channel auto-definition feature.

1.4 Enhancement of Application Interface Development

The additional developer feature includes further language support for C++, Java and PL/1 and interoperability with current and previous MQSeries versions:

- Lotus Notes link
- SAP R3
- Web Internet Gateway
- Java application
- Support for Encina (by MQSeries on Windows NT)

1.5 Internet Support

MQSeries Version 5 support for the Internet includes:

- Internet Gateway

  This provides a bridge between the synchronous world of the World Wide Web and asynchronous MQSeries applications.

- MQSeries Client for Java

  This functions lets you write Java applets that can connect to a queue manager that runs in the same machine as the Web server.

- MQSeries Bindings for Java

  This set of Java classes lets you write server applications using Java and MQSeries.

- HTML publications

  MQSeries manuals are available in HTML format an the product CD.

1.6 Enhancement of DCE Security

Authentication and encryption are supported for both MQSeries server-to-server and MQSeries client-to-server links. For MQ/Client links, the message exits are not present, so send/receive exits must be used.

1.6.1 Message Authentication

MQseries security exits, message and send/receive exits are provided for optional use. The security exits require and use DCE security to provide authentication of MQSeries partners before any messages are sent or received.

1.6.2 Message Encryption

A choice of either message exits or send/receive exits is provided for encryption of messages.

1.6.3 Channel Exits and Data Conversion Exits

This prevents system users from running their own set of exits, contrary to the system administrator's policy.

1.7 Enhancement of Problem Determination

New means of problem resolution will assist in collecting diagnostic information to speed problem determination:

1. New trace functions
2. Log dump utility
3. FFST improvements for diagnostic of internal MQSeries problems
4. Addition of RAS folder to hold diagnostic information

1.8 Integration in IBM Software Server Product Package

IBM Software Server Product package provides a set of common IBM products for application servers which include DB2, Transaction Server (CICS, Encina) and MQSeries. By integrating IBM Software Server standards, MQSeries fits more comfortably into customer application server solutions that include multiple components.

1.9 Integration in IBM Suite NT Product Package

MQSeries is included in the IBM Suite NT Product Package which contains a set of IBM software products: DB2, CICS, Encina, Lotus Notes, ADSTAR Distributed Storage Manager, Netscapes, DCE, LSX Support, IBM Communications Server.

MQSeries Version 5 Web Page

http://www/software.ibm.com/ts/mqseries/v5

Chapter 2. Transaction Coordination

MQSeries is already an XA-compliant resource manager. This allows it to participate in a two-phase commit coordinated by an XA transaction manager such as CICS, Encina or Tuxedo.

A two-phase commit ensures that updates to resources that belong to different resource managers can be made with integrity. For example, consider a CICS transaction that updates both an MQSeries queue and a table in a DB2 database. For integrity to be maintained, the updates to the MQSeries queue and the DB2 table must both succeed or both fail.

In MQSeries Version 5, the queue manager is both an XA resource manager and an XA resource coordinator. When the queue manager is acting as the XA coordinator it becomes possible to write a mixed MQI and SQL application and use the MQCMIT verb to commit or the MQBACK verb to roll back the changes to the queues and databases together.

In this chapter, we provide examples on how to use the transaction monitor supplied with MQSeries Version 5 for external syncpoint coordination.

Note: You can use the transaction monitor only with a server application. This support is not available to client applications.

In the following sections we explain:

- What a local and global unit of work is
- What the new verb MQBEGIN does
- What an XA switch file is for and how to create it
- How to tell the queue manager what databases to coordinate
- How to create, populate and work with a database
- How to write and compile programs that update queues and databases
- How to commit and back out transactions
- How to monitor database updates

The examples in this chapter not only describe how to write the code for transaction coordination but also what to do when something goes wrong.

Diskette

The source code and the make files for all examples are on the diskette. Refer to Appendix H, “Diskette Contents” on page 261.

2.1 Units of Work

With MQSeries Version 5, the new verb MQBEGIN is introduced to start a *global* unit of work. A global unit of work includes both database and queue updates while a *local* unit of work consists of queue updates only. The latter is the pre-Version 5 unit of work.

2.1.1 Local Unit of Work

Local units of work only update queues that belong to the queue manager itself. A local unit of work is started by an MQGET, MQPUT or MQPUT1 call which specifies the corresponding syncpoint option. Subsequent calls which also specify the syncpoint option are considered to be part of the same unit of work, until the unit of work is committed or rolled back. Table 1 shows an example.

Units of work can be committed explicitly by an MQCMIT call, or implicitly by an MQDISC call. Units of work can be rolled back explicitly by an MQBACK call, or implicitly if the application terminates without first disconnecting.

Table 1. Local Unit of Work
MQCONN() ⋮ MQGET(MQGMO_SYNCPOINT) ⋮ MQCMIT() ⋮ MQPUT(MQPMO_SYNCPOINT) ⋮ MQDISC

2.1.2 Global Unit of Work

Global units of work can update both queues and databases and need to be started with MQBEGIN. This new verb has been added to the MQI to start a unit of work that involves other resource managers. These units of work are *global* since they update more than just local resources.

After the MQBEGIN call has been issued, local resources can be updated using MQGET, MQPUT or MQPUT1 calls made under syncpoint. Updates to databases need to be made using the SQL API provided by the appropriate database manager. Table 2 on page 9 shows an example.

The method for committing global units of work is the same as for local units of work. The unit of work can be committed by an MQCMIT or an

MQDISC call. Alternatively, the unit of work will be rolled back by an MQBACK call, or if the application terminates without first disconnecting.

Table 2. Global Unit of Work
MQCONN() ⋮ MQBEGIN() ⋮ MQGET(MQGMO_SYNCPOINT) ⋮ EXEC SQL UPDATE ⋮ MQCMIT() MQDISC()

2.1.3 Mixing Units of Work

It is possible to write an application that consists of both local and global units of work.

Units of work that make only queue manager updates can be started using an MQGET, MQPUT or MQPUT1 call specifying the appropriate syncpoint option. Units of work that also need to update global resources owned by a database manager need to be started using an MQBEGIN call. It is also possible to start a global unit of work that:

- Only updates local queues
- Only updates databases (doesn′t make any queue updates)
- Doesn′t make any updates (effectively a no-op)

Table 3 shows one valid and two invalid units of work. There can only be a single unit of work in existence at one time. It is an error to try to start another unit of work while there is another already in progress.

Table 3. Mixing Units of Work

Valid	**Invalid**	**Invalid**
MQCONN () MQPUT (MQPMO_SYNCPOINT) MQCMIT () ⋮ ⋮ MQBEGIN () MQGET (MQGMO_SYNCPOINT) EXEC SQL UPDATE MQCMIT ()	MQCONN () MQPUT (MQPMO_SYNCPOINT) ⋮ MQBEGIN ()	MQCONN () ⋮ MQBEGIN () ⋮ MQBEGIN ()

2.1.4 The MQBEGIN Verb

The MQBEGIN verb has the following syntax:

```
MQBEGIN  (HConn, BeginOptions, CompCode, Reason)
```

The options structure is provided only for future extensibility. You are expected to pass the default structure, MQBO_DEFAULT. In a C program, you may specify a NULL pointer. The call can return the following errors:

- 2121 - MQRC_NO_EXTERNAL_PARTICIPANTS

  The queue manager has not been configured with any external resource managers. A unit of work is still started but it may only involve queue updates.

- 2122 - MQRC_PARTICIPANT_NOT_AVAILABLE

  One of the databases which the queue manager has been configured with is not available at the moment. A unit of work is still started but it won′t be able to update the unavailable databases.

- 2134 - MQRC_BO_ERROR

  The BeginOptions structure is not valid. No unit of work is started.

- 2128 - MQRC_UOW_IN_PROGRESS

  A local or global unit of work is already in progress. No new unit of work is started.

2.1.5 Outcome of a Unit of Work

Two new error responses have been introduced to cater for failures that can occur during the syncpoint of a unit of work that involves other resource managers. They can be returned from an MQCMIT, MQDISC or MQBACK.

- 2123 - MQRC_OUTCOME_MIXED

  This denotes a failure to commit a unit of work. It indicates that shared resources are in a potentially inconsistent state. Some of the updates made within the unit of work were committed whereas others were rolled back.

  You have to look at the queue manager′s error logs for messages relating to the mixed outcome. The messages identify the resource managers that are affected. Use procedures local to the affected resource managers to re-synchronize the resources.

- 2124 - MQRC_OUTCOME_PENDING

  This warning can be returned when a database manager becomes unavailable during the second phase of commit. The database will remain in doubt until the queue manager re-synchronizes with it when it becomes available again. While the database updates remain in doubt, the possibility of a mixed outcome of a unit of work remains.

2.2 Database Configuration

In general, the queue manager should be able to support any XA-compliant database manager. Initially, only DB2 (Version 2.1.1) and Oracle (Version 7.3.2) are supported. Refer to the announcement material for more details.

The ability to coordinate global transactions involving updates to databases is only supported on the queue manager server. Client applications will receive a runtime error if they issue an MQBEGIN to start a global unit of work.

Applications must run locally on the same machine as the queue manager. Updates to databases must also be made on this same machine. These can be local if the database server is on the same machine as the queue manager. If the database server resides on a different machine then the database needs to be accessed through an XA-compliant client feature provided by the database manager, *not* the queue manager. Figure 1 shows the two configurations.

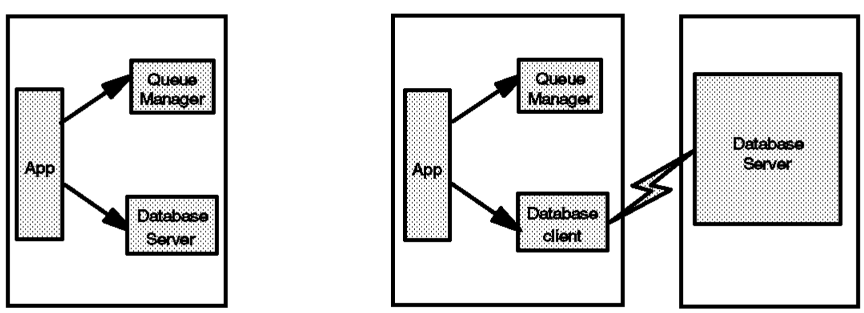


Figure 1. Database Client/Server Configurations

2.2.1 Multiple Databases

Coordination of multiple databases is supported. You can include updates to more than one database within one global unit of work. The databases may be of the same or different kinds and the database server does not need to reside in the same machine as the queue manager.

The queue manager places no restrictions upon the number of databases that can be updated in a unit of work, though it is hard to imagine an application that will need to update more than two or three databases.

MQSeries is an XA-compliant resource manager and it would be a reasonable question to ask whether one queue manager can coordinate updates made to queues owned by another queue manager. This is *not* allowed because an application can only connect to a single queue manager at one time.

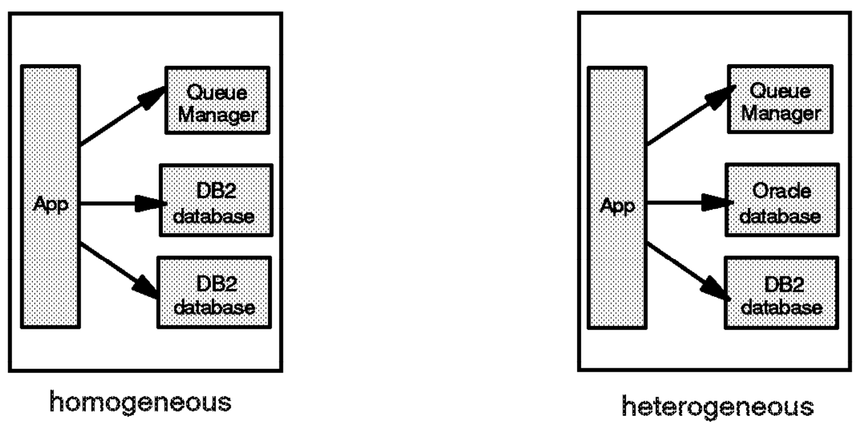


Figure 2. Coordination of Multiple Databases

2.2.2 Configuring Database Managers

There are four steps to perform before a database manager can participate in global units of work:

1. Create an XA-switch for the database manager.
2. Define the database managers in the qm.ini file.
3. Set the tp_mon_name parameter.
4. Set database security.

A brief description and an example follows.

Note: For a detailed description refer to *MQSeries System Administration*, SC33-1873 and the *MQSeries Application Programming Guide*, SC33-0807.

2.2.2.1 The XA-Switch

A resource manager′s XA-switch is a DLL or shared library with a single entry point. When it is called it returns the address of the xa_switch_t structure for the resource manager. This structure contains the name of the resource manager, option flags and all the XA function pointers. Since the programmer does not use this structure we don′t describe it here; however, we have to create the *XA-switch load file*.

The files to create the XA-switch are in the following directories:

NT mqm\tools\c\samples\xatm

OS/2 mqm\tools\c\samples\xatm

AIX /usr/lpp/mqm/samp/xatm

The names of the source files for the switch file are:

- For DB2, db2swit.c
- For Oracle, oraswit.c

How to make the switch file is described in 2.5, “Exercise 1: Setup for XA Coordination” on page 27.

2.2.2.2 The XA Resource Manager Stanza

An XAResourceManager stanza must be added to the QM.INI file. You find this file in the directory directory \mqm\qmgrs\queue_manager_name. Figure 3 on page 14 shows a qm.ini file for Windows NT with one resource manager stanza. The parameters are:

```
XAResourceManager:
   Name=<database_manager_name> <database_name>
   SwitchFile=<switch_file_name>
   XAOpenString=<database_name>
   ThreadOfControl=THREAD
```

Figure 3 on page 14 shows an example of a qm.ini file that contains one resource manager stanza.

```
ExitPath:
   ExitsDefaultPath=C:\MQM\exits
Service:
   Name=AuthorizationService
   EntryPoints=9
ServiceComponent:
   Service=AuthorizationService
   Name=MQSeries.WindowsNT.auth.service
   Module=C:\MQM\bin\amqzfu.dll
   ComponentDataSize=0

XAResourceManager:
   Name=DB2 MQBANKDB
   SwitchFile=c:\MQM\BIN\DB2SWIT.DLL
   XAOpenString=MQBANKDB
   ThreadOfControl=THREAD

Log:
   LogPrimaryFiles=3
   LogSecondaryFiles=2
   LogFilePages=256
   LogType=CIRCULAR
   LogBufferPages=17
   LogPath=C:\MQM\LOG\DIETER\
```

Figure 3. Queue Manager Configuration File QM.INI

Notes:

1. The database manager name for DB2 is DB2.
2. The database name and the XAOpenString is either MQBANKDB or MQFEEDB.
3. The name of the switch file is db2swit.dll. Specify its path.
4. For DB2 on OS/2 and Windows, the ThreadOfControl parameter is always THREAD. Omit this line if you configure MQSeries and DB2 on a UNIX system.

Important

You need a separate stanza for each database *even* if they use the same resource manager.

2.2.2.3 The TP_MON_NAME Parameter

This is only required for DB2 on OS/2 and Windows NT. It names MQMAX.DLL as the library that DB2 uses to call the queue manager. The DLL resides in the directory \mqm\bin. Make sure you have a LIBPATH for it. In a DB2 command window, enter this command:

```
db2=> update dbm cfg using TP_MON_NAME mqmax
```

2.2.2.4 Database Security

Refer to the documentation provided with the database manager to determine the security implications of running your database under the XA model.

2.3 Software

To develop the examples for this book we used the following operating systems:

- AIX Version 4.1.4.0
- OS/2 Warp Version 4.0
- Windows NT Version 4.0

On all systems we installed the Universal Database Version 5 (DB2) and the DB2 Software Developer′s Kit from the DB2 Application Developer′s Kit which is a separate product.

The C samples are written in ANSI C and can be compiled with the following compilers:

AIX C for AIX Compiler Version 3.1.4 (xlc)

OS/2 IBM VisualAge C++ Version 3.0 for OS/2

NT Microsoft Visual C++ Version 2.0 and Version 3.5.3

IBM VisualAge C++ Version 3.5 for Windows

The COBOL examples have been developed under Windows NT using VisualAge for COBOL Version 2.1.

2.3.1 Installation Hints for Windows NT

The following tips help you to test the examples provided with this book:

1. Set up a user ID with the Name *Admin* and include it in the group *Administrators*. The user ID Administrator is one character too long for MQSeries.
2. Log on as Admin and use this ID from now on.

3. Install MQSeries Version 5 and re-boot.

   Note: The user ID Admin becomes a member of the group *mqm*.

4. Create a default queue manager.

   Note: Since Version 5 this includes creating all default objects for that queue manager.

5. Install the Universal Database Version 5 and the SDK, and re-boot.

   Notes:

   a. DB2 Version 2.1.2 will work, too. We used this version from the Demo Package June 97.

   b. With the SDK you install the C samples.

6. Install one of the C compilers.

If you are using the MicroSoft C compiler, the following environment values must be added. If you already have the variable, (for example, PATH) add the value to the end of the existing chain. If it does not exist (for example, CPU) add the variable and value.

Path	`C:\MSVC20\bin`
Libpath:	`C:\MSVC20\LIB`
Include:	`C:\MSVC20\INCLUDE`
CPU:	`i386`
	The ′i′ should not be capitalized!

Note: The above example is for Microsoft Visual C++ Version 2.0.

The settings for the environment are in the System Properties:

- Select **Start**, then **Settings**, then **Control Panel**.
- Click on **System**, then on the **Environment** tab.
- Select **System Variable**, edit the value and click on **Set**.
- For a new variable, enter its name, type the value and click on **Set**.
- When finished, click on **Apply** and then on **OK**.

2.3.2 Installation Hints for AIX

The following tips help you to set up your AIX system to test the database coordination examples:

1. Log in as root.

2. To install the Universal Database and the SDK on your AIX system we need two files. They are located in the following directories:

   - /pub/db2install/db2_v500/aix/gold970815/image_aix.tar
   - /pub.db2install/db2_v500/sdk/gold970815/sdk2-unix/image_aix.tar

3. You can order the products or get them from the internal FTP site ftp3.torolab.ibm.com as shown Figure 4 on page 18.

   Note: This site is for IBM internal use only.

4. Uncompress the image file with the following command:

   `rs60001:/home/db2_image >` **`tar -xvf image_aix.tar`**

5. Next start the installation process with the following command:

   `rs60001:/home/db2_image/db2/aix >` **`./db2setup`**

6. In the DB2 Installer window shown in Figure 5 on page 19, select **Install**.

7. In the Install DB2 V5 window shown in Figure 6 on page 19, select the three products marked with an asterisk and then click on **OK**.

8. In the second Install DB2 V5 window, shown in Figure 7 on page 20, select the **DB2 Sample Database Source**. These examples help you write DB2 programs.

9. The next Install DB2 V5 window in Figure 8 on page 20 lets you choose products from the DB2 Software Developer's Kit. Select the two products marked with an asterisk.

10. In the Create DB2 Services window shown in Figure 9 on page 21, type a user name and a group name. We used db2inst1 and db2iadm.

11. Log off as root and log in as db2inst1.

```
(1) DB2 DATABASE system file
rs60001:/home/db2_image > ftp ftp3.torolab.ibm.com
Connected to enterprise.torolab.ibm.com.
220 enterprise FTP server (Version wu-2.4(2) Fri Apr 21 16:06:09 CUT 1995) ready.
Name (ftp3.torolab.ibm.com:db2inst1): anonymous
331 Guest login ok, send your complete e-mail address as password.
Password:
230-
230-Welcome to the IBM Toronto Lab FTP server.
230-
230-This server contains various files for Anonymous FTP as well
230-as all the Web documents available on w3.torolab.ibm.com.
230-You are user 9 of a maximum of 200.
230-
230 Guest login ok, access restrictions apply.
ftp> cd /pub/db2install/db2_v500/aix/gold970815
250 CWD command successful.
ftp> dir
200 PORT command successful.
150 Opening ASCII mode data connection for /bin/ls.
total 269217
-rw-r--r--   1 3576     200          1052 Nov 10 14:45 README.FTP
-rw-r--r--   1 3576     200           190 Sep 11  1997 WARNING.FTP
-rw-r--r--   1 3576     200      49960960 Sep 11  1997 books_en.tar
-rw-r--r--   1 3576     200     223641600 Oct 20 20:24 image_aix.tar
-rw-r--r--   1 3576     200         13881 Sep 11  1997 readme.txt
-rw-r--r--   1 3576     200       2058240 Sep 11  1997 repl_en.tar
226 Transfer complete.
ftp> bin
200 Type set to I.
ftp> get image_aix.tar
200 PORT command successful.
150 Opening BINARY mode data connection for image_aix.tar (223641600 bytes).

(2) SDK image file
ftp> cd /pub/db2install/db2_v500/sdk/gold970815/sdk2-unix
250 CWD command successful.
ftp> dir
200 PORT command successful.
150 Opening ASCII mode data connection for /bin/ls.
total 261275
-rw-r--r--   1 3576     200      33331200 Sep 12  1997 books_en.tar
-rw-r--r--   1 3576     200      66744320 Sep 12  1997 image_aix.tar
-rw-r--r--   1 3576     200      28293120 Sep 12  1997 image_hp.tar
-rw-r--r--   1 3576     200      27084800 Sep 12  1997 image_sco.tar
-rw-r--r--   1 3576     200      24453120 Sep 12  1997 image_sgi.tar
-rw-r--r--   1 3576     200      26931200 Sep 12  1997 image_sinix.tar
-rw-r--r--   1 3576     200      60692480 Sep 12  1997 image_solaris.tar
-rw-r--r--   1 3576     200         14518 Sep 11  1997 readme.txt
226 Transfer complete.
ftp> bin
200 Type set to I.
ftp> get image_aix.tar
200 PORT command successful.
150 Opening BINARY mode data connection for image_aix.tar (223641600 bytes).
ftp> bye
```

Figure 4. Obtain the Universal Database and the SDK

```
+------------------------------- DB2 Installer --------------------------------+
|                                                                              |
|   Select Install to select products and their components to install, or     |
|   select Create to create the DB2 services.                                 |
|                                                                              |
|                                                                              |
|   To select products and their components, select          [ Install... ]  |
|   Install.                                                                   |
|                                                                              |
|   To create a DB2 Instance, or the Administration           [ Create... ]   |
|   Server, select Create.                                                     |
|                                                                              |
|                                                                              |
|                                                                              |
|                                                                              |
|                                                                              |
|                                                                              |
|                                                                              |
|                                                                              |
|                                                                              |
|   [  Close  ]                                                   [  Help  ]  |
+------------------------------------------------------------------------------+
```

Figure 5. DB2 Installer Window on AIX

```
+------------------------------- Install DB2 V5 -------------------------------+
|                                                                              |
|   Select the products you are licensed to install. Your Proof of            |
|   Entitlement and License Information booklet identify the products for     |
|   which you are licensed.                                                    |
|                                                                              |
|   To see the preselected components or customize the selection, select     |
|   Customize for the product.                                                 |
|   :*: DB2 Client Application Enabler                      [ Customize... ]  |
|   :*: DB2 UDB Workgroup Edition                           [ Customize... ]  |
|   : : DB2 UDB Enterprise Edition                          : Customize... :  |
|   : : DB2 Connect Enterprise Edition                      : Customize... :  |
|   : : DB2 UDB Extended Enterprise Edition                 : Customize... :  |
|   :*: DB2 Software Developer's Kit                        [ Customize... ]  |
|                                                                              |
|   To choose a language for the following components, select Customize for  |
|   the product.                                                               |
|       DB2 Product Messages                                [ Customize... ]  |
|       DB2 Product Library                                 [ Customize... ]  |
|                                                                              |
|                                                                              |
|   [  OK  ]                    [ Cancel ]                        [  Help  ]  |
+------------------------------------------------------------------------------+
```

Figure 6. Install DB2 V5 Window 1 on AIX

```
+------------------------------- Install DB2 V5 -------------------------------+
+--- DB2 Universal Database Workgroup Edition ---------------------------------+
|                                                                              |
|   Required:     DB2 Client                                                   |
|                 DB2 Run-time Environment                                     |
|                 DB2 Engine                                                   |
|                 DB2 Communication Support - TCP/IP                           |
|                 DB2 Communication Support - IPX/SPX                          |
|                 DB2 Communication Support - SNA                              |
|                 DB2 Communication Support - DRDA Application Server          |
|                 Administration Server                                        |
|                 License Support                                              |
|   Optional:     [ ] Open Database Connectivity (ODBC)                        |
|                 [ ] Java Database Connectivity (JDBC)                        |
|                 : : Replication                                              |
|                 :*: DB2 Sample Database Source                               |
|                 Code Page Conversion Support:                                |
|                     : : Japanese     : : Simplified Chinese                  |
|                     : : Korean       : : Traditional Chinese                 |
|                                                                              |
|   [ Select All ]            [ Deselect All ]                [ Default ]      |
|   [  OK  ]                     [ Cancel ]                    [  Help  ]      |
+------------------------------------------------------------------------------+
```

Figure 7. Install DB2 V5 Window 2 on AIX

```
+------------------------------- Install DB2 V5 -------------------------------+
|                                                                              |
|   Select the products you are licensed to install. Your Proof of             |
|   Entitlement and License Information booklet identify the products for      |
|   which you are licensed.                                                    |
|                                                                              |
+--- DB2 Software Developer's Kit ---------------------------------------------+
|                                                                              |
|   Required:     DB2 Client                                                   |
|                 DB2 Application Development Tools (ADT)                      |
|   Optional:     [ ] Open Database Connectivity (ODBC)                        |
|                 [ ] Java Database Connectivity (JDBC)                        |
|                 :*: DB2 Sample Applications                                  |
|                 :*: Create Links for DB2 Libraries                           |
|                                                                              |
|   [ Select All ]            [ Deselect All ]                [ Default ]      |
|   [  OK  ]                     [ Cancel ]                    [  Help  ]      |
+------------------------------------------------------------------------------+
|       DB2 Product Library                              [ Customize... ]      |
|                                                                              |
|                                                                              |
|   [  OK  ]                     [ Cancel ]                    [  Help  ]      |
+------------------------------------------------------------------------------+
```

Figure 8. Install DB2 V5 Window 3 on AIX

```
+--------------------------- Create DB2 Services -----------------------------+
|+--- DB2 Instance ---------------------------------------------------------+|
||                                                                          ||
||   Authentication:                                                        ||
||       Enter User ID, Group ID and Password that will be used for         ||
||       the DB2 Instance.                                                  ||
||       User Name            [db2inst1]                                    ||
||       User ID              :        :                [*] Use default UID ||
||       Group Name           [db2iadm1]                                    ||
||       Group ID             :        :                [*] Use default GID ||
||       Password             [                ]                            ||
||       Verify Password      [                ]                [ Default ] ||
||                                                                          ||
||   Protocol:                                                              ||
||       Select Customize to change the default           [ Customize... ] ||
||       communication protocol.                                            ||
||                                                                          ||
||   [*] Auto start DB2 Instance at system boot.                            ||
||   [ ] Create a sample database for DB2 Instance.                         ||
||                                                                          ||
||   [  OK  ]                    [ Cancel ]                     [  Help  ]  ||
|+--------------------------------------------------------------------------+|
+-----------------------------------------------------------------------------+
```

Figure 9. Create DB2 Services Window on AIX

12. Add a search path to your DB2 instance user profile:

```
#
# Run $HOME/sqllib/db2profile
#
. $HOME/sqllib/db2profile
```

13. Add the following line to .profile so that you can run the MQ sample programs.

```
#
# Add MQM sample execution program
#
PATH=${PATH}:/usr/lpp/mqm/samp/bin:
```

    This completes the installation and the setup.

14. The next time you log in as db2inst1 you may want to display the profile.

```
AIX Version 4
(C) Copyrights by IBM and by others 1982, 1994.
login: db2inst1
db2inst1's Password:
[db2inst1@rs60001]/home/db2inst1 > cat .profile
PATH=/usr/bin:/etc:/usr/sbin:/usr/ucb:$HOME/bin:/usr/bin/X11:/sbin:.
export PATH
if [ -s "$MAIL" ]           # This is at Shell startup.  In normal
then echo "$MAILMSG"        # operation, the Shell checks
fi                          # periodically.
#
# Run $HOME/sqllib/db2profile
#
. $HOME/sqllib/db2profile
#
# Add MQM sample execution program
#
PATH=${PATH}:/usr/lpp/mqm/samp/bin:${PATH}:/usr/lpp/mqm/inc
PATH=${PATH}:/usr/lpp/db2_01_01_0000/include
export PATH
INCLUDE=/usr/lpp/db2_05_00/include
export INCLUDE
[db2inst1@rs60001]/home/db2inst1 >
```

15. Next, add the DB2 instance user db2inst1 to the mqm group. You may use smitty to do that.

    a. From the smitty menu, select the following:

    ```
    Security and Users
       Groups
          Change / Show Characteristics of a Group
    ```

    b. Type `mqm` as group name.

    c. In the next screen add db2inst1 to the user list as shown below.

    d. You may add dbinst1 to the list of administrators, too.

    e. Logoff

```
                        Change Group Attributes

Type or select values in entry fields.
Press Enter AFTER making all desired changes.

                                                     [Entry Fields]
  Group NAME                                      [mqm]
  Group ID                                        [200]              #
  ADMINISTRATIVE group?                            true              +
  USER list                                       <hugo,otto,jpc,db2inst1] +
  ADMINISTRATOR list                              [mqm,wkshop1,root,jpc,v> +










F1=Help             F2=Refresh          F3=Cancel           F4=List
Esc+5=Reset         F6=Command          F7=Edit             F8=Image
F9=Shell            F10=Exit            Enter=Do
```

2.4 Application Programming Samples

Figure 10 on page 24 shows the application programming samples provided with MQSeries Version 5. They are in the directories:

C	\mqm\tools\c\samples\xatm (amqsxa...)
COBOL	\mqm\tools\cobol\samples\xatm (amq0xa...)
AIX	/usr/lpp/mqm/samp (C and COBOL)

Note: We used modified versions of these programs.

The programs read a message from a queue (under syncpoint), then, using the information in the message, obtain the relevant data from the database and update it. The new status of the database is then displayed.

The program logic is as follows:

1. Use name of input queue from program argument.
2. Connect to the default queue manager (or optionally supplied name in the C program) using MQCONN.
3. Open queue (using MQOPEN) for input while no failures.
4. Start a unit of work using MQBEGIN.
5. Get next message (using MQGET) from queue under syncpoint.

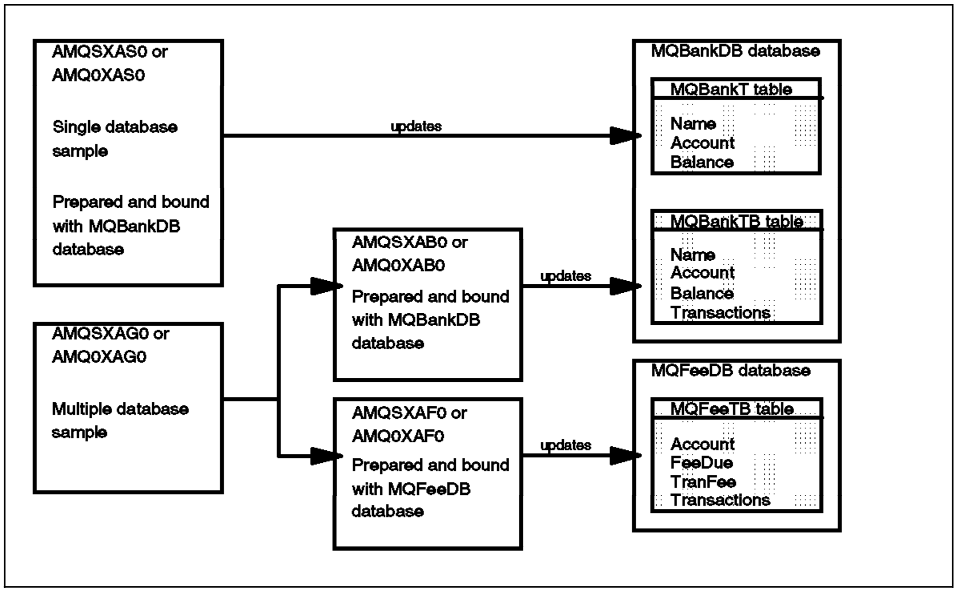


Figure 10. Sample Programs Supplied with MQSeries

6. Get information from databases.
7. Update information from databases.
8. Commit changes using MQCMIT.
9. Print updated information (no message available counts as failure).
10. Close queue using MQCLOSE.
11. Disconnect from queue using MQDISC.

SQL cursors are used in the samples, so that reads from the databases (that is, multiple instances) are locked while a message is being processed; thus multiple instances of these programs can be run simultaneously. The cursors are explicitly opened, but implicitly closed by the MQCMIT call.

The single database samples (AMQSXAS0 and AMQ0XAS0) have no SQL CONNECT statements and the connection to the database is implicitly made by MQSeries with the MQBEGIN call.

The multiple database samples (AMQXSAG0, AMQSXAF0, AMQSXAB0) have SQL CONNECT statements, as some database products allow only one active connection. If this is not the case for your database product, or if you are accessing a single database in multiple database products, the SQL CONNECT statements can be removed.

2.4.1 Operational Considerations

The queue manager can be started and stopped independently of the database managers. The queue manager tolerates any or all of the database managers not being available when it is started. Global transactions can be started but they will not include those database managers that are not available. Applications issuing MQBEGIN will receive the warning 2122, participant not available.

Database managers can be started and stopped independently of the queue manager. A queue manager does not have to restart itself when a database manager becomes unavailable. The availability and unavailability of database managers is reported in the queue manager′s error logs by messages AMQ7604 and AMQ7625.

When a database manager becomes unavailable the possibility exists that it may have updates that are still in doubt. That means, the database manager has been told by the queue manager to prepare to commit, but it hasn′t yet received the outcome of the transaction. When database updates are in-doubt the records that were updated remain locked. So it is important that in-doubt transactions are resolved as quickly as possible.

The queue manager provides two new commands to manage in-doubt transactions:

- The *dspmqtrn* command lists all in-doubt transactions. This also shows the state of all of the participants in the transaction. The state can be either:

  prepared — The resource manager is prepared to commit its updates.

  committed — The resource manager has committed its updates.

  rolled back — The resource manager has rolled back its updates.

  participated — The resource manager is a participant, but has not prepared, committed, or rolled back its updates.

- The *rsvmqtrn* command can be used to instruct the queue manager to resolve any or all in doubt transactions.

The XA coordinator also makes extensive use of the error logs whenever something unexpected occurs. Check for AMQS76xx messages. In particular, look for the following:

AMQ7605 is written whenever a resource manager returns something unexpected from an XA call.

AMQ7606 is written whenever a resource manager rolls back instead of committing.

AMQ7607 is written whenever a resource manager commits instead of rolling back.

2.4.2 The Databases

For the DB2 examples in the book, we create two databases:

- MQBankDB with the tables MQBANKT and MQBANKTB
- MQFeeDB with the table MQFEETB

The database tables contain the initial values shown in Table 4 through Table 6.

Table 4. MQBankDB Database Table MQBANKT

Name	Account	Balance
Mr. Jesse James	1	0
Ms. Lona Lovely	2	0
Mrs. Loretta Lonely	3	0

Table 5. MQBankDB Database Table MQBankTB

Name	Account	Balance	Transactions
Mr. Jesse James	1	0	0
Ms. Lona Lovely	2	0	0
Mrs. Loretta Lonely	3	0	0

Table 6. MQFeeDB Database Table MQFeeTB

Account	Fee Due	Transaction Fee	Transactions
1	0	50	0
2	0	50	0
3	0	50	0

2.4.3 Objectives of the Examples

The remainder of this chapter includes four exercises and database setup instructions. Their purpose is to have you understand the setup and configuration of an XA resource manager and the XA coordinator, and the programming involved. The examples will also point out some of the areas which can cause problems.

First, we explain how to set up the environment. This includes creating the XA-Switch, and configuring both MQSeries and DB2. We will also use a sample program that contains a logical unit of work which includes both MQI calls and SQL calls.

In the second exercise, we modify the sample program to gain a clearer picture of the mechanics of coding a logical unit of work coordinating resources.

The third exercise expands upon the preceding examples by adding the use of a second database to the logical unit of work.

Finally, we will make some changes to the configuration that might occur in real life which should help you understand some of the problems that might occur.

2.5 Exercise 1: Setup for XA Coordination

In this section, we create the databases, queues and the XA-Switch file necessary for the other examples. The XA-Switch file is a DLL or shared library with a single entry point. When called it returns the address of the xa_switch_t structure for the resource manager. The xa_switch_t structure contains the name of the resource manager, option flags and all the XA function pointers.

2.5.1 Creating a Queue for the Examples

Create a queue for the messages that cause database updates. Make sure that the queue manager is running. Under NT and OS/2 logon as *Admin*, and on AIX log in as *mqm*.

```
strmqm
runmqsc
def ql(BANK) usage(normal) defpsist(yes)
end
```

2.5.2 Starting DB2

To work with DB2 use the ID *db2inst* on AIX or *Admin* on NT.

On NT, start DB2 with the following command:

```
db2start
```

On AIX, type the following command to ensure that DB2 is running:

```
ps -ef  grep | db2
```

The following is a sample output for this command:

```
db2inst1 13506 24394  40 09:26:36  pts/5  0:00 ps -ef
db2inst1 15670 25640   0 16:09:26      -  0:07 db2agent (idle)
db2inst1 16430 24870   0 16:09:14      -  0:00 db2resyn
    root 17186     1   0 16:09:12      -  0:00 db2wdog
db2inst1 18266 25640   0 16:10:05      -  0:05 db2agent (idle)
db2inst1 21796 17186   0 16:09:12      -  0:01 db2sysc
db2inst1 22826 21796   0 16:09:13      -  0:00 db2tcpcm
db2inst1 23084 21796   0 16:09:13      -  0:00 db2tcpim
db2inst1 24394 34376   1 08:18:23  pts/5  0:01 -ksh
db2inst1 24870 21796   0 16:09:12      -  0:00 db2gds
db2inst1 25640 21796   0 16:09:13      -  0:00 db2ipccm
db2inst1 32708 24394   3 09:26:36  pts/5  0:00 grep db2
db2inst1 37854 25640   0 16:20:03      -  0:01 db2agent (idle)
```

2.5.3 The DB2 Environment on Windows NT

On Windows NT, you have to enter the DB2 environment before you can work with databases. To start the DB2 Command Line Processor select:

```
Start
   Programs
      DB2 for Windows NT
         DB2 Command Line Processor
```

In the DB2 CLP - db2.exe window, you will see the same information and the prompt db2 =>.

You are now in the DB2 command environment, which has a special setup to issue DB2 commands. You want to be in such a window whenever you want to issue any DB2 commands other than db2start and db2stop.

Note: This default command environment is equivalent to the state you are in after entering runmqsc in MQSeries.

Since the DB2 commands you need to issue have been put into a script file, you can ″pipe″ them rather than type them. To do this you need to quit the command environment to get to a Windows prompt:

```
db2 => quit
DB20000I  The QUIT command completed successfully.
C:\SQLLIB\BIN>
```

2.5.4 Creating the Databases

With DB2 started we can now create the databases for the examples. The three tables are shown in Table 4 on page 26 through Table 6 on page 26. Log on as user Admin on NT or db2inst1 on AIX.

The db.sql file shown in Figure 11 is the input file for DB2. It contains the commands to create the two databases, MQBankDB and MQFeeDB, connect to them one at a time, create the tables, MQBANKT, MQFEETB and MQFEETB, and disconnect.

```
create database MQBankDB
connect to MQBankDB
create table MQBankT(Name VARCHAR(40) NOT NULL, Account INTEGER NOT NULL,
Balance INTEGER NOT NULL, PRIMARY KEY (Account))
create table MQBankTB(Name VARCHAR(40) NOT NULL, Account INTEGER NOT NULL,
Balance INTEGER NOT NULL, Transactions INTEGER, PRIMARY KEY (Account))
disconnect MQBankDB

create database MQFeeDB
connect to MQFeeDB
create table MQFeeTB(Account INTEGER NOT NULL, FeeDue INTEGER NOT NULL,
TranFee INTEGER NOT NULL, Transactions INTEGER, PRIMARY KEY (Account))
disconnect MQFeeDB
```

Figure 11. SQL File to Create Databases

AIX

When you create a database and do not assign a file system DB2 uses by default /home as database file system location. For each database 15 MB disk space is required.

To specify the database directory on AIX, modify the two create statements in Figure 11 as follows:

```
create database MQBANKDB on /home/db2data
create database MQFeeDB on /home/db2data
```

To create the databases, issue the db2 command with db.sql (NT) or dbcreate.sql as input files. While in the DB2 command prompt environment, type the following:

NT `db2 < db.sql`

AIX `db2 < dbcreate.sql`

Note: Make sure that you are in the right directory.

Be patient!

Creating and populating a database takes some time. Several messages will be displayed. Wait until you see the command prompt before you proceed.

2.5.5 Populating the Databases

After the databases are created we need to insert some data. To do this we use the file data.sql shown in Figure 12. This file is also on the diskette. Enter the following command:

NT `db2 < data.sql`

AIX `db2 < data.sql`

```
connect to MQBankDB

insert into MQBankT values ('Mr. Jesse James',1,0)
insert into MQBankT values ('Ms. Lona Loveley',2,0)
insert into MQBankT values ('Mrs. Lorretta Lonely',3,0)

insert into MQBankTB values ('Mr. Jesse James',1,0,0)
insert into MQBankTB values ('Ms. Lona Loveley',2,0,0)
insert into MQBankTB values ('Mrs. Lorretta Lonely',3,0,0)

disconnect MQBankDB

connect to MQFeeDB

insert into MQFeeTB values (1,0,50,0)
insert into MQFeeTB values (2,0,50,0)
insert into MQFeeTB values (3,0,50,0)

disconnect MQFeeDB
```

Figure 12. SQL File Populate Databases

2.5.6 Grant Database Access to Other Users

If your user ID did set up the database, you may skip this step. For a different user, access can be granted the following way.

On Windows NT, type the following commands when in the DB2 environment:

```
C:\SQLLIB\BIN>grant connect on MQBankDB to user <UID>
C:\SQLLIB\BIN>grant connect on MQFeeDB to user <UID>
```

On AIX you can use the file grant.sql shown in Figure 13 and supplied with this book as input for the following command:

db2 < grant.sql

```
connect to mqBankdb
grant connect   on  database  to PUBLIC
grant ALL PRIVILEGES on TABLE db2inst1.mqbankt  to PUBLIC
grant ALL PRIVILEGES on TABLE db2inst1.mqbanktb to PUBLIC
disconnect mqbankdb

connect to mqFeedb
grant connect   on  database  to PUBLIC
grant ALL PRIVILEGES on TABLE db2inst1.mqfeetb to PUBLIC
disconnect mqFeedb
```

Figure 13. SQL File to Grant Access to the Databases

2.5.7 Creating the XA Switch File

Now we need to create the XA Switch file. The source for this file is called db2swit.c and can be found in the following directories:

NT \mqm\tools\c\samples\xatm

AIX /usr/lpp/mqm/samp/xatm

You will also find a make file and a def file which make compilation fairly easy. There is a difference in commands depending on which compiler you are using:

Table 7. Commands to Create XA Switch File

Compiler	Command
Microsoft Visual C/C++	nmake -f xaswit.mak db2swit.dll
IBM Visual Age C/C++	nmake -f xaswiti.mak db2swit.dll
CSet++ for AIX	make -f xaswit.mak db2swit

Before you execute any of these commands read the following platform dependent sections.

Note

The XA Switch file is only available in C.

2.5.7.1 AIX

You need to modify the xaswit.mak file in Figure 14 on page 33 for your DB2 library path before you use it.

```
#
# Name:          XASWIT.MAK
#
# Description:   AIX make file for DB2 and Oracle XA switch files
#
# Statement:     Licensed Materials - Property of IBM
#                84H2000, 5765-B73
#                (C) Copyright IBM Corp. 1997
#
#
# To make the DB2 XA switch load file, run the command:-
#                make -f xaswit.mak db2swit
#
# To make the Oracle XA switch load file, run the command:-
#                make -f xaswit.mak oraswit
#
# Note: If your database libraries are in a different directory
#       to the one listed in the xxxLIBPATH statement, you will
#       need to alter or add that directory to the statement.

#---------------------------------------------------------------------
# DB2 XA switch file
#---------------------------------------------------------------------
DB2LIBS=-l db2
#DB2LIBPATH=-L /usr/lpp/db2_00_00/lib
DB2LIBPATH=-L /usr/lpp/db2_05_00/lib

db2swit:
        $(CC) -e MQStart $(DB2LIBPATH) $(DB2LIBS) -o $@ db2swit.c

#---------------------------------------------------------------------
# Oracle XA switch file
#---------------------------------------------------------------------
ORALIBS=-l clntsh -l m
ORALIBPATH=-L $(ORACLE_HOME)/lib -L /usr/lib

oraswit:
        xlc_r -e MQStart $(ORALIBPATH) $(ORALIBS) -o $@ oraswit
```

Figure 14. Make File to Create XA-Switch on AIX

Log in as root and execute the following commands:

1. cd /usr/lpp/mqm/samp/xatm.
2. cp xaswit.mak db2xaswit.mak
3. vi db2xaswit.mak

   You must change DB2LIBPATH to the path where the DB2 library is stored. The change is highlighted in Figure 14.
4. make -f db2xaswit.mak db2swit
5. cp db2swit /usr/lpp/mqm/bin

2.5.7.2 Windows NT and OS/2

The resulting DLL must be in the path. You may choose where this will reside. For our purposes we will copy it to \mqm\bin.

```
copy db2swit.dll c:\mqm\bin
```

If the user ID that is going to execute the example programs is different from the one that installed DB2 you have to grant authority as described in 2.5.6, "Grant Database Access to Other Users" on page 30.

The last part of setup is to tell DB2 the name of the DLL for dynamic registration purposes. This is the file MQMAX.DLL that comes with MQSeries. The file must be somewhere in a LIBPATH included directory. This step is only required on NT and OS/2. If you forget to do it you will get a DB/2 error referring to ax_reg. This (ax_reg) tells the queue manager that the resource manager is a participant in the current transaction.

```
db2 => update dbm cfg using TP_MON_NAME mqmax
db2 => disconnect MQBANKDB
db2 => quit
```

2.5.8 You Need UTIL.C from DB2

To compile the sample programs you need the routines in the file UTIL.C. This program uses the header file UTIL.H. You find the files in the following directories:

NT	\sqllib\samples\c
AIX	/usr/lpp/db2_05_00/samples/c

Note

UTIL.C is only available in C, *not* in COBOL.

To compile the program under Windows NT, it may be easier to copy the util.h header file into the DB2 include library. Issue the following command:

copy c:\sqllib\samples\c\util.h c:\sqllib\include

2.5.8.1 Creating the Object File for UTIL.C

The code is shipped with DB2 and will be linked to the MQSeries sample programs. Depending on the compiler you use, issue one of the following commands to create the object file:

Table 8. Commands to Compile UTIL.C

Compiler	Command
Microsoft Visual C/C++	cl util.c /C
IBM Visual Age C/C++	icc util.c /C
CSet++ for AIX	xlc -c util.c -I /usr/lpp/db2_05_00/sample/c

You may also include the above in the make files that compile the sample programs.

2.6 Hints for Working with the Databases

In this section, we describe some functions and provide some files that will help you go through the examples in this chapter. The hints are intended for programmers who want to test their MQSeries/DB2 programs but have little DB2 experience.

2.6.1 Open a DB2 Command Window on Windows NT

You have two options to open a window that is initialized for DB2:

1. DB2 Command Window

   ```
   Start
      Programs
         DB2 for Windows NT
            DB2 Command Window
   ```

2. DB2 Command Line Processor

   ```
   Start
      Programs
         DB2 for Windows NT
            DB2 Command Line Processor
   Type quit
   ```

2.6.2 Using SQL Command Files

1. Start DB2 with the command **db2start**.
2. Open a DB2 Command Window (NT).
3. Type **db2 < db.sql > db.out**.

   This will run the SQL statements in db.sql against the DB2 engine and write the DB2 responses to the file db.out.

Note: Start DB2 before you run any SQL commands. Otherwise, you will get an error message.

2.6.3 Lookup Information in a Database

You can connect to a database and look at the table entries at any time. In a DB2 command window, type the following commands:

```
DB2
connect to MQBANKDB
select * from MQBANKT
quit
```

You can also use an input file that contains SQL commands. The file select.sql in Figure 15 is an example. It displays the contents of the tables in both databases. Enter the following command:

NT **db2 < select.sql**

AIX **db2 < select.sql**

```
connect  to       MQBankDB
select   *   from MQBankT
select   *   from MQBankTB
disconnect        MQBankDB

connect  to       MQFeeDB
select   *   from MQFeeTB
disconnect        MQFeeDB
```

Figure 15. SQL File to View the Databases

2.6.4 Drop a Table

The following is an example of deleting the tables in the two databases. You may enter the commands by hand or use the file tbldrop.sql on the diskette.

```
connect  to       MQBankDB
drop     table    MQBankT
drop     table    MQBankTB
disconnect        MQBankDB

connect  to       MQFeeDB
drop     table    MQFeeTB
disconnect        MQFeeDB
```

Figure 16. SQL File to Drop Database Tables

2.6.5 Drop a Database

If you want to clean out the database, use the following commands:

drop database MQBankDB
drop database MQFeeDB

You may put the commands into an sql file (dbdrop.sql on the diskette).

2.6.6 Monitor Database Connections on Windows NT

You can use the DB2 Database Director to monitor commits and backouts. To start the Database Director select **Start**, **Programs**, **DB2 for Windows NT** and then **Database Director**. Make sure that the database manager is running (db2start). The Database Director window is shown in Figure 17.

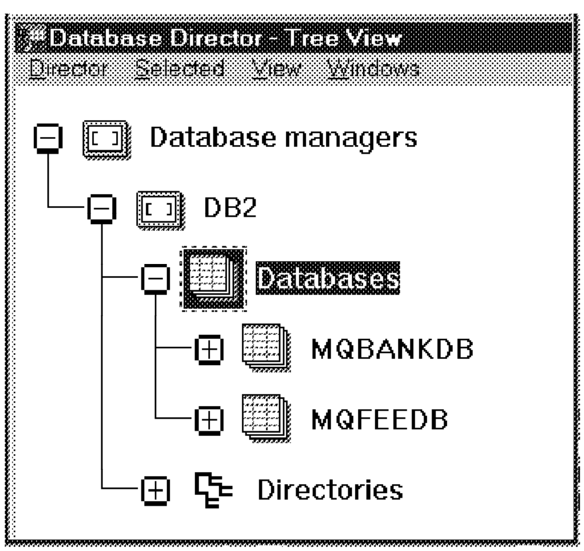


Figure 17. Database Director - Tree View

The following steps explain how to use it:

1. Expand the DB2 icon and then the Databases icon.
2. Select the database you want to monitor and click on it with the right mouse button. In this example, we use MQBANKDB.
3. Select **Start monitoring** from the pop-up menu. This brings up the Snapshot Monitor window shown in Figure 18.
4. *Minimize* the Database Director - Tree View.

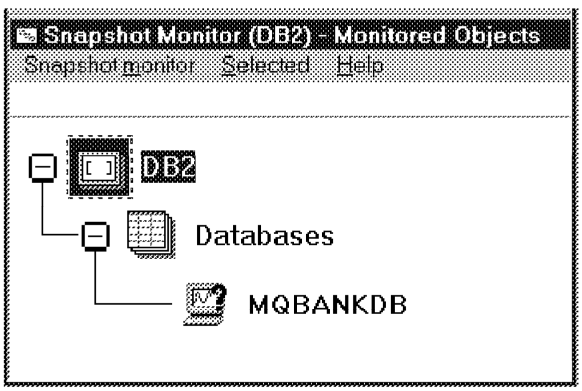


Figure 18. Snapshop Monitor (DB2) - Monitored Objects

5. In the Snapshot Monitor window, double-click on **MQBANKDB**.

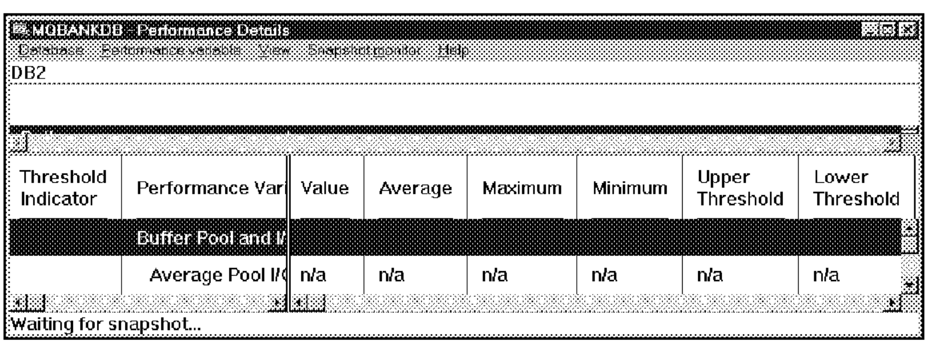


Figure 19. Performance Details Window

6. From the View menu, select **Include performance variables**
7. Select all *Displayed* items in Figure 20 on page 39 and remove them with the **<** button.

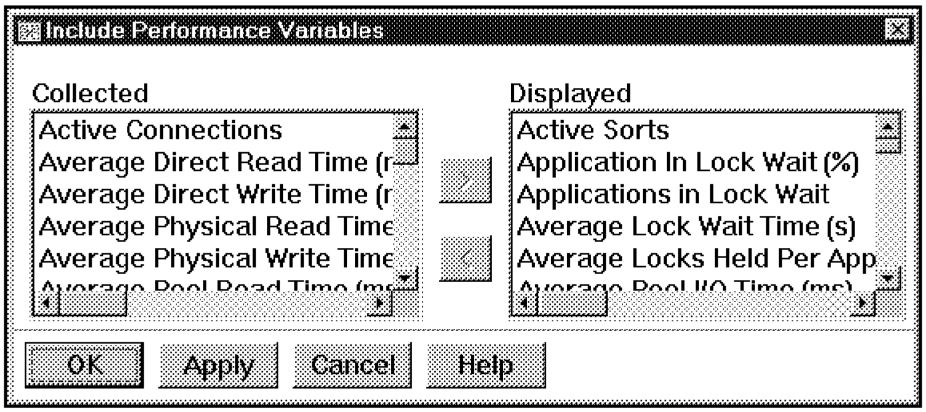


Figure 20. Performance Variables Window

8. From the Collected list on the left, select the following items and move them into the Displayed list by clicking on ″>″.
   - Commits Attempted
   - Rollbacks Attempted
9. Click on **Apply** and then on the **OK** button. The customized Performance Details window is shown in Figure 21.

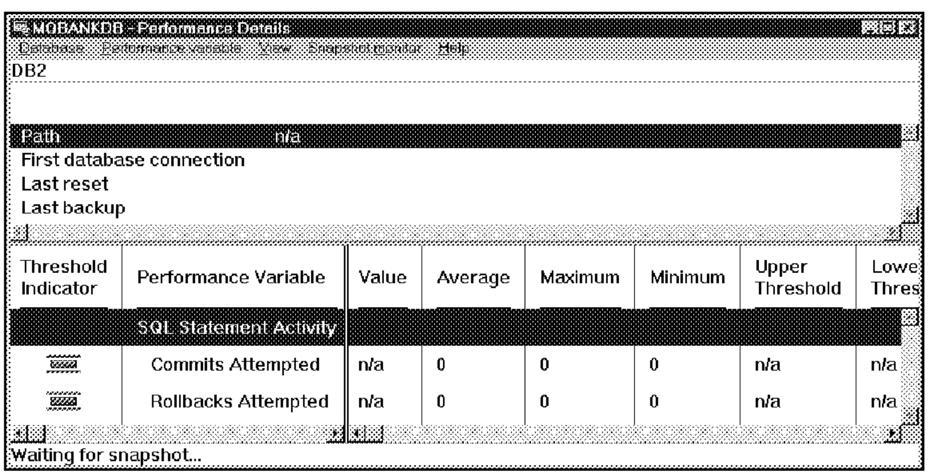


Figure 21. Customized Performance Details Window

10. You can now minimize all DB2 windows except the Performance Details window shown above.

2.7 Exercise 2: Using One XA Resource

This example is based on the program amqsxas0, which is supplied with MQSeries. The program logic is described in 2.4, "Application Programming Samples" on page 23.

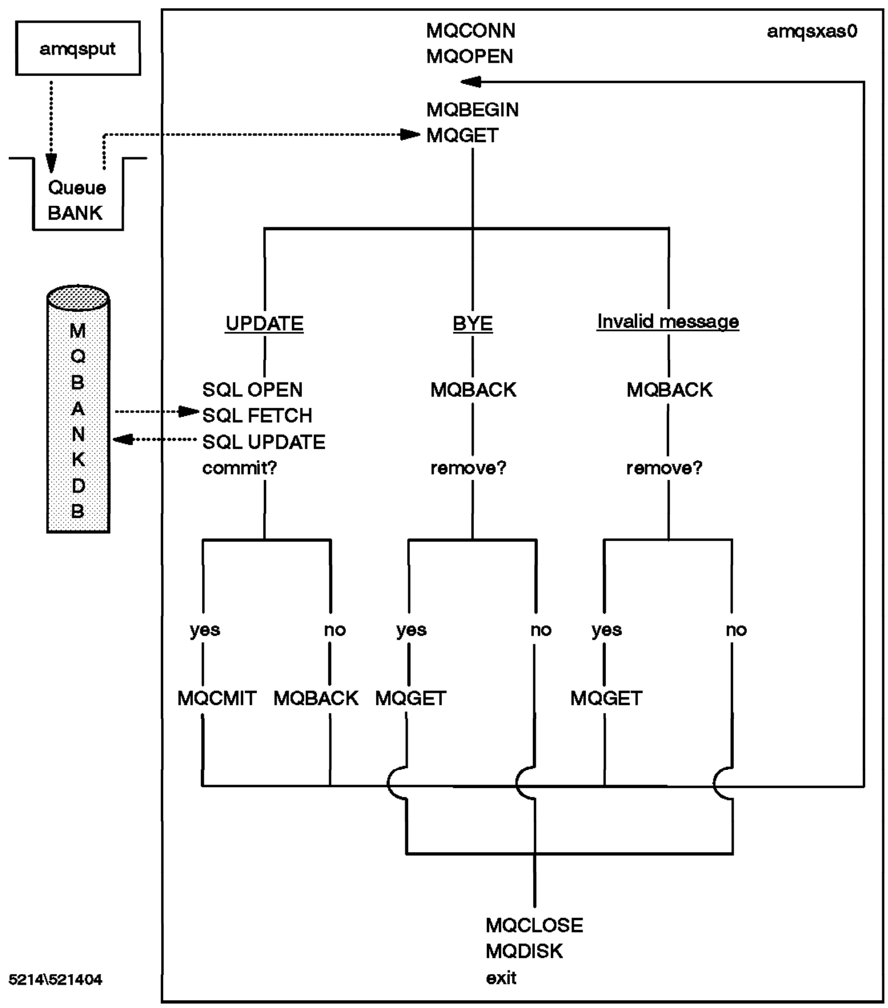


Figure 22. Program Logic of Modified Sample AMQSXAS0

We use the program amqsput to put messages on the queue MQBANK we created earlier (refer to 2.5.1, "Creating a Queue for the Examples" on page 27). The program amqsxas0 reads the message from the queue and processes it. There are three types of messages:

1. Valid messages that update the database have the following form:

   ```
   UPDATE Balance change=$ WHERE Account=n
   ```

   The command is case-sensitive. After the update, the program then asks you if you want to commit or back out the transaction. Backed out messages remain in the queue.

2. A message with the text `BYE` ends the program. This message will automatically be backed out. You will be asked whether the message shall remain in the queue or not. In any case, the program will end.

3. Any other text constitutes an invalid message that is automatically backed out. You are asked whether it should be removed from the queue or not. If you don′t remove the message the program will end.

2.7.1 Building an Executable for Windows NT

The following explains how to build the executable for amqsxas0 on a Windows NT system. Remember, every program must be pre-compiled and bound by DB2.

The sample program source amqsxas0.sqc can be found in the following directories:

C \mqm\tools\c\samples\xatm (sqc)

COBOL \mqm\tools\cobol\samples\xatm (sqb)

Note: We use a modified version of the program.

Included in the samples you will find .bat files which bind and compile the programs. Compile the sample program with one of the following commands:

Table 9. Commands to Build Executable of AMQSXAS0

Compiler	Command
Microsoft Visual C/C++	msmake amqsxas0 MQBANKDB |more
IBM Visual Age C/C++	ibmmake amqsxas0 MQBANKDB |more

The parameters of the make file are:

amqsxas0 Source Code file name, without extension (.SQC)

MQBANKDB Database to be used for binding

Got an error? If you are trying this on your system and you get an error it may be that you are executing this in a standard NT window. Use the DB2 Command Window instead.

```
rem ------------------------------------------------------------
rem - Build-File for C (/ C++) Programs w. MS Vis.C++ Compiler -
rem -          (W) 10/08/1997 M.Schuette, IBM                 -
rem - Builds Progr. w. (DB/2) embedded SQL and MQSeries Calls  -
rem - For use with MS VC++ 2.0+, DB/2 V2.1.2+, MQSeries V5+    -
rem ------------------------------------------------------------
rem Usage: msmake <prog_name> <db_name> <addlibs>

db2 connect to %2
db2 prep %1.sqc bindfile
db2 bind %1.bnd
db2 connect reset

rem Compile and link the program.
rem ????? To build a C++ program, change the source file extension to '.cxx'
rem and include the -Tp option.
rem Include other libraries at the end of the link-command !

cl -Z7 -Od -c -W2 -D_X86_=1 -DWIN32 -I%DB2PATH%\include %1.c
link -debug
:full -debugtype:cv -out:%1.exe %1.obj util.obj db2api.lib mqm.lib %3
```

Figure 23. Make File for Microsoft C Compiler on Windows NT

```
rem ------------------------------------------------------------
rem - Build-File for C (/ C++) Programs w. IBM Vis.C++ Compiler -
rem -          (W) 10/08/1997 M.Schuette, IBM                 -
rem - Builds Progr. w. (DB/2) embedded SQL and MQSeries Calls  -
rem - For use with IBM VA/C++ 3.5+, DB/2 V2.1.2+, MQSeries V5+    -
rem ------------------------------------------------------------
rem Usage: ibmmake <prog_name> <db_name> <addlibs>

db2 connect to %2
db2 prep %1.sqc bindfile
db2 bind %1.bnd
db2 connect reset

rem Compile and link the program.
rem ????? To build a C++ program, change the source file extension to '.cxx'
rem and include the -Tp option.
rem Include other libraries at the end of the link-command !

icc /Gm /Ti- %1.c /C
ilink  %1.obj util.obj db2api.lib mqm.lib %3
```

Figure 24. Make File for IBM C Compiler on Windows NT

2.7.2 Building an Executable for AIX

The following explains how to build the executable for amqsxas0 on an AIX system. Remember, every program must be pre-compiled and bound by DB2.

The C and COBOL sample program source amqsxas0.sqc can be found in the following directory:

`../usr/lpp/mqm/samp/xatm (sqc and sqb)`

Note: For our example, we modified the sample program provided with MQSeries.

Below you find a shell file and a make file you can use to compile the programs under AIX. To compile the program follow these steps:

Step 1. Log in as db2inst1 or mqm.

Step 2. Create the shell file in Figure 25: vi amqsxas0.sh.

Step 3. Create the make file in Figure 26 on page 44: vi amqsxas0.mak.

Step 4. chmod + x amqsxas0.sh.

Step 5. amqsxas0.sh (the shell file calls the make file).

```
#------------------------- amqsxas0.sh ----------------------------#
#                                                                  #
#  AIX MQSeries Link DB2 Application program                       #
#  Make file for connect DB2 DataBase                              #
#  1) Call db2 to pre-compile .sqc                                 #
#  2) Call makefile named is amqsxas0.mak to generate executable   #
#                                                                  #
#------------------------------------------------------------------#
#
echo Connect to DB2 DataBase MQBANKDB
#
db2 connect to MQBANKDB
db2 prep amqsxas0.sqc bindfile
db2 bind amqsxas0.bnd
db2 connect reset
#
echo Call make -f amqsxas0.mak
make -f amqsxas0.mak
```

Figure 25. Shell File AMQSXAS0.SH to Build an Executable on AIX

```
#*********************************************************************
#*                                                                   *
#*  amqsxas0.mak: Source file generated by the Class Compiler        *
#*                11/29/95     20:39:48 language = C                 *
#*                                                                   *
#*********************************************************************
.SUFFIXES:
.SUFFIXES: .o .c

CC = xlc

OBJS= amqsxas0.o util.o
CFLAGS = -g -c  -I/usr/lpp/db2_05_00/samples/c /usr/lpp/mqm/inc
#CFLAGS = -g -c  -I/usr/lpp/mqm/inc
#CFLAGS = -g -c -Dsigned= -Dvolatile= -D_Optlink -I. -M
#LFLAGS = -L. -lXm -lXt -lX11 -L/usr/lpp/mq3t/lib -lbmqapic -e LibMain -bM:SRE
#--------------------------------------------------------------------
# MQM Library file and seraching path
#--------------------------------------------------------------------
MQMLIBS=-l mqm
MQMLIBPATH=-L /usr/lpp/mqm/lib
DB2LIBS=-l db2
DB2LIBPATH=-L /usr/lpp/db2_05_00/lib

#HEADERS = /usr/lpp/mqm/inc /usr/lpp/db2_05_00/samples/c
#HEADERDB2 = /usr/lpp/db2_05_00/samples/c

.c.o:
        $(CC) $(CFLAGS) $<

all: amqsxas0

util.o: util.c\
        $(HEADERS)

amqsxas0.o: amqsxas0.c\
        $(HEADERS)
#
# Link all Object files
#
amqsxas0:  $(OBJS)
        $(CC) -o amqsxas0 $(MQMLIBPATH) $(MQMLIBS) $(DB2LIBPATH)\
        $(DB2LIBS) $(OBJS)
```

Figure 26. Make File AMQSXAS0.MAK to Build an Executable on AIX

2.7.3 Define the Database to MQSeries

To define the database, that is, an XA resource manager to the queue manager, we have to add a stanza to the qm.ini file. The stanza and an example are described in 2.2.2.2, “The XA Resource Manager Stanza” on page 13.

1. Make sure that the queue manager is stopped. Stop MQSeries immediately with this command:

   endmqm /i <QMgrName>

2. Define the XA resource manager

   This must be done once per DB2 database that is to be accessed by MQSeries. The queue manager needs some details about the database it will use as an external XA resource manager. You specify these details in the queue manager's qm.ini file in the directories:

   NT \mqm\qmgrs\<QmgrName>

   AIX /var/mqm/qmgrs/<QmgrName>

   For this example, fill in the values as follows:

   Windows NT

```
XAResourceManager:
      Name=DB2 MQBANKDB
      SwitchFile=c:\mqm\bin\db2swit.dll
      XAOpenString=MQBANKDB
      ThreadOfControl=THREAD
```

   AIX

```
XAResourceManager:
      Name=DB2 MQBANKDB
      SwitchFile=/usr/lpp/mqm/bin/db2swit.dll
      XAOpenString=MQBANKDB
```

   Notes:

   a. In this example we use only one database.

   b. The database manager name is DB2 and the name of the database is MQBANKDB.

   c. Make sure that the path for db2swit.dll is correct.

   d. For XAOpenString enter the database name MQBANKDB.

e. For DB2 on OS/2 and Windows, the ThreadOfControl parameter is always THREAD. Omit this line if you configure MQSeries and DB2 on a UNIX system.

2.7.4 What Happens when MQSeries Starts but not DB2

Before you start the queue manager clear the error log AMQERR01.LOG in the queue manager's directory, such as:

`\mqm\qmgrs\MyQMgr\errors`

If you are doing this on your system and you are sure that you don't need it, simply erase it and MQSeries will create a new one at startup. For this exercise, follow these steps:

Step 1. Make sure DB/2 is stopped. Issue the command `db2stop`

Step 2. Now start MQSeries: `strmqm [QmgrName]`

You see the message `MQSeries queue manager started`.

Step 3. Now browse the error log using Notepad or whatever other editor you prefer. You will see several messages. The first is as follows:

```
AMQ7604: The XA resource manager 'DB2 MQBANKDB' was not available
when called for xa_open. The queue manager is continuing without
this resource manager.
```

There is more to this message and associated actions to try. The main suggestions are that either the resource manager we named in our qm.ini file was incorrect (spelling, whatever) or it was simply not available at the startup. Since we had earlier done a db2stop the latter suggestion is a strong possibility.

Note: The queue manager does come up. In the error log, you can see the message `AMQ8003 MQSeries queue manager started`. It simply cannot participate in any transactions with the named resource.

Step 4. Now we'll take a look at what happens when you try to execute the program amqsxas0 in this state. Execute the following command:

```
amqsxas0 BANK
Target queue is BANK
MQBEGIN ended with reason code 2122
```

First of all, notice we are using a new MQSeries call: MQBEGIN. This is used to start a logical unit of work. If we look in the *MQSeries Application Programming Reference, SC33-1673-03* we see:

```
MQRC_PARTICIPANT_NOT_AVAILABLE        2122
```

Step 5. There is an easy solution to this problem. Issue the following command, which should make our participant available:

```
db2start
```

Step 6. If you start the program again you will see messages displayed by the program:

```
amqsxas0 BANK
Target queue is BANK
Unit of work started
```

Step 7. Press Ctrl+Break to end the program.

2.7.5 Executing the Sample Program

In order to use the amqsxas0 sample program, it is first necessary to put some messages on the queue, and we will use amqsput to do that.

Step 1. Start the sample again, with DB2 available this time.

```
db2start
SQL1063N  DB2START processing was successful.
strmqm
MQSeries queue manager started.
amqsxas0 BANK
Target queue is BANK
Unit of work started
```

Step 2. The transaction program is now waiting for a message. We use the sample program amqsput to send messages to it. The amqsxas0 program expects its messages in a certain format. If you try this on your own do not deviate from the format - it won't work.

The rows in our MQBANKT table are as follows:

Account Number	Account Holder	Initial Balance
1	Mr. Jesse James	100
2	Ms. Lona Lovely	100
3	Mrs. Loretta Lonely	100

In the NT command prompt window we enter:

```
C:\>amqsput BANK
Sample AMQSPUT0 start
target queue is BANK
UPDATE Balance change=50 WHERE Account=3
UPDATE Balance change=50 WHERE Account=2
```

Step 3. While in the DB2 prompt window where we have amqsxsa0 running we have the following:

```
C:\Sample2>amqsxas0 BANK
Target queue is BANK
Unit of work started
Account No 3 Balance updated from 100 to 150 Mrs. Lorretta Lonely
Do you want to commit this Update [Yes|No] ?
y
Unit of work successfully completed
Unit of work started
Account No 2 Balance updated from 100 to 150 Ms. Lona Loveley
Do you want to commit this Update [Yes|No] ?
y
Unit of work successfully completed
Unit of work started
```

Step 4. The program is now coordinating a logical unit of work, or a transaction, between MQSeries and DB2. In this case MQSeries is acting both as a resource manager (as is DB2) and a resource coordinator.

You may commit or back out the unit of work. If you back out, of course, the transaction remains in the queue.

Step 5. Messages that contain invalid data are automatically backed out. They remain in the queue. You can remove them from the queue by answering *y* when the following question appears:

```
Remove BACKOUTed message [Yes | No] ?
```

Step 6. To end the program send a message that contains **BYE**. This message will be backed out. The program ends regardless of whether you remove the message from the queue or not.

2.7.6 Monitoring Database Transactions

You can use the DB2 Performance Monitor to find out how many commits and backouts have been issued. How to start this utility is described in 2.6.6, "Monitor Database Connections on Windows NT" on page 37.

Use the sample program amqsput to send the message below to amqsxas0:

```
C:\>amqsput BANK
Sample AMQSPUT0 start
target queue is BANK
UPDATE Balance change=25 WHERE Account=1
```

The program amqsxas0 runs in a DB2 Command window and should display the following:

```
C:\Sample2>amqsxas0 BANK
Target queue is BANK
Unit of work started
Account No 1 Balance updated from 0 to 25 Mr. Jesse James
Do you want to commit this Update [Yes|No] ?
y
Unit of work successfully completed
Unit of work started
```

After a while, the Performance Details window will be updated and show the following information:

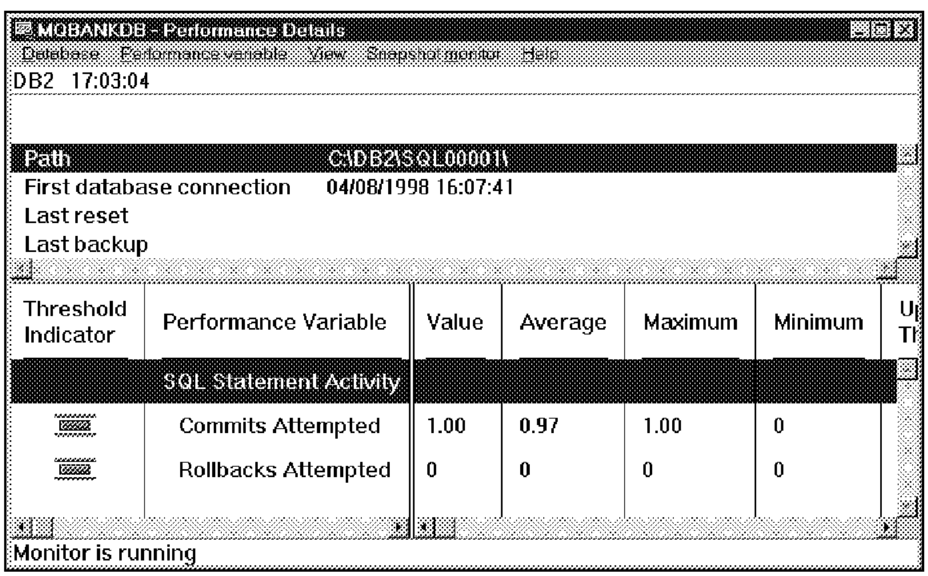


Figure 27. Performance Details Window Showing a Committed Transaction

The default interval for updating the window is 20 seconds. You can change this value. From the Snapshot monitor menu, select **Open as settings**. In the subsequent window change the capture interval.

2.8 Exercise 3: Understanding Backout

In the program we executed in the previous example messages remain on the queue even though you decide to back out the unit of work. Notice what happens when we choose not to commit the unit of work:

```
C:\Sample1>amqsxas0 BANK
Target queue is BANK
Unit of work started
Account No 1 Balance updated from 0 to 25 Mr. Jesse James
Do you want to commit this Update [Yes|No] ?
n
MQBACK successfully issued
Unit of work started
Account No 1 Balance updated from 0 to 25 Mr. Jesse James
Do you want to commit this Update [Yes|No] ?
n
MQBACK successfully issued
Unit of work started
Account No 1 Balance updated from 0 to 25 Mr. Jesse James
Do you want to commit this Update [Yes|No] ?
```

The message stays on the queue and the program loops back to get it again. In a testing environment this may not be bad, since you can go in manually and clear the queue. In a production environment you would want the program to do something to bypass the message, or to check how often a message had been backed out of a logical unit of work.

In the next example we will take the same program, modify it to check if a message has been backed out more than twice, then remove the message from the queue displaying a message that confirms the deletion. Furthermore, we end the program only with a BYE command. Figure 28 on page 51 shows the program logic.

2.8.1 Information about Backout

Our first task might be to check out some of the documentation regarding backout. This can be found in the following documents:

- *MQSeries Application Programming Reference*, SC33-1673
- *MQSeries Command Reference*, SC33-1369

The *MQSeries Application Programming Guide, SC33-0807-07* contains a general description of syncpointing, which includes backout.

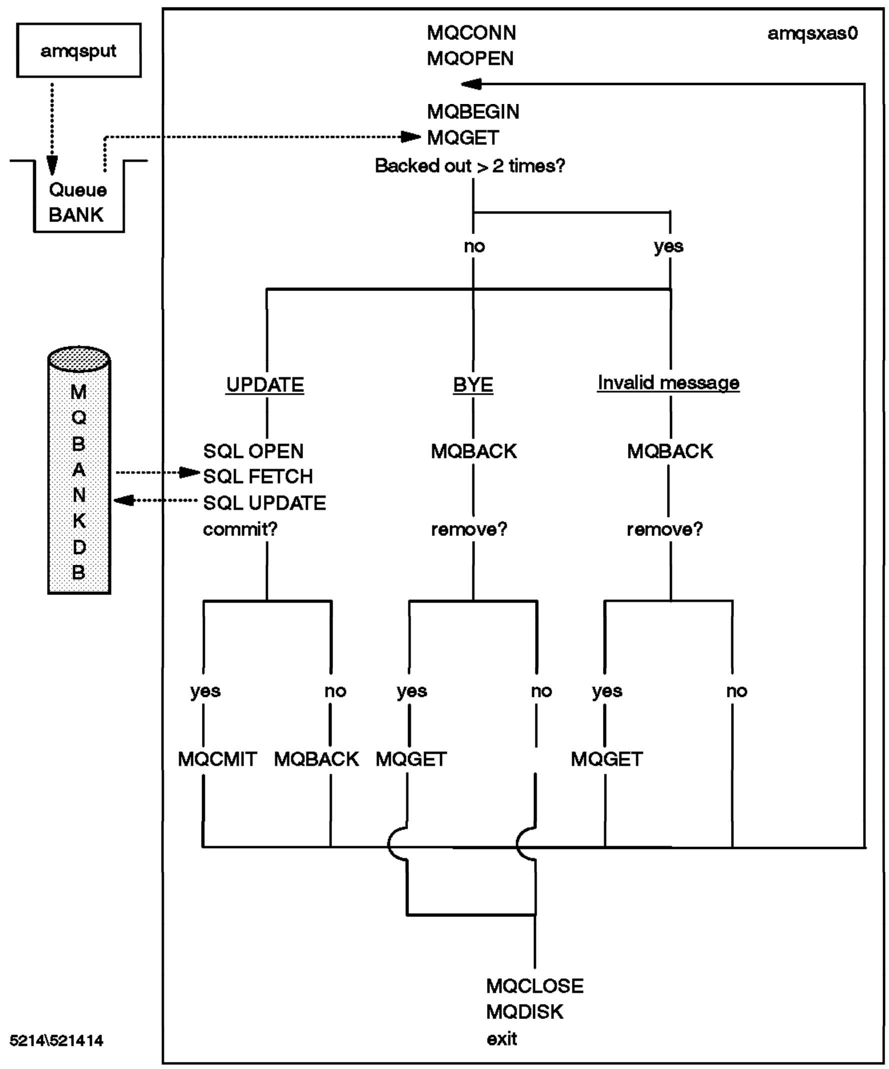


Figure 28. Program Logic of Example AMQSXAS1

Information about backout is maintained in the following fields:

- BackoutThreshold (BOTHRESH) in local queue definition
- BackoutRequeue (BOQNAME) in local queue definition
- BackoutCount field in the message descriptor (md)

The BackoutCount field contains the number of times a message has been backed out. The queue manager does nothing with the threshold and requeue fields except maintain them. An application could inquire as to their values and make decisions based on those values and the value of the backout count.

In our sample we make changes that check to see if a message has been backed out more than twice, then ask if it should be removed and act accordingly.

2.8.2 Program Logic

The program understands three messages:

1. Valid messages contain valid UPDATE commands, such as:

   ```
   UPDATE Balance change=25 WHERE Account=1
   ```

   Valid commands can be committed or backed out. Backed out messages remain in the queue, and the program does not end.

   Note: The message text is case-sensitive.

   If any of the MQI calls or SQL commands cause an error, the unit of work will automatically be backed out and the program ends.

   When a valid message has been backed out more than two times it is treated like an invalid message and backed out automatically. The user decides whether the message remains in the queue or is removed.

2. The special message to end the program is:

   ```
   BYE
   ```

   This message is processed like an invalid command and backed out. The user can keep the message in the queue or remove it.

   Note: This message text is case-sensitive, too.

3. Any other message is invalid and backed out automatically. The operator can choose to leave the message in the queue or to remove it. The program continues and either reads the same message again or waits for another one to arrive.

Figure 29 on page 53 shows the sequence of the MQI calls and SQL commands. The program terminates under one of the following conditions:

- If any of the MQI or SQL calls fails
- If a BYE message has been processed
- If the MQCMIT fails

```
             Valid messages                  Invalid messages
             --------------                  ----------------
SQL DECLARE CURSOR
                 ok        not_ok
             MQCONN        |
                 ok        not_ok
             MQOPEN        |
                 ok        not_ok
         begin loop        |
            MQBEGIN        |
                 ok        not_ok
              MQGET        |
                 ok        not_ok
      backout > 2 ?        |
                 no        yes
 invalid command ?         |
                 no        yes  -------------->|
   SQL OPEN CURSOR                             MQBACK
                 ok        not_ok              |
          SQL FETCH        |                   remove from queue?
                 ok        not_ok              yes              no
         SQL UPDATE        |                   MQGET             |
                 ok        not_ok              ok          not_ok
           commit ?        |                      BYE message?
                yes        no                     yes       no
                  |        MQBACK                  |        loop
                  |        loop                    |
             MQCMIT                                |
                 ok        not_ok --------------> MQCLOSE
               loop                               MQDISC
                                                  exit
```

Figure 29. SQL Calls in Example AMQSXAS1

2.8.3 Writing the Sample Program

The complete source code is listed in Appendix A, “Example Using One XA Resource” on page 175, and you will also find it on the diskette. Here we describe the important MQI calls and SQL commands used in amqsxas0.sql and amqsxas1.sql.

Note: Both programs are *modified versions* of the sample program supplied with MQSeries.

First, we have to declare the table MQBANKT of the database MQBankDB. The table contains three fields as shown in Figure 30 on page 54.

```
EXEC SQL BEGIN DECLARE SECTION;
char name[40];
long account;
long balance;
EXEC SQL END DECLARE SECTION;
```

Figure 30. Code to Declare a Database

The cursor for locking the reads from the database is declared as follows:

```
EXEC SQL DECLARE cur CURSOR FOR
         SELECT Name, Balance
         FROM MQBankT
         WHERE Account = :account
         FOR UPDATE OF Balance;
```

Figure 31. Code to Declare a Cursor for Locking Reads from a Database

After connecting to a queue manager and opening the input queue, the program performs a loop getting messages from the queue until there is a failure. First, a global unit of work is started:

```
MQBEGIN (hCon, &bo, &compCode, &reason);
if (reason == MQRC_NONE)
   printf("Unit of work started\n");
else {
   printf("MQBEGIN ended with reason code %li\n", reason);
   rc = NOT_OK;                      /* stop reading messages */
}
if (compCode == MQCC_FAILED)
   printf("Unable to start a unit of work\n");
```

Figure 32. Code to Start a Global Unit of Work

The begin options (&bo) are set to MQBO_DEFAULT. Currently, there are no other options.

The MQBEGIN call implicitly makes the connection to the database specified in the XA resource manager stanza in the qm.ini file.

An MQGET with an unlimited wait interval gets a message from the queue. The code for the get is shown in Figure 33 on page 55.

```
memcpy(md.MsgId, MQMI_NONE, sizeof(md.MsgId));
memcpy(md.CorrelId, MQCI_NONE, sizeof(md.CorrelId));
msgBufLen = sizeof(msgBuf) - 1;
gmo.Options = MQGMO_WAIT + MQGMO_CONVERT + MQGMO_SYNCPOINT;
gmo.WaitInterval = MQWI_UNLIMITED;
MQGET(hCon, hObj, &md, &gmo, msgBufLen, msgBuf, &msgLen,
      &compCode, &reason);
```

Figure 33. Code of MQGET with Unlimited Wait

The update of the database is done with the statements in Figure 34.

```
EXEC SQL OPEN cur;
CHECKERR ("OPEN CURSOR");
EXEC SQL FETCH cur INTO :name, :balance;
CHECKERR ("FETCH");

balance += balanceChange;
EXEC SQL UPDATE MQBankT SET Balance = :balance
                        WHERE CURRENT OF cur;
CHECKERR ("UPDATE MQBankT");
printf ("Account No %li Balance updated from %li to %li %s\n",
         account, balance - balanceChange, balance, name);
```

Figure 34. Code to Update a Database

After that the program prompts you to either commit or back out the transaction. One of the following two statements will be executed:

```
MQCMIT(hCon, &compCode, &reason);
MQBACK(hCon, &compCode, &reason);
```

The sample amqsxas1 is based on amqsxas0 from the previous exercise. It contains the changes to automatically back out a unit of work when the message has been backed out three or more times.

The queue manager increases the field BackoutCount in the message header each time the message has been backed out as part of a unit of work. The code in Figure 35 on page 56 checks this count in each message, regardless of whether the message is valid, invalid or contains BYE.

```
if (compCode != MQCC_FAILED && rc == OK){

   if (md.BackoutCount > 2) {
      printf("The following message has been backed out %li times.
              md.BackoutCount);
      printf("%s\n",msgBuf);
      rc = NOT_OK;                 /* Bypass database update */
      invCmd = 1;                  /* Ask whether to delete messag
   }
:
}
```

Figure 35. Code to Check if a Message Has Been Backed Out

Setting the rc to NOT_OK bypasses all code between this point and the MQBACK. The MQBACK is executed and the message remains in the queue.

Setting invCmd to 1 causes the program to ask whether the message has to be removed from the queue or remain there.

2.8.4 Compiling the Sample Program

Compile the program in the same way as you did amqsxas0.sqc. For this exercise, too, we use only the database MQBankDB. Use one of the following commands in Table 10.

Note: For AIX, change the program name in the shell and make files from amqsxas0 to amqsxas1.

Table 10. Commands to Compile amqsxas1

Compiler	**Command**
Microsoft Visual C/C++	msmake amqsxas1 MQBANKDB |more
IBM Visual Age C/C++	ibmmake amqsxas1 MQBANKDB |more
CSet++ for AIX	chmod +x amqsxas0.sh amqsxas1.sh (The shell file calls the make file.)

2.8.5 Executing the Sample Program

To test this application, you need (besides amqsxas1) the sample program amqsput and the DB2 Performance Monitor.

1. Make sure that DB2 and the queue manager are started. At any command prompt type:

db2start
strmqm

2. Open a DB2 Command Window and execute the following command:

   amqsxas1 BANK

3. In another DB2 Command Window or in a Command Prompt window start the program that puts commands for amqsxas1 in the queue BANK:

   amqput BANK

4. Open the Database Director as described in 2.6.6, “Monitor Database Connections on Windows NT” on page 37.

5. Use amqsput to place some valid and invalid messages on the queue and watch what happens in the amqsxas1 window.

   - Commit and back out some messages.
   - Use runmqsc to check how many messages are in the queue.
   - Use amqsbcg to check the backout count in the message header.
   - DB2 reports in the Performance Details window the number of commits and rollbacks. Figure 36 shows an example.

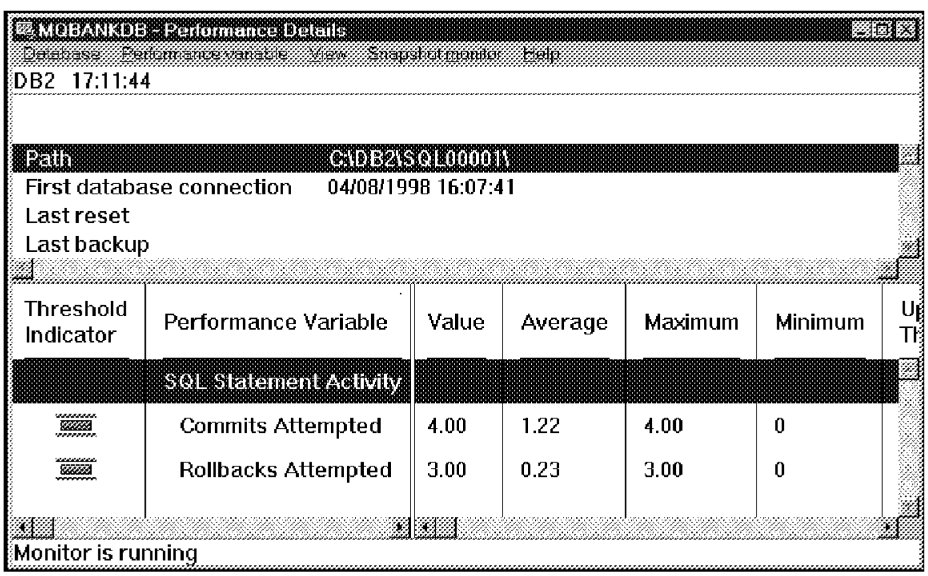


Figure 36. Performance Details Window with Committed or Rolled Back Transactions

2.9 Exercise 4: Using Two XA Resources

In a real-world environment there is often more than one database associated with a transaction. In this section, we explain how to connect a program to two databases and how to tell MQSeries about it.

To demonstrate this function MQSeries provides three sample programs:

AMQSXAB0.SRC contains functions that access table MQBankTB in database MQBANKDB.

AMQSXAF0.SRC contains functions that access table MQFeeTB in database MQFEEDB.

AMQSXAG0.C reads messages from the queue BANK and calls the functions in the other programs to update the databases.

The application consists of three programs because you can′t bind one program to more than one DB2 database at a time.

Note: For this exercise, we used a modified version of amqsxag0.c.

You can use this exercise to learn:

- How programs and databases behave when transactions are entered
- How to make the second resource known to the queue manager
- How to use the Database Director to monitor the events

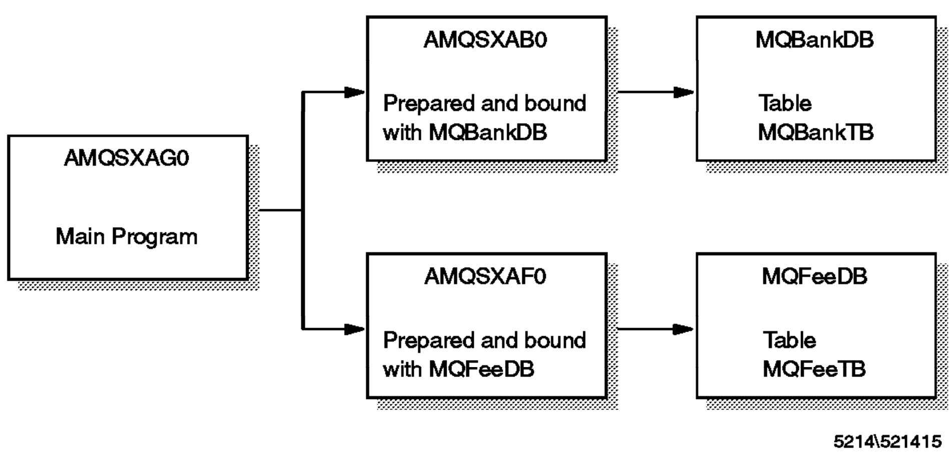


Figure 37. Updating Multiple Databases

2.9.1 Program Logic

This application consists of three programs and two databases with one table each. For the big picture, refer to Figure 37 on page 58.

The main program, amqsxag0.c, is similar to the program used in the first exercise. It waits an unlimited time for a message and understands the same input:

- `UPDATE Balance change=nnn WHERE Account=mmm`
- `BYE`
- `Any other stuff is rejected`

Note: The first two are valid commands and case-sensitive!

With the UPDATE command, you change the account balance in MQBANKDB and increase the fee amount in MQFEEDB.

With the BYE command you end the application.

Any other stuff you may type will be treated as an invalid message and backed out.

All SQL commands are removed from the main program amqsxag0.c and placed into an .sql file. Since we use two databases, we need two .sql files, one to work with MQBankDB and the other to work with MQFeeDB. Figure 38 on page 60 shows in what sequence amqsxag0 executes the MQI and SQL calls.

This program checks also if the two databases are out of sync. The two modules amqsxab0 and amqsxaf0 count the number of database updates. These counters are returned to the main program together with the contents of the database tables. The program ends when the counters do not match.

Since DB2 allows only one active connection to a database, we have to connect explicitly to each database. MQCMIT disconnects implicitly.

For example, the main routine calls the function that connects to MQBankDB with this statement:

```
rc = ConnectToMQBankDB();
```

The routine that does the connect is shown in Figure 39 on page 60.

```
       amqsxag0.c                    amqsxab0.sqc / amqsxaf0.sqc
       ----------                    ---------------------------
           MQCONN
           MQOPEN
                  ----------------->  SQL DECLARE CURSOR for MQBankDB
                  ----------------->  SQL DECLARE CURSOR for MQFeeDB
       begin loop
          MQBEGIN
            MQGET
invalid command ?
               no  (yes = end)
                  ----------------->  SQL CONNECT TO MQBankDB
                  ----------------->  SQL OPEN curBank
                                      SQL FETCH curBank
                  ----------------->  SQL CONNECT TO MQFeeDB
                  ----------------->  SQL OPEN CurFee
                                      SQL FETCH curFee
  DB out of sync?
               no(yes = end)
                  ----------------->  SQL UPDATE MQFeeDB
                  ----------------->  SQL CONNECT TO MQBankDB
                  ----------------->  SQL UPDATE MQBankDB
         commit ?
              yes         no
           MQCMIT         MQBACK
             loop         ...
                          ...
```

Figure 38. SQL Calls to Access Two Databases

:i2/connect

```
int ConnectToMQBankDB(void)
{
  long rc=OK;
  EXEC SQL CONNECT TO MQBankDB;
  CHECKERR ("CONNECT TO MQBankDB");
  return (rc);
}
```

Figure 39. Code to Connect to a Database

2.9.2 Creating the Executable

Depending on the platform and the compiler you use, select one of the command sequences from Table 11:

Table 11. Commands to Create Executable that Accesses Two Databases

Compiler	Command
Microsoft Visual C/C++	msmake amqsxab0 MQBANKDB |more msmake amqsxaf0 MQFEEDB |more msmake2 amqsxag0 amqsxab0.obj amqsxaf0.obj
IBM Visual Age C/C++	ibmmake amqsxab0 MQBANKDB |more ibmmake amqsxaf0 MQFEEDB |more ibmmake2 amqsxag0
CSet++ for AIX	chmod +x amqsxag0.sh amqsxasg.sh (The shell file calls the make file.)

Note: Use xxxmake**2** to compile and link amqsxag0!

You find the make files for Windows NT and AIX in Appendix B, "Example Using Two XA Resources" on page 185 and also on the diskette.

2.9.3 Testing the Program

To test the program under Windows NT, open two Command Prompt windows via the DB2 command line processor to initialize the DB2 environment. Make sure you are in the right directory and use the correct user ID.

In the first window, start DB2, the queue manager, amqsput and then put a valid UPDATE message on the queue:

```
db2start
strmqm
amqsput BANK
UPDATE Balance change=25 WHERE Account=3
```

The first test gives you an error. Start amqsxag0 in the other window and watch what happens:

```
amqsxag0 BANK
Target queue is BANK
Unit of work started
--- error report ---
ERROR occured : CONNECT TO MQFeedDB.
SQLCODE : -1248
SQL1248N  Database "" not defined with the transaction manager. SQLSTATE=42705
--- end error report ---
MQBACK successfully issued
```

The reason for the error is that the queue manager knows only about the database MQBankDB. How do we fix that?

Stop the queue manager and add a second XA resource manager stanza to the qm.ini file as shown below:

Windows NT

```
XAResourceManager:
      Name=DB2 MQBANKDB
      SwitchFile=c:\mqm\bin\db2swit.dll
      XAOpenString=MQBANKDB
      ThreadOfControl=THREAD
XAResourceManager:
      Name=DB2 MQFEEDB
      SwitchFile=c:\mqm\bin\db2swit.dll
      XAOpenString=MQFEEDB
      ThreadOfControl=THREAD
```

AIX

```
XAResourceManager:
      Name=DB2 MQBANKDB
      SwitchFile=/usr/lpp/mqm/bin/db2swit.dll
      XAOpenString=MQBANKDB
XAResourceManager:
      Name=DB2 MQFEEDB
      SwitchFile=/usr/lpp/mqm/bin/db2swit.dll
      XAOpenString=MQFEEDB
```

Now start the queue manager again and then the program. You will see that it works fine. Use the file select.sql in Figure 15 on page 36 to check if both databases are correctly updated.

2.10 Exercise 5: Configuration Issues

This exercise is designed to test the impact of a different queue manager configuration on the behavior of the system.

Let us see what happens when we execute the program amqsxas0 from 2.7, “Exercise 2: Using One XA Resource” on page 40 using the queue manager from the previous exercise. Remember, the qm.ini file contains two XA resource manager stanzas.

Make sure that DB2 and the queue manager are running and that the queue BANK is empty. Then start the program amqsxas0 and watch what happens:

```
amqsxas0 BANK
Target queue is BANK
Unit of work started
--- error report ---
ERROR occurred : OPEN CURSOR.
SQLCODE : -805
SQL0805N Package "ADMIN.AMQSXAS0" was not found.  SQLSTATE=51002
--- end error report ---
MQBACK successfully issued
```

The program did run fine before. However, there are now two external resource managers defined to the queue manager and amqsxas0 does not select one.

If more than one XAResourceManager is defined for one queue manager, then we first have to connect to the correct database before we use it. To do this we add the following lines to our program just before we open the cursor:

```
/******************************************************/
/* Get details from database                          */
/******************************************************/
if (rc == OK) {
   EXEC SQL CONNECT TO MQBankDB;
   CHECKERR ("CONNECT TO MQBankDB");
}
if (rc == OK) {
   EXEC SQL OPEN cur;
   CHECKERR ("OPEN CURSOR");
}
:
```

The example program is the same used in Exercise 2. You can find it on the diskette. The file name is amqsxas2.sqc.

Chapter 3. Message Segmentation

MQSeries Version 5 introduces the concept of message segmentation. Message segmentation allows an application to PUT and GET logical messages that are larger than 4 MB and yet send only physical messages that are 4 MB or smaller across the network. Of course, using MQSeries Version 5 you can also send individual messages that are as large as 100 MB, though you may not want to do that because of resources consumed or because one of the nodes in your MQSeries environment through which your messages must pass cannot deal with anything larger than 4 MB.

First of all, a few definitions are in order:

Physical Message The basic unit that appears in a queue. It can be a segment of a logical message, or it can be a complete message if the message is not segmented.

Logical Message A single unit of application information. This could be a physical message or several message segments.

Message Segment Part of a logical message. In general a message segment is created because the logical message the application wants to send is larger than that which the queue or the queue manager can accommodate.

Prior to MQSeries Version 5 everything was equal. A physical message always was the same as a logical message, and a message was never segmented; therefore each physical message was always a complete logical message.

In this chapter, we provide an introduction to segmenting and explain examples that demonstrate how it works. You can find the source code for the examples in Appendix C, “Message Segmenting Examples” on page 203 and also on the diskette that comes with this book.

To support segmenting, new flags and fields have been added to some structures. For the queue manager to recognize the extensions to the structures you have to set the version number of the structure to 2. The message header (MQMD) is one of them. MQSeries Version 2 queue managers don′t know about the new fields and ignore them. MQSeries Version 5 queue managers ignore them when the version of the structure is set to 1.

3.1 System and Application Segmentation

MQSeries Version 5 provides two kinds of message segmentation:

- System or arbitrary segmentation
- Application segmentation

To control message segmentation several new flags have been introduced and some fields have been added to the message descriptor. We discuss them in the following sections.

3.1.1 Arbitrary Segmentation

The simplest way is to let MQSeries do the segmentation for you. One might use it when a message is not too large for the sender's or receiver's buffers, but perhaps too large for a queue manager or a queue along the route. The sender does not want to segment the message itself in order to take account of the limitations of intervening queues or the queue managers the message passes through.

The queue manager is allowed to segment a message if it is longer than the maximum message length values specified in MAXMSGL in the queue manager or the queue. The default for MAXMSGL is 4 MB. The queue manager splits messages at 16-byte boundaries. That means only the last segment can be smaller than 16 bytes.

Arbitrary segmentation is transparent to the sending and receiving applications.

The changes to the programs that send and receive segmented messages are minimal. The sending application has to inform the queue manager that it can segment the message if necessary, and the receiving application has to tell the queue manager to reassemble the message if it has been segmented.

To allow the queue manager to perform segmenting on its own when the message exceeds MAXMSGL, specify in the message descriptor:

- md.Version = MQMD_VERSION_2;
- md.MsgFlags = MQMF_SEGMENTATION_ALLOWED;

The receiving application can get each message segment individually or have the queue manager reassemble the message and return it to the program after all segments have been received. The queue manager makes sure that the segments are in the correct order. We deal here with a version 2 message, of course. Add one parameter to the get message options:

- gmo.OPTIONS += MQGMO_COMPLETE_MSG;

Note: If not specified segments are returned individually.

3.1.1.1 A Simple Scenario

The following shows what options have to be set for the MQPUT and MQGET call:

```
MQPUT
  PMO     No options set
  MD      MQMF_SEGMENTATION_ALLOWED
          MQMD_VERSION_2
MQGET
  GMO     MQGMO_COMPLETE_MSG
  MD      No options set
```

Comments:

- The MD Version must be set to MQMD_VERSION_2. This is *not* the default. The MD structure is version 2, but the MD Version field is set to MQMD_VERSION_1 by default for compatibility.
- Since the message is reassembled on MQGET by the queue manager, the application buffer must be large enough to hold the reassembled message unless the MQGMO_ACCEPT_TRUNCATED_MSG option is specified.
- Data conversion, if required, can be done only on the reassembled message by the getting application. The message is already reassembled when the exit is called. Data conversion for a segmented message in the sender channel will fail if the data format is such that the conversion exit cannot handle incomplete data.
- MQRC_NO_MESSAGE_AVAILABLE (2033) is returned on an MQGET if none of the segments are available.
- A unit of work is necessary for persistent messages that require arbitrary segmentation. While MQPUTs and/or MQGETs within a unit of work cause no problems, the queue manager handles MQPUTs and/or MQGETs outside a unit of work as follows:
  - If no unit of work exists, the queue manager creates one and commits automatically.
  - The error MQRC_UOW_NOT_AVAILABLE (2055) is returned if no unit of work exists.

3.1.2 Application Segmentation

This type of segmentation can be used when the application:

- Wants to control segment boundaries
- Wants to PUT before message generation is completed
- Cannot handle the message size in its buffers

For example, the sender is creating a very large message and wants to start sending before the generation of the message is complete. This might be because the message is too large to be handled by the applications. The advantage of this over sending multiple messages is that the receiver can GET all the segments in a single operation. Of course, the receiver could also view them as individual segments before the entire logical message has arrived.

Another likely scenario is that the application wants to control the segment boundaries. This might be for any reason including data conversion. An example may be an order form that consists of a header containing information such as date and address, and several items specifying the products to be ordered. The header and each of the items could be sent as segments.

When you create your own segments specify in the:

- Message descriptor:
  - md.Version = MQMD_VERSION_2;
  - md.MsgFlags = MQMF_SEGMENT;
  - md.MsgFlags = MQMF_LAST_SEGMENT;
- Put message options:
  - pmo.Options = MQPMO_LOGICAL_ORDER;

MQPMO_LOGICAL_ORDER indicates that the application issues successive PUTs to put segments into a logical message. The queue manager maintains sequence number and offset.

Note: The last segment only may be zero length.

The receiving queue manager reassembles the logical message on behalf of the getting application. In the receiving program, specify in the get message option:

- mqgmo.OPTIONS += MQGMO_COMPLETE_MSG;

This is the only option that causes the queue manager to reassemble message segments.

All segments must be available before the logical message may be received. The reason code MQRC_NO_MSG_AVAILABLE (2033) is returned on the MQGET if all segments comprising the composite logical message are not available.

Note: For persistent messages, the queue manager can reassemble the segments only within a unit of work. The persistence of all segments must be consistent.

3.1.2.1 A Simple Scenario

In the following scenario, the application splits a logical message into three segments, but it inhibits any further segmentation by the queue manager or by queue managers the segments pass through. The receiving application wants to get the complete logical message with a single MQGET.

```
MQPUT
  PMO     MQPMO_LOGICAL_ORDER
  MD      MQMF_SEGMENT              (all 3 segments)
          MQMF_LAST_SEGMENT         (last  segment )
MQGET
  GMO     MQGMO_COMPLETE_MSG
  MD      No options set
```

Notes:

1. MQMF_SEGMENT is optional for the last message.
2. MQMF_LAST_SEGMENT must be set on the last segment.

Comments:

- The putting application keeps MQPUTting subsequent segments in the correct order, using the same queue handle for each segment.
- The effect of the MQMF_LOGICAL_ORDER flag is that the queue manager generates and maintains a group ID, the offset and the message sequence number in the message descriptor. The putting application should not set any of these fields. The application is only responsible for the MD flags such as MQMF_SEGMENT. The MQMF_LAST_SEGMENT flag terminates the logical message, and a subsequent MQPUT on the same queue handle will result in a new logical message with its own unique group ID.

- The message descriptor returned by an MQPUT contains the actual group ID, offset, etc. used for each segment. These can be saved if necessary for matching incoming segments or error recovery.
- The queue manager automatically generates a single unique group ID which is used for all the segments comprising a single logical message.
- The presence of any of the group/segmentation flags in the message descriptor causes a group ID to be generated. The putting application can only supply its own group ID when MQPMO_LOGICAL_ORDER is reset.
- It is an error to break the logical sequence, or issue an MQCLOSE without putting the last segment.
- To put an unsegmented message while in the middle of a logical sequence, just open another handle to the same queue.
- The Encoding and CCSID of segments should be consistent. Since this example uses MQGMO_COMPLETE_MSG on the MQGET, these must be consistent in this case.
- The MD version does *not* travel with the message. The version of the MD on return from MQGET is dependent on the version of the MD on input of the MQGET. If a Version 1 MD is supplied on the MQGET, the MD2 specific fields are packaged in an MD extension (MDE) and prepended to the data.
- All messages must be available before the logical message may be received. The reason code MQRC_NO_MSG_AVAILABLE (2033) is returned on the MQGET if one or more of the segments are missing.
- If segments of a logical message are found to have different CCSIDs or Encoding, reassembly of the logical message stops at that point, and all segments up to that point are returned with a warning and reason code MQRC_INCONSISTENT_CCSIDS (2243) or ..._ENCODINGS (2244).

3.1.2.2 A Scenario for Getting Individual Segments

The next scenario is different in two aspects:

- MQFM_SEGMENTATION_ALLOWED on the MQPUT allows further arbitrary segmentation if required.
- The getting application gets the individual segments as they are sent.

```
MQPUT
  PMO     MQPMO_LOGICAL_ORDER
  MD      MQMF_SEGMENT                  (1st nd 2nd segment)
          MQMF_LAST_SEGMENT             (3rd (last) segment)
          MQMF_SEGMENTATION_ALLOWED     (all segments)
MQGET
  GMO     MQGMO_LOGICAL_ORDER           (all segments)
          MQGMO_ALL_SEGMENTS_AVAILABLE  (1st segment)
  MD      No options set
```

Comments:

- The getting application keeps getting segments using the same queue handle until the last segment is detected. The last segment is detected by testing either the MD flags or the GMO SegmentStatus output field to ascertain if more segments exist before re-iterating the MQGET.
- The effect of the MQGMO_LOGICAL_ORDER flag is that the queue manager maintains the expected group ID, offset and message sequence number in the message header. The queue is traversed in logical order.
- By specifying MQGMO_ALL_SEGMENTS_AVAILABLE, the first segment will not be returned until all the segments comprising the logical message are available. Segments of incomplete messages will not be returned.
- If the MQGMO_LOGICAL_ORDER flag is set without MQGMO_ALL_SEGMENTS_AVAILABLE, MQGET will iterate through the segments until a gap in the logical sequence is encountered, resulting in a MQRC_NO_MSG_AVAILABLE (2033) being returned, despite the existence of later segments. MQGETs on this queue handle will only proceed when the missing segment becomes available.
- Another application that does not use MQGM_LOGICAL_ORDER could remove individual segments even after MQGMO_ALL_SEGMENTS_AVAILABLE has ascertained their presence, causing a gap in the logical message. MQGMO_BROWSE_* with MQGMO_LOCK can be used to lock all segments of the entire message.
- It is an error, and you will get a warning message, if you break the logical sequence by issuing an MQGET without MQGMO_LOGICAL_ORDER or closing the queue without getting the last segment.

- If MQGMO_LOGICAL_ORDER is not set, the queue is traversed in physical order. Individual segments from different logical messages could be retrieved in any order.
- As the putting application specified MQMF_SEGMENTATION_ALLOWED, any of the segments may have been further segmented by any queue manager. Therefore, there may be more gets than there were puts.
- A getting application can use MQMO_MATCH_GROUPID and/or MQMO_MATCH_OFFSET to more precisely define the expected message. The group ID and offset fields of the MD are used as input fields to be matched by the incoming message.
- The returned MD contains precise segment details, such as group ID, offset and flags. These can be used for error recovery or to forward the message segment with the original segment information.

3.1.3 What about Existing Programs

You can execute pre-version 5 programs and they will work just as they did before. The following scenario shows that segmentation is inhibited by default.

```
MQPUT
  PMO     No options set
  MD      MQMF_SEGMENTATION_INHIBITED
MQGET
  GMO     No options set
  MD      No options set
```

MQMF_SEGMENTATION_INHIBITED is the default. Its value is 0.

3.2 About the Message Segmenting Examples

In this chapter you will find programming examples which demonstrate the methods available to you to accomplish message segmentation. The first of these will demonstrate automatic segmentation where the queue manager determines if and when to segment a message. This is also called *arbitrary segmentation* since the program does nothing to control when or where a message is segmented. In this case you will see that all you really have to do is allow it to occur; that is, the program needs to tell the queue manager that it is OK to segment messages.

The second example will show how to code an application that controls segmentation itself rather than letting the queue manager take care of it automatically. You may want to do this if you want to control where the segmentation occurs, for data conversion for example.

```
/******************************************************************/
/*  MQMD Structure -- Message Descriptor                          */
/******************************************************************/

typedef struct tagMQMD {
   MQCHAR4   StrucId;          /* Structure identifier           */
   MQLONG    Version;          /* Structure version number       */
   MQLONG    Report;           /* Report options                 */
   MQLONG    MsgType;          /* Message type                   */
   MQLONG    Expiry;           /* Expiry time                    */
   MQLONG    Feedback;         /* Feedback or reason code        */
   MQLONG    Encoding;         /* Data encoding                  */
   MQLONG    CodedCharSetId;   /* Coded character set identifier */
   MQCHAR8   Format;           /* Format name                    */
   MQLONG    Priority;         /* Message priority               */
   MQLONG    Persistence;      /* Message persistence            */
   MQBYTE24  MsgId;            /* Message identifier             */
   MQBYTE24  CorrelId;         /* Correlation identifier         */
   MQLONG    BackoutCount;     /* Backout counter                */
   MQCHAR48  ReplyToQ;         /* Name of reply-to queue         */
   MQCHAR48  ReplyToQMgr;      /* Name of reply queue manager    */
   MQCHAR12  UserIdentifier;   /* User identifier                */
   MQBYTE32  AccountingToken;  /* Accounting token               */
   MQCHAR32  ApplIdentityData; /* Application data relating to
                                  identity                       */
   MQLONG    PutApplType;      /* Type of application that puts
                                  the message                    */
   MQCHAR28  PutApplName;      /* Name of application that puts
                                  the message                    */
   MQCHAR8   PutDate;          /* Date when message was put      */
   MQCHAR8   PutTime;          /* Time when message was put      */
   MQCHAR4   ApplOriginData;   /* Application data relating to
                                  origin                         */
   MQBYTE24  GroupId;          /* Group identifier               */
   MQLONG    MsgSeqNumber;     /* Sequence number of logical
                                  message within a group         */
   MQLONG    Offset;           /* Offset of data in physical msg
                                  from start of logical message  */
   MQLONG    MsgFlags;         /* Message flags                  */
   MQLONG    OriginalLength;   /* Length of original message     */
  } MQMD;
typedef MQMD MQPOINTER PMQMD;
```

Figure 40. New Fields in the Message Header

It is also possible to accomplish both automatic segmentation and application determined segmentation within the same program. This will not be demonstrated in this book.

All of the programming examples are modified versions of programs which are supplied in the distribution of MQSeries Version 5. The unmodified programs (which you can find in your ″samples″ directory) are:

amqsput0.c Sample C program that puts messages to a message queue

amqsget0.c Sample C program that gets messages from a message queue

amqsbcg0.c Sample C program to read and output both the message descriptor fields and the message content of all the messages on a queue

Segmentation uses fields that have been added to the end of the MQMD structure. Figure 40 on page 73 shows the new fields in bold. We refer to them in the next exercises.

3.3 A Program to Create a Very Large File

The first task to be accomplished for the exercises to come is to create a file that will become our message. We create this large message with the program *big.c* shown in Figure 41. It creates a file that is 4950 bytes long.

```
#include <stdio.h>
#include <string.h>

int main(argc, argv)
        int argc;
        char *argv[];
{
int i = 0;
for (i=1; i < 100; i++)
 printf(″THISISAVERYLARGEFILETOTESTARBITRARYSEGMENTATION!!!″);
return(0);
}
```

Figure 41. Program that Creates a Very Large File

To compile the program use one of the commands in Table 12 on page 75.

Table 12. Commands to Compile BIG.C	
Compiler	**Command**
Microsoft Visual C/C++	cl big.c
IBM Visual Age C/C++	icc big.c
CSet++ for AIX	xlc big.c -o big

To create the file, simply execute ″big″ piping the output to a flat file called ″very_large_file″.

```
big > very_large_file
```

3.4 Exercise 6: Arbitrary Segmentation

The purpose of this exercise is to demonstrate how the new MQSeries Version 5 queue managers can:

- Take a message which is too big
  (either for the queue manager or for the specific queue)
- And automatically segment it

MQSeries breaks the message into smaller messages (called segments) and sends them wherever the ″too large″ message would have been sent.

The local queue manager at the receiving end puts the pieces back together and is able to present the ″too large″ message to the GETting application as though it had never been segmented.

Optionally, the receiving application can also view the segments as separate messages. When using arbitrary (system-generated) segmentation. However, it is more common to let MQSeries worry about the details and for the receiving application to view the message as a whole.

3.4.1 Writing a Program for Arbitrary Segmentation

We modify amqsput0.c and create the program PUT_SEG1. The sample amqsput0 puts messages on a message queue, and is an example of the use of MQPUT. PUT_SEG1 allows for the possibility of messages which exceed those allowed on the queue or queue manager. Messages are sent to the queue named by a parameter. You may also specify a queue manager name. Both programs get their input from StdIn.

A complete listing of PUT_SEG1 is in C.1, “PUT_SEG1 Performing Arbitrary Segmenting” on page 203. The changes are marked in bold. The necessary modifications to allow arbitrary segmentation are summarized below:

- First we have to allow the program to handle larger messages. So we change the buffer size from 100 to 5000 bytes.

```
char     buffer[5000];           /* our large message buffer      */
```

- Next, we have to allow the queue manager to perform segmentation. For segmentation, it uses version 2 of the API functions. So we change the message header version to 2. We also need to set a flag that allows the queue manager to segment. Put the following statements somewhere between the MQOPEN and the MQPUT.

```
md.MsgFlags = MQMF_SEGMENTATION_ALLOWED;
md.Version  = MQMD_VERSION_2 ;
```

These are all the changes required for arbitrary segmenting. The MQPUT does not change. However, let us mention that there are new values for the Options field of the Put Message Options:

- MQPMO_NEW_MSG_ID
- MQPMO_NEW_CORREL_ID

These options relieve the need for the memcpy statements below that cause MQ to reset MsgId and CorrelId and generate new ones.

```
memcpy(md.MsgId, MQMI_NONE, sizeof(md.MsgId) );
memcpy(md.CorrelId, MQCI_NONE, sizeof(md.CorrelId) );
```

3.4.2 Writing a Program that Reads Logical Messages

We modify the program amqsbcg0.c and create GET_SEG1.C. The program listing is in C.2, “BCG_SEG1 Browsing only Logical Messages” on page 207. The sample prints out both message descriptor and message text. The version of amqsbcg supplied with MQSeries Version 5 recognizes the new fields in the MDMD structure. It was changed from previous releases to allow the possibility of segmented messages. Specifically the new fields in the header were added and formatted and the API version was specified:

```
MsgDesc.Version = MQMD_VERSION_2;
```

For our purposes we wanted to process only complete messages, not the individual segments. Therefore we needed the following addition:

```
GetMsgOpts.Options += MQGMO_COMPLETE_MSG;
```

Remember, in the PUT program we also had to change the buffer size to 5000 to allow for a single larger message. AMQSBCG0.C already handled messages up to 32767 bytes long. If a longer message is read it will fail with the reason ″truncated-msg″.

The modifications to read only complete messages are summarized below:

1. Use a version 2 MQMD in case the message is segmented or grouped.

   ```
   MsgDesc.Version = MQMD_VERSION_2 ;
   ```

2. The function printMD prints the new fields in the MQMD structure with the following instructions:

   ```
   printf("\n  GroupId : X'");

   for (i = 0 ; i < MQ_GROUP_ID_LENGTH ; i++)
     printf("%02X",MDin->GroupId[i] );

   printf("'");
   printf("\n  MsgSeqNumber   : '%d'", MDin->MsgSeqNumber);
   printf("\n  Offset         : '%d'", MDin->Offset);
   printf("\n  MsgFlags       : '%d'", MDin->MsgFlags);
   printf("\n  OriginalLength : '%d'", MDin->OriginalLength);
   ```

   Note: MDin is the pointer to the message descriptor structure.

   We will analyze these fields when we test our application.

3. To get the system to reassemble the message we set the get message options as follows:

   ```
   GetMsgOpts.Options += MQGMO_COMPLETE_MSG ;
   ```

Note: The new field MatchOptions in the Get Message Options relieves the need for resetting the MessageID and CorrelID each time.

3.4.3 Compiling the Programs

Compile the programs using one of the compilers in Table 13.

Table 13. Commands to Compile Programs for Arbitrary Segmentation

Compiler	Command
Microsoft Visual C/C++	cl put_seg1.c mqm.lib cl bcg_seg1.c mqm.lib
IBM Visual Age C/C++	icc put_seg1.c mqm.lib icc bcg_seg1.c mqm.lib
CSet++ for AIX	xlc put_seg1.c -l mqm -o put_seg1 xlc bcg_seg1.c -l mqm -o bcg_seg1

3.4.4 Creating a Queue

The next thing that needs to be done to set up the environment is to create a queue in which to put some messages. To do this run the *runmqsc* command and enter the following commands:

```
runmqsc
define qlocal(SEGTEST1) like(SYSTEM.DEFAULT.LOCAL.QUEUE) MAXMSGL(500)
end
```

This new queue called SEGTEST1, is limited to messages of 500 bytes in length. This is how we force the queue manager to do segmentation for us. If you look at the size of ″very_large_file″ that we created in 3.3, “A Program to Create a Very Large File” on page 74, you will see that it is considerably larger than 500 bytes.

3.4.5 Testing Arbitrary Segmentation

If this is not the first time you have been through the execution of these programs use the standard sample GET program to clear the queue:

```
amqsget SEGTEST1 [QMgrName]
```

Note: The queue manager name is optional; if you are connecting to the default queue manager you do not need it.

Now use the new put program to put a large message in the queue SEGTEST1. Make sure that the very large file exists. You should have created it with the program ″big″ described in 3.3, “A Program to Create a Very Large File” on page 74.

```
put_seg1 SEGTEST1 [QMgrName] < very_large_file
```

The message was too large for our definition of the queue SEGTEST1 (500 bytes) and should have generated 10 physical messages on the queue. These are the segments. You can prove that with runmqsc:

```
runmqsc
dis ql(SEGTEST1) curdepth
     1 : dis ql(SEGTEST1) curdepth
AMQ8409: Display Queue details.
   QUEUE(SEGTEST1)                              CURDEPTH(10)
end
```

The standard browse sample program, which does not ask the queue manager to put all of the segments into a single logical message, displays the ten segments. Figure 42 on page 79 shows the first of them. We are especially interested in the new fields added to the MQMD structure printed in bold.

The messages 2 through 9 are identical with the exception of the offset field in the MQMD structure. Table 14 on page 80 shows those version 2 fields for all ten segments. The first nine messages are 496 bytes long and the tenth 486 which gives us a total of 4950.

```
 MQGET of message number 1
****Message descriptor****

  StrucId  : 'MD  '  Version : 2
  Report   : 0  MsgType : 8
  Expiry   : -1  Feedback : 0
  Encoding : 546  CodedCharSetId : 437
  Format : 'MQSTR   '
  Priority : 0  Persistence : 0
  MsgId : X'414D512042474147452020202020202020004F000034E2BA5B'
  CorrelId : X'000000000000000000000000000000000000000000000000'
  BackoutCount : 0
  ReplyToQ       : '                                                '
  ReplyToQMgr    : 'QMGR1                                           '
  ** Identity Context
  UserIdentifier : 'OS2         '
  AccountingToken :
  X'0131000000000000000000000000000000000000000000000000000000000000'
  ApplIdentityData : '                                '
  ** Origin Context
  PutApplType    : '4'
  PutApplName    : 'D:\REDBOOK\PUT_SEG1.EXE     '
  PutDate  : '19980212'    PutTime  : '09011538'
  ApplOriginData : '    '

  GroupId : X'414D512042474147452020202020202020004F010034E2BA5B'
  MsgSeqNumber   : '1'
  Offset         : '0'
  MsgFlags       : '3'
  OriginalLength : '496'

****   Message      ****

 length - 496 bytes

00000000:  5448 4953 4953 4156 4552 594C 4152 4745 'THISISAVERYLARGE'
00000010:  4649 4C45 544F 5445 5354 4152 4249 5452 'FILETOTESTARBITR'
00000020:  4152 5953 4547 4D45 4E54 4154 494F 4E21 'ARYSEGMENTATION!'
:
:
000001C0:  2121 5448 4953 4953 4156 4552 594C 4152 '!!THISISAVERYLAR'
000001D0:  4745 4649 4C45 544F 5445 5354 4152 4249 'GEFILETOTESTARBI'
000001E0:  5452 4152 5953 4547 4D45 4E54 4154 494F 'TRARYSEGMENTATIO'
```

Figure 42. A Message Segment (Arbitrary Segmentation)

Now let's look at the new fields in the message descriptor.

Table 14. New Fields in Message Descriptor										
Field	**Seg1**	**Seg2**	**Seg3**	**Seg4**	**Seg5**	**Seg6**	**Seg7**	**Seg8**	**Seg9**	**Seg10**
MsgSeqNumber	1	1	1	1	1	1	1	1	1	1
Offset	0	496	920	1488	1984	2480	2976	3472	3968	4464
MsgFlags	3	3	3	3	3	3	3	3	3	7
OriginalLength	496	496	496	496	486	496	496	496	496	486

The fields contain the following information:

- MsgSeqNumber

  This consistently has the value ″1″. Why? Because this field is not used for segmentation; rather it is used for Message Groups, which is discussed in another chapter.

- Offset

  This determines the position in the logical message that this segment occupies.

- MsgFlags

  For the first segment this has a value of ″3″. Looking at the C header file, that is equal to the sum of:

  MQMF_SEGMENTATION_ALLOWED (value of ″1″) and
  MQMF_SEGMENT (value of ″2″).

  This is consistent for all of the segments except the final one which has a value of ″7″. This is the sum of the previous values plus:

  MQMF_LAST_SEGMENT (value of ″4″).

- Original Length

  You might think this was pertinent here, but looking in the *MQSeries Application Programming Reference*, SC33-1673 this field is only relevant for report messages and in this case we are looking at the original length of the segment, not the logical message.

3.4.6 Putting Segments Back Together

Finally, to show you how MQSeries Version 5 puts these segments back together run the modified version of amqsbcg.

Looking at the same fields as in Table 14 there are some differences to note in Figure 43 on page 81.

- OriginalLength

```
 MQGET of message number 1
****Message descriptor****

  StrucId  : 'MD  '   Version : 2
  Report   : 0  MsgType : 8
  Expiry   : -1  Feedback : 0
  Encoding : 546  CodedCharSetId : 437
  Format : 'MQSTR   '
  Priority : 0  Persistence : 0
  MsgId : X'414D512042474147452020202020202020004F000034E2BA5B'
  CorrelId : X'000000000000000000000000000000000000000000000000'
  BackoutCount : 0
  ReplyToQ       : '                                                '
  ReplyToQMgr    : 'QMGR1                                           '
  ** Identity Context
  UserIdentifier : 'OS2         '
  AccountingToken :
  X'0131000000000000000000000000000000000000000000000000000000000000'
  ApplIdentityData : '                                '
  ** Origin Context
  PutApplType    : '4'
  PutApplName    : 'D:\REDBOOK\PUT_SEG1.EXE     '
  PutDate  : '19980212'    PutTime  : '09011538'
  ApplOriginData : '    '

  GroupId : X'414D512042474147452020202020202020004F010034E2BA5B'
  MsgSeqNumber   : '1'
  Offset         : '0'
  MsgFlags       : '7'
  OriginalLength : '4950'

****   Message      ****

 length - 4950 bytes

00000000:  5448 4953 4953 4156 4552 594C 4152 4745 'THISISAVERYLARGE'
00000010:  4649 4C45 544F 5445 5354 4152 4249 5452 'FILETOTESTARBITR'
00000020:  4152 5953 4547 4D45 4E54 4154 494F 4E21 'ARYSEGMENTATION!'
:
:
00001330:  4152 4745 4649 4C45 544F 5445 5354 4152 'ARGEFILETOTESTAR'
00001340:  4249 5452 4152 5953 4547 4D45 4E54 4154 'BITRARYSEGMENTAT'
00001350:  494F 4E21 2121                          'ION!!!          '
```

Figure 43. A Reassembled Logical Message (Arbitrary Segmentation)

This now has a value of 4950, the size of the logical message. In the dump of the message all ten segments have been presented to the program as a single message 4950 bytes long.

- MsgFlags

  This now has the value 7, which says that segmentation is allowed, that this message is a segment, and that it is the last segment. In other words, this is a complete logical message.

- Offset

  This now shows a value of ″0″. This is consistent with the first of our segments and makes sense if this one segment makes up the entire message.

3.5 Exercise 7: Application Segmentation

The programmer can control the segmentation of messages within his application. The local queue manager at the receiving end puts the pieces back together and is able to present the segmented message to the GETting application as though it had never been segmented.

The receiving application can also view the segments as separate messages. When using application segmentation it is common to want to view the segments at the receiving end BOTH as a complete message AND as individual segments. To see as a complete message, we use bcg_seg1 from Exercise 6. To see the segments individually, we use amqsbcg which is one of the MQSeries samples.

3.5.1 Writing a Program for Application Segmentation

We take amqsput0.c, modify it and create PUT_SEG2.C. The modifications cause the message to be segmented on application defined boundaries.

There are more changes here than in the previous example which used automatic segmentation. To summarize them briefly:

1. We change the buffer size so that the buffer can hold our ″very large message″ from 100 to 5000.

   ```
   char     buffer[1000];           /* message buffer */
   ```

2. Since segmenting uses the new fields in the MQMD structure we have to tell the queue manager to use the new MQMD version.

   ```
   md.Version = MQMD_VERSION_2 ;
   ```

3. In the message flags we have to indicate that the physical message is a segment.

```
md.MsgFlags = MQMF_SEGMENT ;
```

4. The following statement says to retrieve the messages in logical order.

   ```
   pmo.Options = MQPMO_LOGICAL_ORDER ;
   ```

   Without this option segments are retrieved in physical order.

5. The program splits the large message into five segments. At the end it sends one extra segment with the flag:

   ```
   md.MsgFlags = MQMF_LAST_SEGMENT ;
   ```

In short, we use Version 2 functions to create segments. We want to maintain logical order and we put one extra segment in the queue which we mark as the last segment.

For the complete code refer to C.3, “PUT_SEG2 Performing Application Segmenting” on page 217.

Compile the program with one of the following commands:

Table 15. Commands to Compile put_seg2

Compiler	Command
Microsoft Visual C/C++	cl put_seg2.c mqm.lib
IBM Visual Age C/C++	icc put_seg2.c mqm.lib
CSet++ for AIX	xlc put_seg2.c -l mqm -o put_seg2

3.5.2 Creating a Queue

The program PUT_SEG2 uses the same ′very_large_file′ we created earlier. Refer to 3.3, “A Program to Create a Very Large File” on page 74. Next, we have to create a queue in which to put some messages. To do this run the ″runmqsc″ command and enter the following commands:

```
runmqsc
define qlocal(SEGTEST2)
end
```

The new queue is called SEGTEST2. It has the default maximum message size of 4 MB. Our ″very_large_file″ would fit in a single message, though we will break it up into 1000-byte segments.

3.5.3 Testing Application Segmentation

You are now ready to test this new program. If you have run the PUT program before be sure to clear the queue first:

```
amqsget SEGTEST2 [QMgrName]
```

Note: The queue manager name is optional.

Now run our new PUT_SEG2 program to create the segments:

```
put_seg2 SEGTEST2 < very_large_file
Sample PUT_SEG2 start
target queue is SEGTEST2
Sample PUT_SEG2 end
```

The message should have been segmented by our application. We expect five segments plus one extra one. You can prove that with runmqsc:

```
runmqsc
dis ql(SEGTEST2) curdepth
     1 : dis ql(SEGTEST2) curdepth
AMQ8409: Display Queue details.
   QUEUE(SEGTEST2)                              CURDEPTH(6)
end
```

We can also verify this by running amqsgbr, the standard sample browse program that comes with MQSeries.

```
amqsgbr  SEGTEST2
Sample AMQSGBR0 (browse) start

Messages for SEGTEST2
1 <THISISAVERYLARGEFILETOTESTARBITRARYSEGMENTATION!!!>
  --- truncated
2 <!THISISAVERYLARGEFILETOTESTARBITRARYSEGMENTATION!!>
  --- truncated
3 <!!THISISAVERYLARGEFILETOTESTARBITRARYSEGMENTATION!>
  --- truncated
4 <!!!THISISAVERYLARGEFILETOTESTARBITRARYSEGMENTATION>
  --- truncated
5 <N!!!THISISAVERYLARGEFILETOTESTARBITRARYSEGMENTATIO>
  --- truncated
6 <>
no more messages
Sample AMQSGBR0 (browse) end
```

The message length of the first four segments is 999 bytes while the fifth is 954 and the last 0, which comes to a total of 4950.

You can see in Table 16 on page 86 that the fields in the message descriptor relating to segmenting are different from the previous example featuring arbitrary segmentation. All other fields in the six headers are the same.

```
 MQGET of message number 1
****Message descriptor****

  StrucId  : 'MD  '  Version : 2
  Report   : 0  MsgType : 8
  Expiry   : -1  Feedback : 0
  Encoding : 546  CodedCharSetId : 437
  Format : 'MQSTR   '
  Priority : 0  Persistence : 0
  MsgId : X'414D512042474147452020202020202000072000034F1ACAD'
  CorrelId : X'000000000000000000000000000000000000000000000000'
  BackoutCount : 0
  ReplyToQ       : '                                                '
  ReplyToQMgr    : 'QMGR1                                           '
  ** Identity Context
  UserIdentifier : 'OS2         '
  AccountingToken :
  X'0131000000000000000000000000000000000000000000000000000000000000'
  ApplIdentityData : '                                '
  ** Origin Context
  PutApplType    : '4'
  PutApplName    : 'D:\REDBOOK\PUT_SEG2.EXE     '
  PutDate  : '19980223'    PutTime  : '17065299'
  ApplOriginData : '    '

  GroupId : X'414D512042474147452020202020202000072010034F1ACAD'
  MsgSeqNumber   : '1'
  Offset         : '0'
  MsgFlags       : '2'
  OriginalLength : '999'

****   Message      ****

 length - 999 bytes

00000000:  5448 4953 4953 4156 4552 594C 4152 4745 'THISISAVERYLARGE'
00000010:  4649 4C45 544F 5445 5354 4152 4249 5452 'FILETOTESTARBITR'
00000020:  4152 5953 4547 4D45 4E54 4154 494F 4E21 'ARYSEGMENTATION!'
:
:
000003C0:  594C 4152 4745 4649 4C45 544F 5445 5354 'YLARGEFILETOTEST'
000003D0:  4152 4249 5452 4152 5953 4547 4D45 4E54 'ARBITRARYSEGMENT'
000003E0:  4154 494F 4E21 21                       'ATION!!         '
```

Figure 44. A Message Segment (Application Segmentation)

Table 16. New Fields in Message Descriptor						
Field	**Seg1**	**Seg2**	**Seg3**	**Seg4**	**Seg5**	**Seg6**
MsgSeqNumber	1	1	1	1	1	1
Offset	0	999	1998	2997	3996	4950
MsgFlags	2	2	2	2	2	6
OriginalLength	999	999	999	999	954	0

As you can see, there are six messages on the queue including our last (empty) message. To see the details of the messages execute amqsbcg:

amqsbcg SEGTEST2 |more

Figure 44 on page 85 shows the first segment in the queue.

The fields contain the following information:

- MsgSeqNumber

  The sequence number remains the same for all messages. Why? It is because sequence number is used when using message groups. You can see examples of this in the next chapter.

- Offset

  Offset reflects the beginning of a segmented message relative to its position in the logical message. Our first segment above has an offset of 0 and a length of 999. Segment 2 has an offset of 999; therefore segment 2 starts where segment 1 ends, and so on.

- MsgFlags

  The flags are always 2 except the last which is 6. The C language header file cmqc.h tells you that:

  MQMF_SEGMENT is defined as 0x00000002L and

  MQMF_LAST_SEGMENT is defined as 0x00000004L

  The last segment bears both flags which add up to 6.

- OriginalLength

  This field is used for report messages and refers to the length of the original physical message, not the logical message.

3.5.4 Putting Segments Back Together

Finally, to show how MQSeries pastes these segments back together again we run our browse program put_seg2:

put_seg2 SEGTEST2

The reassembled message is shown in Figure 45 on page 88. Let us look at the fields related to segmenting:

- Offset

  The offset is 0. Since the program only sees the logical message there is only one offset.

- MsgFlags

  This is equal to 6. In other words, even though this message is the first segment we see it is also the last or the complete message.

  Note: With arbitrary segmentation, MsgFlags contained 7. The additional bit for ″segmentation is allowed″ is not set for application segmentation.

- OriginalLength

  This now reflects what the application saw as the complete message or 4950.

```
 MQGET of message number 1
****Message descriptor****

  StrucId  : 'MD  '  Version : 2
  Report   : 0  MsgType : 8
  Expiry   : -1  Feedback : 0
  Encoding : 546  CodedCharSetId : 437
  Format : 'MQSTR   '
  Priority : 0  Persistence : 0
  MsgId : X'414D5120514D475231202020202020C4D2FF3413300100'
  CorrelId : X'000000000000000000000000000000000000000000000000'
  BackoutCount : 0
  ReplyToQ       : '                                                '
  ReplyToQMgr    : 'QMGR1                                           '
  ** Identity Context
  UserIdentifier : 'bgage       '
  AccountingToken :
  X'0131000000000000000000000000000000000000000000000000000000000000'
  ApplIdentityData : '                                '
  ** Origin Context
  PutApplType    : '11'
  PutApplName    : 'C:\redbook\PUT_SEG2.EXE     '
  PutDate  : '19980306'    PutTime  : '18361497'
  ApplOriginData : '    '

  GroupId : X'414D5120514D475231202020202020C4D2FF3423300100'
  MsgSeqNumber   : '1'
  Offset         : '0'
  MsgFlags       : '6'
  OriginalLength : '4950'

****   Message      ****

 length - 4950 bytes

00000000:  5448 4953 4953 4156 4552 594C 4152 4745 'THISISAVERYLARGE'
00000010:  4649 4C45 544F 5445 5354 4152 4249 5452 'FILETOTESTARBITR'
00000020:  4152 5953 4547 4D45 4E54 4154 494F 4E21 'ARYSEGMENTATION!'
:
:
00001330:  4152 4745 4649 4C45 544F 5445 5354 4152 'ARGEFILETOTESTAR'
00001340:  4249 5452 4152 5953 4547 4D45 4E54 4154 'BITRARYSEGMENTAT'
00001350:  494F 4E21 2121                          'ION!!!          '
```

Figure 45. A Reassembled Logical Message (Application Segmentation)

Chapter 4. Message Groups

Message groups are new to MQSeries Version 5. In the chapter discussing message segmentation you saw examples of creating logical messages which were made up of one or more physical segments. Message groups are really collections of one or more logical messages. You can think of a message group as some application-oriented collection of messages. For example, you might want to process all of the items in an order as a group.

Processing messages in a group gives you the ability to easily process messages in logical order rather than physical order. It also allows you to only start processing the group when all of the messages have been received.

It is possible for a message that is segmented to be part of a group of messages.

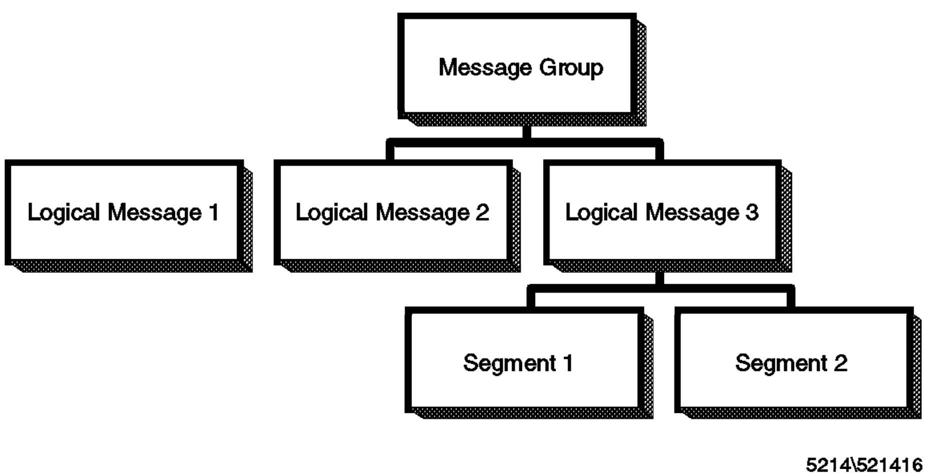


Figure 46. A Message Group

In Figure 46 you see a message group made up of logical messages 1, 2 and 3. Messages 1 and 2 are complete messages which are physical messages on the queue. Message 3, on the other hand, is a logical message made up of segments 1 and 2. Segments 1 and 2 are the physical messages which make up logical message 3 of the message group.

It is possible to put messages under groups and *at the same time* allow application segmentation of messages and *at the same time* allow further arbitrary (system) segmentation.

Messages in a group will all have the same Group ID which can be controlled by the application or generated by the queue manager. The logical sequence of messages in a group is maintained using the message sequence number field in the message header.

Optionally, the receiving application can take a physical view and see all the messages regardless of group, and the segments regardless of message. On the other hand, the receiver can take a logical view and see the segments only as reassembled messages and view the individual messages as part of a group.

Note: Persistence of all messages within a group must be consistent.

4.1 A Simple Grouped Message Scenario

This scenario summarizes what has to be done to retrieve messages within a group in the same order as sent. MQGMO_ALL_MSGS_AVAILABLE can be used to ensure that all messages in the group are available before any are retrieved. The group consists of three messages. The following shows what has to be specified in the put and get statements.

```
MQPUT
  PMO     MQPMO_LOGICAL_ORDER
  MD      MQMD_VERSION_2
          MQMF_MSG_IN_GROUP              1st & 2nd  message
          MQMF_LAST_MSG_IN_GROUP         3rd (last) message
          MQMF_SEGMENTATION_ALLOWED      all messages
MQGET
  GMO     MQGMO_COMPLETE_MSG             all messages    (if segmentation
          MQGMO_LOGICAL_ORDER            all messages     is allowed)
          MQGMO_ALL_MSGS_AVAILABLE       1st message
  MD      No options set
```

Comments:

- This scenario is similar to the one described in 3.1.2.1, “A Simple Scenario” on page 69, substituting ″message within group″ for segment.
- The rules regarding messages within groups are very similar to segments within logical messages, except:
  - The MQMF_*_MSG_IN_GROUP replaces MQMF_*_SEGMENT flags.
  - The message sequence number field in the message descriptor replaces the offset for segments. The message sequence number is incremented by 1 while the offset increments by the segment size.

- MQGMO_ALL_MSGS_AVAILABLE replaces MQGMO_ALL_SEGMENTS_AVAILABLE.
- There is no equivalent to MQGMO_COMPLETE_MSG. Messages within a group are still individual messages.

- The logical sequence when traversing a queue with MQGMO_LOGICAL_ORDER set follows the physical sequence of first messages in a group or ungrouped messages, then iterates through the messages in the group in order of the message sequence number.
- Another application getting the queue without MQGMO_LOGICAL_ORDER could remove individual messages even after MQGMO_ALL_MSGS_AVAILABLE has ascertained their presence, causing a gap in the logical sequence. The browse lock applies only to multiple segments within a single logical message. There is no equivalent for grouped messages.
- If the putting application specified MQMF_SEGMENTATION_ALLOWED any of the messages may have been segmented by any queue manager. MQGMO_COMPLETE_MSG will ensure that these are reassembled by the receiving queue manager.

4.2 A Scenario for Grouped Segmented Messages

For this scenario, we assume three messages with three segments each.

```
MQPUT
  PMO     MQPMO_LOGICAL_ORDER
  MD      MQMD_VERSION_2
          MQMF_MSG_IN_GROUP + MQMF_SEGMENT               Msg 1, Segm 1 & 2
          MQMF_MSG_IN_GROUP + MQMF_LAST_SEGMENT          Msg 1, Segm 3

          MQMF_MSG_IN_GROUP + MQMF_SEGMENT               Msg 2, Segm 1 & 2
          MQMF_MSG_IN_GROUP + MQMF_LAST_SEGMENT          Msg 2, Segm 3

          MQMF_LAST_MSG_IN_GROUP + MQMF_SEGMENT          Msg 3, Segm 1 & 2
          MQMF_LAST_MSG_IN_GROUP + MQMF_LAST_SEGMENT     Msg 3, Segm 3
MQGET
  GMO     MQGMO_LOGICAL_ORDER              all messages and segments
          MQGMO_ALL_MSGS_AVAILABLE         Msg 1, Segm 1
  MD      No options set
```

Comments:

- The hierarchy is segments with grouped messages.

- MQMF_LAST_MSG_IN_GROUP must be set for all segments of the last message.
- MQGMO_ALL_MSGS_AVAILABLE implies MQGMO_ALL_SEGMENTS_AVAILABLE.
- UOW and CCSIDs should be consistent within a logical message, but may vary between messages within the group.

4.3 About the Message Grouping Example

In this chapter you will find a programming example written in C which demonstrates what needs to be done to send and receive messages in groups. All of the programming examples are modified versions of programs which are supplied in the distribution of MQSeries Version 5. You can find the unmodified programs in your ″samples″ directory. The programs to demonstrate message grouping are:

PUT_SEG1.C This program is a modification of the standard PUT sample program amqsput0. It builds a group of logical messages when it puts them on a queue.

BCG_SEG1.C This program is a modification of amqsbcg0 which browses and prints the headers and details of messages.

A complete listing of the programs is in Appendix D, "Message Grouping Examples" on page 223. In this exercise, we also demonstrate:

- What happens when we try to read messages of an incomplete group.
- How to read messages that belong to a group regardless of whether all messages have arrived or not.

4.4 Exercise 8: Putting Message Groups

The purpose of this exercise is to demonstrate how the new MQSeries Version 5 queue manager puts logical messages in a group and when and how they are gotten by the receiving application.

4.4.1 Writing a Program that Puts Messages in a Group

The program PUT_GRP1 will be a version of amqsput0.c with the necessary modifications made to it to cause it to put each entered line as one message in a group. The final null entry will cause the end of group message to be sent.

Note: The input buffer is 100 bytes long.

If you look through the program in D.1, “Source of PUT_GRP1” on page 223 and compare it to the standard amqsput0.c sample you will see four changes. The additional statements set flags to tell the queue manager that the program wants to send a group of messages. The parameters are:

- MQMD_VERSION_2
- MQMF_MSG_IN_GROUP
- MQPMO_LOGICAL_ORDER
- MQMF_LAST_MSG_IN_GROUP

Since message grouping uses the additional fields in the message header, we have to tell the queue manager to use Version 2 of the MD instead of Version 1 which is the default. We must use Version 2 of the API in order to use message groups.

```
md.Version  = MQMD_VERSION_2;
```

For each message in a group you must set a MsgFlag to say that the message is in a group and that you want to maintain a logical sequence. Finally, you must finish the group by setting another MsgFlag.

To tell the queue manager that the messages that follow are part of a group we set the following flag in the message header:

```
md.MsgFlags = MQMF_MSG_IN_GROUP;
```

To tell the queue manager that the messages that follow are to be kept in sequence within the group we set a flag in the put message options:

```
pmo.Options = MQPMO_LOGICAL_ORDER;
```

To put all messages but the last in the group we use the standard MQPUT as shown below. Before the put we reset the message ID and the correlation ID to get a new one.

```
memcpy(md.MsgId, MQMI_NONE, sizeof(md.MsgId) );
memcpy(md.CorrelId, MQCI_NONE, sizeof(md.CorrelId) );

MQPUT(Hcon, Hobj, &md, &pmo, buflen, buffer, &CompCode, &Reason);
if (Reason != MQRC_NONE)              /* report reason, if any */
   printf("MQPUT ended with reason code %ld\n", Reason);
```

For the last message, we have to set a flag that tells the queue manager that it is putting the last message in a group.

Note: In this example, the last message is empty.

The code for the MQPUT of the last message follows:

```
md.MsgFlags = MQMF_LAST_MSG_IN_GROUP;
memcpy (md.MsgId, MQMI_NONE, sizeof(md.MsgId) );
memcpy (md.CorrelId, MQCI_NONE, sizeof(md.CorrelId) );

MQPUT (Hcon, Hobj, &md, &pmo, 0, buffer, &CompCode, &Reason);

if (Reason != MQRC_NONE)     /* report reason, if any */
   printf("MQPUT ended with reason code %ld\n", Reason);
```

4.4.2 Writing a Program that Gets Messages of a Group

The program BCG_GRP1 will be a version of amqsbcg0.c with the necessary modifications made to it to ensure no message is retrieved until all of the messages in a group are present in the queue.

AMQSBCG0.C was changed in MQSeries Version 5 from previous releases to allow for the possibility of segmented and/or grouped messages. Specifically the new fields in the header were added and formatted and the API version was specified:

```
MsgDesc.Version = MQMD_VERSION_2;
```

For our purposes we wanted to process messages only after all of the messages in a group have arrived on the queue. Additionally, we want to process the messages in the order they have been put. Therefore, we need to set the following flags in the get message options:

- MQGMO_LOGICAL_ORDER
- MQGMO_ALL_MSGS_AVAILABLE

The first option tells the queue manager that we want to GET the messages in the same order that they were PUT, regardless of physical position within the queue. The second two statements tell the queue manager that we only want to get a message if all of the messages have been received in the queue. We only want to use this code on the first message since that is the only time that all messages will be available.

The code fragment in Figure 47 on page 95 shows how the get program is written.

```
MsgDesc.Version = MQMD_VERSION_2;            /* Version 2 of MD   */
GetMsgOpts.Version = MQGMO_VERSION_2;        /* Version 2 of GMO  */

GetMsgOpts.Options = MQGMO_NO_WAIT;
GetMsgOpts.Options += MQGMO_BROWSE_NEXT;
GetMsgOpts.Options += MQGMO_LOGICAL_ORDER;
                                 /* Loop until MQGET unsuccessful */
for (j = 1; CompCode == MQCC_OK; j++) {
     if (j == 1)                             /* First message only*/
         GetMsgOpts.Options += MQGMO_ALL_MSGS_AVAILABLE;

     pmdin = memcpy(pmdin, &MsgDesc, sizeof(MQMD) );
     pgmoin =  memcpy(pgmoin, &GetMsgOpts, sizeof(MQGMO) );
     memset(Buffer,' ',BUFFERLENGTH);

     MQGET(Hconn, Hobj, pmdin, pgmoin, BufferLength, Buffer,
           &DataLength, &CompCode, &Reason);

     if (CompCode != MQCC_OK) {
         if (Reason != MQRC_NO_MSG_AVAILABLE)
             printf("\n MQGET %d, failed with CompCode:%d Reason:%d",
                     j,CompCode,Reason);
         else
             printf("\n \n \n No more messages ");
     }
     else {
             /* *** Process the message *** */
     }
} /* end of for loop */
```

Figure 47. Getting a Message Group

4.4.3 Compile the Programs

To compile the program use one of the commands in Table 17.

Table 17. Commands to Compile PUT_SEG1 and BCG_SEG1

Compiler	**Command**
Microsoft Visual C/C++	cl put_grp1.c mqm.lib cl bcg_grp1.c mqm.lib
IBM Visual Age C/C++	icc put_grp1.c mqm.lib icc bcg_grp1.c mqm.lib
CSet++ for AIX	xlc put_grp1.c -l mqm -o put_grp1 xlc bcg_grp1.c -l mqm -o bcg_grp1

4.4.4 Creating a Queue for Exercise 8

Setup for this example is fairly simple. We only need to create a queue in which to store our messages. Use runmqsc to create the queue GRPTEST1. Here are the commands:

```
runmqsc
define qlocal(GRPTEST1) like(SYSTEM.DEFAULT.LOCAL.QUEUE) REPLACE
end
```

4.4.5 Putting Messages in a Group

In this section, we will test the two programs put_grp1 and bcg_grp1. We will also use other programs to verify that the queue contains the messages we expect to be there.

4.4.5.1 Clear the Queue

If this is not the first time you have been through the execution of these programs use the standard sample get program to clear the queue:

```
amqsget GRPTEST1 (QMgrName)
```

Note: The queue manager name is optional. If you connect to the default queue manager you do not need to specify it.

4.4.5.2 Put an Incomplete Group in the Queue

Now put a group of messages. In this case we will put some messages in one window and not enter the null line which finishes the group:

```
D:\redbook> put_grp1 GRPTEST1
Sample put_grp1 start
target queue is GRPTEST1
This is message one of the group
This is message two of the group
This is message three of the group
```

4.4.5.3 Check What Is in the Queue

There are two ways to verify that there are now three messages on the queue:

- Using runmqsc and checking the depth of the queue
- Executing the standard browse sample program

Using runmqsc, type the commands shown in bold and you will see that the current depth of the queue is 3.

```
runmqsc
dis ql(GRPTEST3) curdepth
     1 : dis ql(GRPTEST3) curdepth
AMQ8409: display Queue details.
  Queue(SEGTEST3)                                    CURDEPTH(3)
end
```

The standard browse sample program will display the text of the three messages. Execute it and you will get the following output:

```
D:\redbook> amqsgbr GRPTEST3
Sample AMQSGBRO (browse) start

Messages for GRPTEST3
1 <This is message one of the group>
2 <This is message two of the group>
3 <This is message three of the group>
no more messages
Sample AMQSGBRO (browse) end
```

4.4.5.4 Browse the Messages in the Queue

To see more details of all these messages, headers as well as contents run the standard amqsbcg sample program:

```
D:\redbook c:\mqm\tools\c\samples\bin\amqsbcg GRPTEST3
```

Figure 48 on page 98 shows the header and the text of the first message.

Note: The sample amqsbcg reads physical messages and it does not care whether they are part of a group or not.

The message ID is different for each message, of course. Table 18 shows the other message header fields we are interested in for all three messages of the incomplete group.

Table 18. New Fields in Message Descriptor

Field	Msg 1	Msg 2	Msg 3
MsgSeqNumber	1	2	3
Offset	0	0	0
MsgFlags	8	8	8
OriginalLength	32	32	34

```
 MQGET of message number 1
****Message descriptor****

  StrucId  : 'MD  '  Version : 2
  Report   : 0  MsgType : 8
  Expiry   : -1  Feedback : 0
  Encoding : 546  CodedCharSetId : 437
  Format : 'MQSTR   '
  Priority : 0  Persistence : 0
  MsgId : X'414D51204247414745202020202020200005D000034F2E634'
  CorrelId : X'000000000000000000000000000000000000000000000000'
  BackoutCount : 0
  ReplyToQ       : '                                                '
  ReplyToQMgr    : 'QMGR1                                           '
  ** Identity Context
  UserIdentifier : 'OS2         '
  AccountingToken :
  X'0131000000000000000000000000000000000000000000000000000000000000'
  ApplIdentityData : '                                '
  ** Origin Context
  PutApplType    : '4'
  PutApplName    : 'D:\REDBOOK\PUT_GRP1.EXE     '
  PutDate  : '19980224'    PutTime  : '15243641'
  ApplOriginData : '    '

  GroupId : X'414D51204247414745202020202020200005D010034F2E634'
  MsgSeqNumber   : '1'
  Offset         : '0'
  MsgFlags       : '8'
  OriginalLength : '32'

****   Message      ****

 length - 32 bytes

00000000:  5468 6973 2069 7320 6D65 7373 6167 6520 'This is message '
00000010:  6F6E 6520 6F66 2074 6865 2067 726F 7570 'one of the group'
```

Figure 48. A Message in a Group

The fields contain the following information:

- MsgFlags

  For all messages this has a value of 8. Looking at the C header file, this is equal to the value of MQMF_MSG_IN_GROUP.

- MsgSeqNumber

  This represents the logical sequence of the messages in the group.

- Offset

  This is always 0 since all of our messages are complete logical messages; there is no segmentation here.

- GroupId

  This is the same in all cases. We have one group of messages.

4.4.6 Getting Messages of a Group

At this time, the queue contains three messages. Table 18 on page 97 shows that the third message does not contain the flag MQMF_LAST_MSG_IN_GROUP. The put program is written in a way that the last message is put after the user of amqsput presses the Enter key without typing any message text. The length of the last message is 0.

Let us use the program bcg_grp1 to read the messages in the queue. The command and its result are shown below:

```
[D:\redbook]bcg_grp1 GRPTEST3

bcg_grp1 - starts here
**********************

 MQOPEN - 'GRPTEST3'

 No more messages
 MQCLOSE
 MQDISC
```

So, why are there no messages on the queue? Because we have not created a message that finishes the group. Going back to the window with the put_grp1 program running enter a null line to finish the program and go through the logic which adds another (empty) record to the queue. Then repeat the execution of bcg_grp1. Now you will see four messages.

Figure 49 on page 100 shows the last segment of the group. Each of the messages has a different message ID. Table 19 on page 101 shows the fields in the message headers we are interested in.

```
 MQGET of message number 4
****Message descriptor****

  StrucId  : 'MD  '  Version : 2
  Report   : 0  MsgType : 8
  Expiry   : -1  Feedback : 0
  Encoding : 546  CodedCharSetId : 437
  Format : 'MQSTR   '
  Priority : 0  Persistence : 0
  MsgId : X'414D51204247414745202020202020200005D000034F2E794'
  CorrelId : X'000000000000000000000000000000000000000000000000'
  BackoutCount : 0
  ReplyToQ       : '                                                '
  ReplyToQMgr    : 'QMGR1                                           '
  ** Identity Context
  UserIdentifier : 'OS2         '
  AccountingToken :
  X'0131000000000000000000000000000000000000000000000000000000000000'
  ApplIdentityData : '                                '
  ** Origin Context
  PutApplType    : '4'
  PutApplName    : 'D:\REDBOOK\PUT_GRP1.EXE     '
  PutDate  : '19980224'    PutTime  : '15302835'
  ApplOriginData : '    '

  GroupId : X'414D51204247414745202020202020200005D010034F2E634'
  MsgSeqNumber   : '4'
  Offset         : '0'
  MsgFlags       : '24'
  OriginalLength : '0'

****   Message      ****

 length - 0 bytes

00000000:                                         '                '
```

Figure 49. A Last Message in a Group

Table 19. Fields in Message Descriptor for a Message Group				
Field	**Msg 1**	**Msg 2**	**Msg 3**	**Msg 4**
MsgSeqNumber	1	2	3	4.
Offset	0	0	0	0
MsgFlags	8	8	8	24
OriginalLength	32	32	34	0

The fields contain the following information:

- GroupId

  This has the same value for all messages.

- MsgSeqNumber

  This is sequential representing the logical order of the messages in the group.

- MsgFlags

  This has the value 8 for all of the messages except the last. 8 is the value for MQMF_MSG_IN_GROUP. The last message has a decimal value of 24 which is the equivalent of MQMF_MSG_IN_GROUP plus MQMF_LAST_MSG_IN_GRP.

- Offset

  This always has a value of 0. This is used for segmentation, not for message groups.

4.4.7 Summary

Message groups can be used, as demonstrated here, to maintain a logical sequence in a logical, application-oriented grouping of messages. You can also use it to ensure you do not process a message until all of the related messages have arrived on the queue.

Chapter 5. Remote Administration and Windows NT Security

In this chapter, we discuss how to administer objects of a different queue manager in the same workstation, in another workstation in the workgroup, or another workstation in the domain.

We want to create an environment to administer a remote queue manager from a local queue manager. We will especially concentrate on the security issues for this.

To get familiar with remote administration we first create a second queue manager on our local machine and define all the MQ objects to enable *remote* administration. In this stage we will not have any problems with security, as you will have the same rights for both queue managers. We will also explore the use of the Service Control Manager. This is a feature of Windows NT to start programs, such as MQSeries automatically at startup of the machine.

In the second part of the exercise we set up real remote administration to another machine. We will consider two situations:

- Both workstations have their own security database; no primary domain controller is involved. This is the so called workgroup environment.
- Both workstations are member of a Windows NT domain. To be able to test this, a dedicated primary domain controller is necessary in the network.

Note: User IDs used with MQSeries must be less than 13 characters and may not contain spaces. This precludes the use of the default ″Administrator″ user ID.

5.1 MQSeries Security Background

In the previous version of MQSeries for Windows NT, only one security database was used for checking the authority of users. If the queue manager was started with a local user ID, then it used only the local security database. If the queue manager was started with a domain user ID, only the domain security database was used. Also, the user ID SYSTEM, which is always defined in Windows NT, was considered a local user ID, and this made it difficult and less useful to run MQSeries in silent mode as a Windows NT service.

When MQSeries runs with a domain user ID, then there is only one group ″mqm″ that controls the administration rights for MQSeries. Every user included in the group ″mqm″ can control all the queue managers in the whole domain.

5.2 Security Improvements

During installation of MQSeries for Windows NT, the local group ″mqm″ is created on the machine where MQSeries is installed. This local group can include any type of principals such as other local groups or global groups. If MQSeries is installed on a PDC, then the global group ″Domain mqm″ is created. When you install MQSeries later on a machine in that domain, the global group ″Domain mqm″ is added to the local group ″mqm″.

MQSeries will search both local and domain security databases, until it can come to a conclusion. There is thus no longer a difference between MQSeries running with a local or a domain user ID.

Note: Be aware of performance impact of remote resolution of principals.

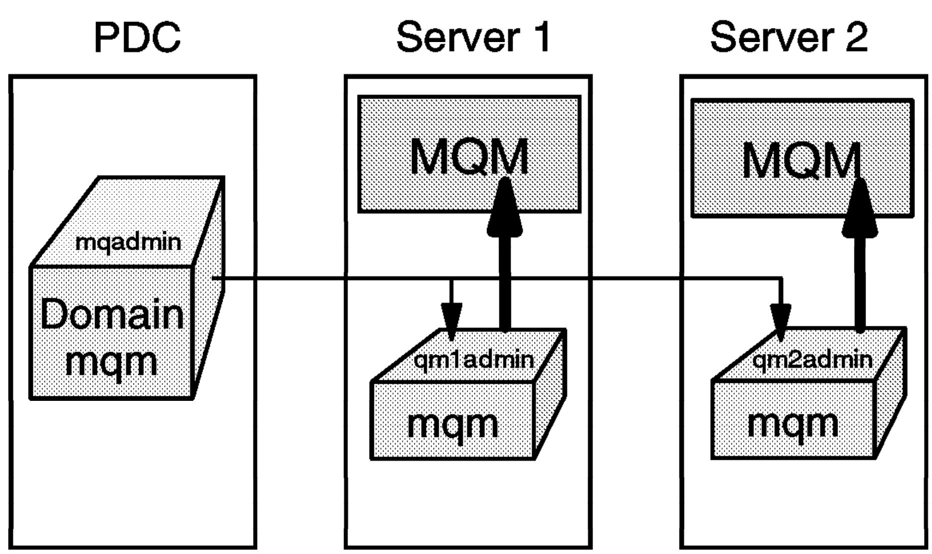


Figure 50. Granularity Example

Figure 50 shows three Windows NT systems, a PDC and two servers. Here, the PDC does not have MQSeries installed. The global group Domain mqm is created and the user mqadmin is a member of this group. You can insert any user in this group that needs domain-wide MQSeries administration authority.

On both servers, MQSeries is installed. The local groups mqm have been created during the installation. The groups include the users mq1admin and mq2admin, respectively. Also included in the two local groups mqm is the global group Domain mqm which contains the user mqadmin. The user mqadmin can control both systems. Users that only need administration authority for a single queue manager should be included in the local group mqm. In this example, qm1admin can only administer the queue managers on Server1, and mq2admin can only administer the queue managers on Server2.

Think of Server1 as a production machine and Server2 as a development system. In earlier MQSeries versions, this setup was only possible with locally defined user IDs. It could not be used by domain user IDs.

5.3 Remote Administration Basics

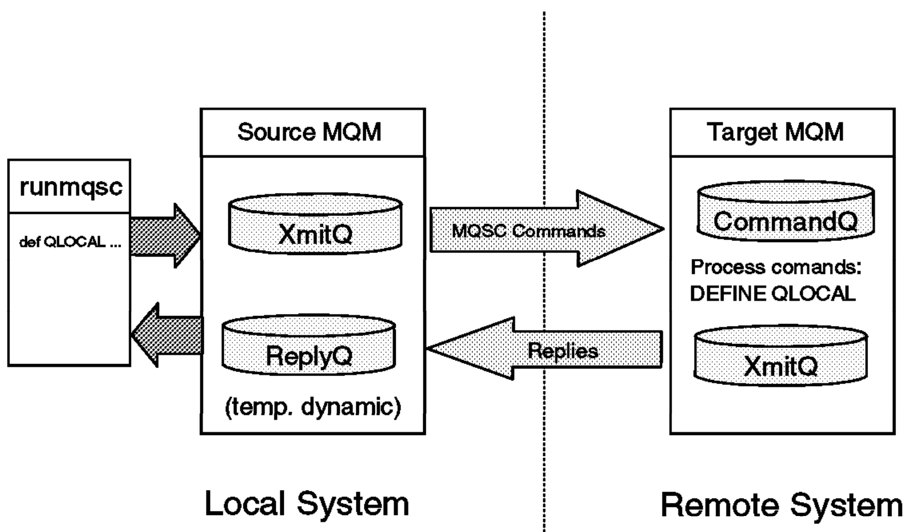


Figure 51. Remote Administration

When using runmqsc in the indirect mode, for example, when you execute:

```
runmqsc -w 15 RemoteQueueManager,
```

then the runmqsc program will connect to the default local queue manager and open the remote queue SYSTEM.ADMIN.COMMAND.QUEUE. Each mqsc command will be transformed into a command message. Runmqsc puts the command message on the transmission queue that leads to the remote queue manager. Thus, that is the first thing to do: create a transmission queue with the same name as the remote queue manager, or a remote queue manager alias pointing to a suitable transmission queue.

Messages on a transmission queue will not travel to the remote destination without a sender channel. And a sender channel needs a receiver channel on the remote side. That makes two channels to define, one on each end.

The destination is the queue SYSTEM.ADMIN.COMMAND.QUEUE. This queue is one of the standard queues and should have been defined when the queue manager was created. In the remote system, the command server has to be started explicitly with the command:

```
strmqcsv [QueueManagername]
```

Note: You do not have to start the command server when you use runmqsc locally. This is also referred to as direct mode.

The destination queue of the reply message is a temporary queue based on SYSTEM.MQSC.REPLY.QUEUE. This model queue is also one of the predefined queues.

The reply message will be put on a transmission queue that leads back to the sender of the command. We will define this transmission queue and the sender channel to serve it.

The number 15 in the above command is the number of seconds you want to wait to receive a reply from the remote location.

To protect our MQSeries resources, the command server executes the command and puts the reply message with the authority of the sender of the command. We will give these authorities via inclusion of the sender′s user ID in the local group ″mqm″.

Note: The command server uses the concept of *Alternate User Authority*. The user ID of the originator is found in the message descriptor of the command message. Therefore, your user ID needs to be defined with the correct authority in the remote machine. The option MQPMO_PASS_IDENTYTY_CONTEXT causes your user ID to be put in the message descriptor of the reply message.

Don′t forget that MQSeries transforms your user ID to lowercase and truncates it to 12 characters. It is this user ID that will be presented to the remote system.

5.4 Exercise 9: Remote Administration in One Machine

In the following sections, we explain what you have to do to administer objects of another queue manager that reside in the same machine. For this exercise, we create two queue managers:

- QMGR1
- QMGR2

5.4.1 Enable The Local Default Queue Manager

First, enable the local queue manager QMGR1 for remote administration. Create the script file qmgr1.in and define the channels and transmission queue leading to the second queue manager:

```
define qlocal(QMGR2) +
       usage(xmitq) +
       trigger +
       trigtype(first) +
       initq(SYSTEM.CHANNEL.INITQ) +
       replace

define chl(QMGR1.QMGR2) +
       chltype(sdr) +
       conname('127.0.0.1(1415)') +
       trptype(tcp) +
       xmitq(QMGR2) +
       replace

alter  qmgr +
       chad(enabled)
```

Execute runmqsc in a command window to define these objects and check the output with any editor:

```
runmqsc < qmgr1.mqs > qmgr1.out
```

Notes:

1. It is no longer necessary to define a process for starting the channels automatically. This is an enhancement of Version 5.
2. Nor is it required to define the receiver channel if you have enabled another new feature called "channel autodetect". It is enabled with the command "alter qmgr chad(enabled)".
3. As you can see, we specify explicitly the port number for QMGR2. Only one queue manager (QMGR1 in this example) can use the default port number 1414.

4. We are using the TCP/IP "loopback" address 127.0.0.1. You could also use your local hostname.

You find the assigned port numbers in Request for Comment (RFC) 1060. Some of them are shown in the following Windows NT file:

```
c:\winnt\system32\drivers\etc\services
```

5.4.2 Creating The Second Queue Manager

Next, create the second queue manager,QMGR2, and its objects. For this example, we need also a dead letter queue, which is not automatically created.

We create the queue manager and start it with the following commands:

```
crtmqm -u SYSTEM.DEAD.LETTER.QUEUE QMGR2
strmqm QMGR2
```

Then we create the script file qmgr2.mqs and define the channels and the transmission queue, and enable channel autodetect:

```
define qlocal(QMGR1) +
       usage(xmitq) +
       trigger +
       trigtype(first) +
       initq(SYSTEM.CHANNEL.INITQ) +
       replace

define chl(QMGR2.QMGR1)
       chltype(sdr) +
       conname('127.0.0.1(1414)') +
       xmitq(QMGR1) +
       trptype(tcp) +
       replace

alter  qmgr +
       chad(enabled)
```

Execute runmqsc to define these objects and check the output:

```
runmqsc QMGR2 < qmgr2.mqs > qmgr2.out
```

5.4.3 Enable Automatic Startup

Enable automatic startup of the two queue managers on your single machine via the Service Control Manager. To do this follow these steps:

1. Create a script file startup1.cmd that contains the following commands:

```
strmqm [QMGR1]
strmqcsv [QMGR1]
runmqchi -q SYSTEM.CHANNEL.INITQ [-m QMGR1]
runmqlsr -t tcp -p 1414 [-m QMGR1]
```

2. Create a script file startup2.cmd that contains the following commands:

   ```
   strmqm [QMGR2]
   strmqcsv [QMGR2]
   runmqchi -q SYSTEM.CHANNEL.INITQ [-m QMGR2]
   runmqlsr -t tcp -p 1415 [-m QMGR2]
   ```

   Note: The script files should not have any blank lines or comments. If something goes wrong, re-create the script file. It can happen that your editor has inserted control characters in the script file that confuse the Service Control Manager.

3. Configure the Service Control Manager:

   ```
   scmmqm -a -s [full_path]startup1.cmd QMGR1
   scmmqm -a -s [full_path]startup2.cmd QMGR2
   ```

4. Go to the Control Panel via Start - Settings - Control Panel, and click on the Services icon.

   a. IBM MQSeries should be in the list.

   b. Select it and click on the Startup... button.

   c. A new window will appear wherein you configure MQSeries for automatic startup with no interaction to the desktop using the Admin user ID.

5. Reboot the machine.

   MQSeries starts under the Admin user ID.

   Be patient when you do this

   If your machine is short on memory or lacks CPU power, it may take some time to start two queue managers!

6. Log on and open a command window.

7. Verify that both queue managers are active and that the channel initiators are running.

   - For QMGR1, use the following commands:

     ```
     runmqsc
     display ql(SYSTEM.CHANNEL.INITQ)
     end
     ```

The channel initiator is running when the value for IPPROCS in the SYSTEM.CHANNEL.INITQ is 1.

- For QMGR2, use the following commands:

```
runmqsc QMGR2
display ql(SYSTEM.CHANNEL.INITQ)
end
```

  The channel initiator is running when the value for IPPROCS in the SYSTEM.CHANNEL.INITQ is 1.

8. Verify that the command server is active. At a command prompt type:

```
dspmqcsv [QMGR1]
dspmqcsv QMGR2
```

5.4.4 Test It Out

If everything ran fine after the reboot, then we are ready for ″remote″ administration. Start a runmqsc session to your second queue manager with the following command:

```
runmqsc -w 15 QMGR2
```

Then try a command such as display qmgr.

Note: You should see a slower response time than previously because channels need to be started on both sides. In fact, the first time you do this it will go through definition of the channel as well if you have CHAD enabled. By the way, the channel definitions remain after they have been automatically defined. The definition process is as permanent as if you had issued the command.

You may try out some more commands and verify their correct execution.

5.4.5 Remove the Second Queue Manager

You should now be familiar with the MQSeries aspects of remote administration. The second queue manager is no longer needed for the remainder of this exercise. It will be used later for different examples. If you wish to remove it this is how you reconfigure the Service Control Manager and delete QMGR2. This will save some resources.

1. Stop the service ″IBM MQSeries″. This can be done via Services in the Control Panel or with the command:

```
net stop IBMMQSERIES
```

2. Go to the Windows NT Event Viewer (Start - Programs - Administrative Tools) and verify that both Queue Managers are stopped. You should see Event Id 8004 for both queue managers in the application log.

3. Go to the Windows NT Task Manager (click with the right mouse button on the task bar) and select the Processes tag. You will see two instances of runmqlsr are still running. With the new version, there is a command to stop these processes in a clean way.

4. Execute the following commands in a command prompt window:

```
endmqlsr -m [QMGR1]
endmqlsr -m QMGR2
```

5. To unload the startup script from the Service Control Manager, type the following command in a command prompt window:

```
scmmqm -d QMGR2
```

6. Delete the queue manager: dltmqm QMGR2

7. We are now ready to restart the service ″IBM MQSeries″. In the command window, type:

```
net start IBMMQSERIES
```

Verify the successful restart by starting a runmqsc session. Eventually, you may want to delete the objects that refer to QMGR2.

5.5 Exercise 10: Remote Administration in a Workgroup

For this example, we will connect a second real machine. Basically, the way to enable remote administration is the same as in the previous example. We need the remote queue manager and agree on the names of the channels.

If your workstation is a member of a domain, you should now log on locally. In our case, we log on as mqadmin1, which is part of user group mqm.

1. We will modify remote1.mqs created in the previous exercise and change the references to QMGR2 to the name of the remote queue manager (RQMGR2):

```
define qlocal(RQMGR2) +
       usage(xmitq) +
       trigger +
       trigtype(first) +
       initq(SYSTEM.CHANNEL.INITQ) +
       replace

define chl(QMGR1.RQMGR2) +
       chltype(sdr) +
       conname('9.24.104.116') +
       xmitq(RQMGR2) +
```

```
       trptype(tcp) +
       replace

define chl(RQMGR2.QMGR1) +
       chltype(rcvr) +
       trptype(TCP) +
       replace
```

2. We save the script as remote3.mqs. Of course, if you are doing this you need to use the correct host name or IP address.

3. Execute runmqsc to define these objects and check the output:

   runmqsc < remote3.mqs > remote3.out

4. Start a runmqsc session with the remote queue manager and issue any command.

```
C:\redbook>runmqsc -w 15 RQMGR2
84H2004,6539-B43 (C) Copyright IBM Corp. 1994, 1997.  ALL RIGHTS RESERVED.
Starting MQSeries Commands.

dis qmgr
     1 : dis qmgr
AMQ8416: MQSC timed out waiting for a response from the command server.
```

No reply!

What went wrong?

5. First, we verify that our message has been sent. Open a local runmqsc session and execute the following MQSC command to see if the message is still on the transmission queue:

```
dis ql(rqmgr2) curdepth
     1 : dis ql(rqmgr2) curdepth
AMQ8409: Display Queue details.
   QUEUE(RQMGR2)                           CURDEPTH(0)
```

6. The queue is empty. Now, go to the remote location and verify that your command has been processed. In a runmqsc session on that machine, type the MQSC command:

```
dis ql(SYSTEM.ADMIN.COMMAND.QUEUE) curdepth ipprocs
     3 : dis ql(SYSTEM.ADMIN.COMMAND.QUEUE) curdepth ipprocs
AMQ8409: Display Queue details.
   QUEUE(SYSTEM.ADMIN.COMMAND.QUEUE)       IPPROCS(1)
   CURDEPTH(0)
```

This is correct. Curdepth should be zero and IPPROCS should be non-zero.

7. The next step is to verify the transmission queue to your own queue manager. In the same runmqsc session, execute the command:

```
dis ql(qmgr1) curdepth
     5 : dis ql(qmgr1) curdepth
AMQ8409: Display Queue details.
   QUEUE(QMGR1)                            CURDEPTH(0)
```

Look at the current depth. This should be zero.

So, where is the message?

8. We now look at the current depth of the dead letter queue and see that it is non-zero!

```
dis ql(system.dead.letter.queue) curdepth
     6 : dis ql(system.dead.letter.queue) curdepth
AMQ8409: Display Queue details.
   QUEUE(SYSTEM.DEAD.LETTER.QUEUE)         CURDEPTH(1)
```

This is an intended error to show you how to handle errors and set up problems in this area. To know why MQSeries has put the message in the dead letter queue, we will use the MQSeries sample program AMQSBCG.

9. In a command window at the remote location execute:

amqsbcg SYSTEM.DEAD.LETTER.QUEUE RQMGR2 > out

10. Open the file ″out″ in an editor. This will show a formatted dump of each message on the dead letter queue.

For each message on the dead letter queue, the original message is prefixed with a dead letter header, which is not formatted by the program. Looking at the message data itself, we then look for the eye-catcher ″DLH″. After this, there should be the version number ′0100 0000′. The next 4 bytes should show you the reason why this message has been put on the dead letter queue. We find 07F3 in reverse byte order (F307). 7F3 in decimal is 2035 which is a normal MQSeries reason code.

If you find something else when you look at similar messages you may need to use the Windows NT calculator.

The *Application Programming Reference* says this means MQRC_NOT_AUTHORIZED. If you find something other than 7F3, check out in the manual what it means and correct the error. Please note that the code you find on that location, can also be a feedback code. The standard MQSeries defined feedback codes are in the range 256-400 and

```
 MQOPEN - 'SYSTEM.DEAD.LETTER.QUEUE'


 MQGET of message number 1
****Message descriptor****

  StrucId  : 'MD  '  Version : 2
  Report   : 16777216  MsgType : 1
  Expiry   : -1  Feedback : 0
  Encoding : 546  CodedCharSetId : 437
  Format : 'MQDEAD  '
  Priority : 0  Persistence : 0
  MsgId : X'414D512052514D47523220202020202020C026003513400000'
  CorrelId : X'000000000000000000000000000000000000000000000000'
  BackoutCount : 0
  ReplyToQ       : 'AMQ.1998030614005704                            '
  ReplyToQMgr    : 'QMGR1                                           '
  ** Identity Context
  UserIdentifier : 'mqadmin1    '
  AccountingToken :
  X'0131000000000000000000000000000000000000000000000000000000000000'
  ApplIdentityData : '                                '
  ** Origin Context
  PutApplType    : '11'
  PutApplName    : 'C:\MQM\BIN\AMQPCSEA.EXE     '
  PutDate  : '19980306'    PutTime  : '17064278'
  ApplOriginData : '    '

  GroupId : X'000000000000000000000000000000000000000000000000'
  MsgSeqNumber   : '1'
  Offset         : '0'
  MsgFlags       : '0'
  OriginalLength : '252'

****   Message      ****

 length - 252 bytes

00000000:  444C 4820 0100 0000 F307 0000 414D 512E 'DLH ....ó...AMQ.'
00000010:  3139 3938 3033 3036 3134 3030 3537 3034 '1998030614005704'
00000020:  2020 2020 2020 2020 2020 2020 2020 2020 '                '
00000030:  2020 2020 2020 2020 2020 2020 514D 4752 '            QMGR'
00000040:  3120 2020 2020 2020 2020 2020 2020 2020 '1               '
00000050:  2020 2020 2020 2020 2020 2020 2020 2020 '                '
00000060:  2020 2020 2020 2020 2020 2020 2202 0000 '            "...'
00000070:  B501 0000 4D51 4144 4D49 4E20 0B00 0000 '·...MQADMIN ....'
00000080:  8F00 0000 0000 0000 0000 0000 0000 0000 '................'
00000090:  0000 0000 0000 0000 0000 0000 3139 3938 '............1998'
000000A0:  3033 3036 3137 3036 3432 3736 0100 0000 '030617064276....'
000000B0:  2400 0000 0100 0000 2600 0000 0100 0000 '$.......&.......'
000000C0:  0100 0000 0000 0000 0000 0000 0200 0000 '................'
000000D0:  0300 0000 1000 0000 F903 0000 0100 0000 '........ù.......'
000000E0:  0400 0000 1C00 0000 C60B 0000 0000 0000 '........Æ.......'
000000F0:  0800 0000 6469 7320 716D 6772           '....dis qmgr    '
```

Figure 52. Message in Dead Letter Queue

are documented in the *MQSeries Application Programming Reference*, SC33-1673.

Now, why did we get a security violation? Because we work in a Workgroup environment, every machine uses its own local security database. And, in normal circumstances, our user ID will not be defined on the remote machine.

11. Start the user manager on the remote machine and create new user mqadmin1 included in the ″mqm″ group on that machine.

12. Returning to the runmqsc session at your local workstation we try another command:

```
dis qmgr qmname
     8 : dis qmgr qmname
AMQ8408: Display Queue Manager details.
   QMNAME(RQMGR2)
```

When you do this if you still get no reply, repeat the above problem handling method to find out about your message.

Note: It may be a good idea to clear the remote dead letter queue with the MQSC command:

```
clear ql(SYSTEM.DEAD.LETTER.QUEUE)
```

5.6 Exercise 11: Remote Administration in a Domain

From an MQSeries point of view, there is no difference in the administration of queue managers in a workgroup or in a domain. The objects you need are the same.

For remote administration in a domain, you need to log on to your machine with a domain user ID.

When the command server on the remote machine checks your authority, it will find that your user ID is known because all user IDs are centrally maintained.

To have the correct authority, your domain user ID needs to be included in the local group ″mqm″ on the remote machine. Or, add your user ID to the global group ″Domain mqm″ on the PDC and include that global group in the local group ″mqm″ on the remote machine.

5.7 Summary

In the previous exercises, we have discussed security and remote administration in a Windows NT environment. However, the same applies for other platforms.

If you want to administer an AIX queue manager from a Windows machine, then your Windows user ID will flow to AIX and will be used to check your authorities, which means at least that your user ID needs to be defined on AIX, even if you do nothing else on AIX.

If you want to administer an MVS queue manager, you should use the -x parameter when starting runmqsc. MQSeries on MVS uses a different command message format and the name of the command queue is SYSTEM.COMMAND.INPUT. Also, on MVS the command server is started automatically but it can be stopped.

Note: Do not forget that MVS user IDs are restricted to 8 characters.

Some additional hints:

The following is a list of problems that we have encountered during the design and test of the exercises.

1. Common mistake: In Exercise 9, the same port number was used for both queue managers.

2. Common mistake: The same objects have been created for both queue managers, such as:

   ```
   runmqsc QM1 < qmgr1.mqs > out
   runmqsc QM2 < qmgr1.mqs > out
   :
   :
   ```

3. Channel auto definition:

   Do not forget to reset the sender channel before you test this. Normally the sender/receiver channels have been used in the previous exercise. The sender channel will have a non-zero message sequence number. If you delete the receiver channel, then the new one (created by MQSeries) will have a sequence number of zero.

4. Another pitfall:

   When stopping the service IBM MQSeries, the Service Control Manager will say that it is stopped, but actually it isn't. Some processes are still running and are stopping asynchronously.

   At this time, the user can try to restart the IBM MQSeries service. This is not possible, but the Service Control Manager will say that the restart was successful! Always look at the Event Viewer and wait until you see Event ID 8004, or watch the Task Manager for amq* processes.

AMQ08101

If you get AMQ08101 - Unexpected error (...) the re-boot the system. MQSeries did not clean up properly after you logged off with another user ID.

Chapter 6. Reference Message

In this chapter we will demonstrate the usage of a new facility available with MQSeries Version 5 called reference messages. Looking at Figure 53 you will see that in order to use reference messages you need to implement channel exits **(**3**)**. MQSeries Version 5 provides a sample exit and sample programs which get and put the reference messages if you want to use them. The sample programs can be used to move a simple flat file between two systems with MQSeries servers as depicted in Figure 53.

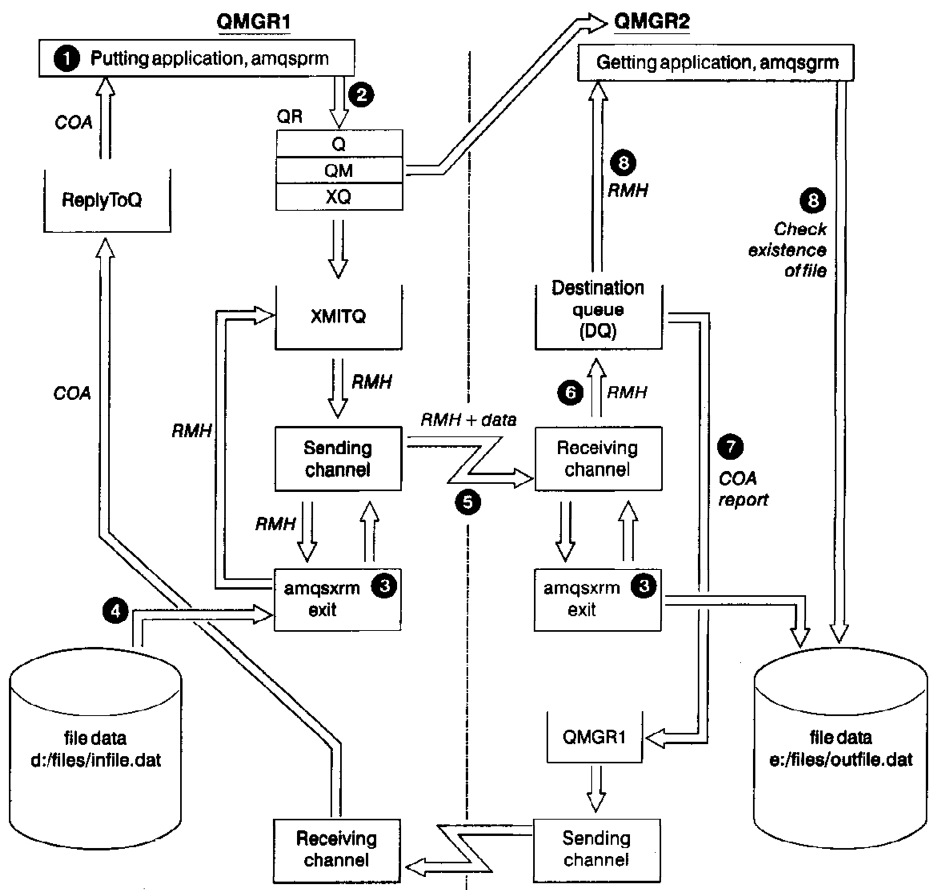


Figure 53. Reference Message Flow (Sample Programs)

Using reference messages, the sending and receiving programs use a new structure, the MQRMH or MQ Reference Message Header. The format of the MQRMH can be found in Figure 61 on page 136. A detailed description of the fields is in the *MQSeries Application Programming Reference*, SC33.1673. The sending program sends only the reference to a file, not the file itself. Similarly, the receiving program only receives notification of the arrival of a file via a reference message, not the actual file. It is the sending MCA which takes care of reading the file and sending it to the receiving MCA.

6.1 Security Issues

Before implementing reference messages be sure to read *MQSeries Application Programming Guide*, SC33-0807 for a complete description of how it works as well as a description of the security implications of implementing it. The reference message exit as well as the MCA run under the security of group mqm. This is most likely different than the authority of the sending and receiving programs.

6.2 The Sample Programs

There are three sample programs involved here:

amqsprma.exe This is the program which PUTs the reference message.

amqsgrma.exe This is the program which GETs the reference message.

amqsxrm.dll This is the exit program which runs on both sides of the channel and receives the reference message, reads the file, and sends it across the MCA.

The source files can be found in the directory \mqm\tools\c\samples.

Figure 53 on page 119 illustrates how the three programs are used.

1. The sending application, amqsprm, puts a reference message on the remote queue QR. The file that has to be sent is on the hard drive.
2. Since a remote queue is only the local definition of a queue that belongs to another queue manager, the message is really put into the transmission queue for the target queue manager. A transmission header is prepended which contains the target queue. This information is obtained from the remote queue definition.
3. When the message channel agent (MCA) is about to transmit the message the channel exit routine amqsxrm is called.
4. This routine reads the file infile.dat from the hard drive and appends it to the message.

5. The message consisting of the reference message header (RMH) and the data (the file) is transmitted to the receiving channel.
6. The receiving MCA calls the channel exit routine amqsxrm which stores the file on the hard drive of the receiving system.
7. The queue manager puts the reference message (without the data) into the target queue.
8. A ″confirm on arrival″ (COA) message is sent to the sending queue manager. The file may now be deleted.
9. The target application, AMQSGRM, gets the reference message and knows now that the file has arrived and is stored in the directory specified in the reference message header.

6.2.1 Program Logic for the PUT Program

- Parse and validate the input parameters. The parameters are described in Table 20 on page 124.
- MQCONN to the local queue manager.
- Determine the queue manager′s coded character set ID with MQINQ.
- MQOPEN a model queue for the report messages.
- Create the reference message.
- MQPUT1 the reference message to the target queue.
- While MQGET WAIT returns a message and COA message not received:

  Display contents of exception or COA message.
- MQCLOSE model (temporary dynamic) queue.
- MQDISC from queue manager.

6.2.2 Program Logic for the GET Program

- Extract queue and queue manager names from the trigger message.
- MQCONN to the specified queue manager.
- Obtain the queue manager′s CCSID using MQINQ.
- Open a temporary dynamic queue for the report messages.
- MQOPEN the specified queue.
- While MQGET WAIT returns a message:

  If the message is a reference message check existence of object.
- MQCLOSE the queue.
- MQDISC from the queue manager.

6.2.3 Definitions for the Sample Programs

In order to run the reference messages sample we will use the queue managers and channel definitions we created in Chapter 5, “Remote Administration and Windows NT Security” on page 103. Specifically, we will need to add pointers to the existence and location of the exit program as well as tell the exit program what kind of file can be handled. In this case we are going to handle files of the type FLATFILE.

Here is the definition of the sender channel in QMGR1 in the script file refmsg1.mqs:

```
define chl(QMGR1.QMGR2) +
       chltype(sdr) +
       descr('Channel to QMGR2')  +
       conname('127.0.0.1(1415)') +
       msgexit('d:\mqm\exits\amqsxrm.dll(MsgExit)')  +
       msgdata(FLATFILE)  +
       trptype(tcp) +
       xmitq(QMGR2) +
       replace

define qr(REFMSG)  +
       rname(REFMSG) +
       rqmname(QMGR2) +
       xmitq(QMGR2)   +
       replace
```

Notes:

1. The ″msgexit″ keyword specifies the name and location of the dll, the executable form required for exit programs in Windows NT and OS/2.
2. The \mqm\exits directory is the default location for exit programs.
3. We have also defined a remote queue, REFMSG, which will be the destination queue for our reference message.

The receiving queue manager QMGR2 needs several objects defined. The script file is refmsg2.mqs.

```
define chl(QMGR1.QMGR2) +
       chltype(rcvr) +
       descr('Channel from QMGR1')  +
       msgexit('d:\mqm\exits\amqsxrm.dll(MsgExit)')  +
       msgdata(FLATFILE)  +
       trptype(tcp) +
       replace
```

```
define ql(INITQ)  +
       replace

define process(PROC) +
       applicid('d:\mqm\tools\c\samples\bin\amqsgrm.exe') +
       replace

define ql(REFMSG) +
       initq(INITQ)  +
       process(PROC) +
       trigger +
       trigtype(first) +
       replace
```

Notes:

1. First, notice the corresponding receiver channel definition on queue manager QMGR2 which points to the same exit program.
2. The next definition is for the initiation queue, since the sample Get Reference Message program is designed to be triggered.
3. The process definition points to the sample Get program.
4. And finally, there is a definition for our local destination queue which corresponds to the remote definition on QMGR1.

Now, update both of the queue managers with the new object definitions. Execute the following commands:

```
C:\test>runmqsc QMGR1 < refmsg1.mqs > refmsg1.out
C:\test>runmqsc QMGR2 < refmsg2.mqs > refmsg2.out
```

6.2.4 Running the Sample Programs

After checking the output files for any errors we can proceed to test the sample programs. Remember that we are using these two queue managers as they were set up in the previous chapter. That means there are channel initiators, and listeners already running. We will still need to add the trigger monitor for QMGR2 (since we are only going to trigger the Get program on the one queue manager). We do this by entering the following on a command line prompt:

```
start runmqtrm -m QMGR2 -q INITQ
```

AMQSPRM, the put program, has several parameters. They are shown in Table 20 on page 124.

Table 20. Parameters for AMQSPRM	
Parameter	**Description**
/m queue-mgr-name	Name of local queue manager (optional). Default is the default queue manager.
/i source-file	Fully qualified name of source file to be transferred. (required). The name is limited to 256 characters but this can easily be changed.
/o target-file	Fully qualified name of file on the destination systems (optional). The name is limited to 256 characters but this can easily be changed. Default is the source filename.
/q queue-name	Destination queue to which the reference message is put (required).
/g queue-mgr-name	Queue manager on which queue, named in /q parameter, exists (optiona l). Defaults to the queue manager specified by the /m parameter or the default queue manager.
/t object-type	Object type (required). Limited to 8 characters.
/w wait-interval	Time (in seconds) to wait for exception and COA reports. Default is 15 seconds. Minimum value is 1.

For our purposes we will need to specify the source, the destination, the remote queue name and the type of file:

```
C:\test>amqsprm /i c:\test\refmsg1.mqs
               /o d:\junk\xyz.xyz /q REFMSG /t FLATFILE
AMQSPRM starting
Source file is c:\test\refmsg1.mqs
Destination file is d:\junk\xyz.xyz
Destination queue is REFMSG
Object type is FLATFILE
Destination queue manager is
Wait interval is 15 seconds
Reference message has arrived on destination queue
AMQSPRM ending
```

When amqsprm executes it specifies that it wants a COA message back from amqsgrm on the receiving queue manager. In this case we will wait for the default 15 seconds for it to arrive.

In the trigger monitor window you will see when the reference message arrives and whether it has successfully started amqsgrm.

```
Waiting for a trigger message

d:\mqm\tools\c\samples\bin\amqsgrm.exe "TMC    2REFMSG
                PROC
                                                    d:\mqm\tools\c\sample
mqsgrm.exe




    QMGR2                                           "
AMQSGRM starting
Queue manager name is QMGR2
Queue name is REFMSG
File d:\junk\xyz.xyz of type FLATFILE does exist
AMQSGRM ending
End of application trigger.
```

6.2.5 More Object Types

You can create multiple exits to handle different types of data. The sample program looks to the message data field to determine the type of data it can process. If we needed a file type other than FLATFILE we could create another exit and chain it to the first. Chaining is another new feature of MQSeries Version 5. In order to demonstrate exit chaining we will add another instance of the same exit program (amqsxrma) which will handle a new file type: NOTFLAT. We have to alter the channel definition in QMGR1 and the corresponding definition in QMGR2 as shown in Table 21.

Table 21. Two Channel Exits

QMGR1	QMGR2
define chl(QMGR1.QMGR2) + chltype(sdr) + trptype(tcp) + conname('127.0.0.1(1415)') + msgexit('d:\mqm\exits\amqsxrm.dll(MsgExit)', + 'd:\mqm\exits\amqsxrm.dll(MsgExit)') + msgdata(FLATFILE,NOTFLAT) + xmitq(QMGR2) + descr('Channel to QMGR2') + replace	define chl(QMGR1.QMGR2) + chltype(rcvr) + trptype(tcp) + msgexit('d:\mqm\exits\amqsxrm.dll(MsgExit)', + 'd:\mqm\exits\amqsxrm.dll(MsgExit)') + msgdata(FLATFILE,NOTFLAT) + descr('Channel to QMGR2') + replace

Now we can execute our Put program using file type NOTFLAT as follows:

```
C:\test>amqsprm /i c:\test\refmsg1.mqs /o d:\junk\yyy.yyy
                     /q REFMSG /t NOTFLAT
AMQSPRM starting
Source file is c:\test\refmsg1.mqs
Destination file is d:\junk\yyy.yyy
Destination queue is REFMSG
Object type is NOTFLAT
Destination queue manager is
Wait interval is 15 seconds
Reference message has arrived on destination queue
AMQSPRM ending
```

Just in case you want to try some other kind of file, say DUMMY, this is what you will get:

```
AMQSGRM starting
Queue manager name is QMGR2
Queue name is REFMSG
File d:\junk\yyy.yyy of type DUMMY    could not be found
AMQSGRM ending
End of application trigger.
```

6.3 Exercise 12: Building a Reference Message

This exercise shows you how to build, send and receive a file using a reference message. For a better understanding of the example and to make it shorter, some of the parameters are hard-coded. In the real world, the sending program would receive the necessary information as input parameters or, at least, interactively as keyboard input.

This sample application consists of two programs, the PUTREF program that generates the reference message and sends it, and the GETREF program that receives the reference message and reads the file from disk. Depending on the MQSeries definitions, both programs can run in the same or in different machines.

For this example we use the existing channel exit program amqsxrm.

6.3.1 Writing the PUTREF Program

This program sends a file to another queue manager. It contains the following functions:

- Connect to the default queue manager.
- Use MQINQ to get the queue manager′s name and CCSID.
- Build the reference message.

- Use MQPUT1 to send the reference and file.
- Disconnect from the queue manager.

The complete listing is in Appendix E, “Reference Message Example” on page 237. The interesting parts of the program are described below.

6.3.1.1 Defining the Reference Message

We define the reference message as shown below. The structure contains the reference message header MQRMH (see Figure 61 on page 136) and two 256-byte fields for paths and names of the source and target file.

```
#define MAX_FILENAME_LENGTH 256
typedef struct tagMQRMHX{
        MQRMH  ref;
        MQCHAR SrcName[MAX_FILENAME_LENGTH];
        MQCHAR DestName[MAX_FILENAME_LENGTH];
} MQRMHX;
 MQRMHX  refx = {{MQRMH_DEFAULT}};     /* reference message  */
```

Figure 54. Defining a Reference Message

6.3.1.2 Obtaining the Queue Manager′s CCSID

After we connect to the default queue manager we get its CCSID and save it. The CCSID will be stored in the reference message header.

```
char    BLANK48[MQ_Q_MGR_NAME_LENGTH+1] = ″″;
MQLONG  flags;
:
memcpy(od.ObjectQMgrName,BLANK48,MQ_Q_MGR_NAME_LENGTH);
flags = MQOO_INQUIRE;
od.ObjectType = MQOT_Q_MGR;
MQOPEN(Hcon,                     /* connection handle            */
       &od,                      /* object descriptor            */
       flags,                    /* inquiry flags                */
       &Hobj,                    /* object handle                */
       &CC,&Reason);             /* completion and reason codes  */
if (CC == MQCC_FAILED)
  printf(″MQOPEN queue manager ended with reason code %d\n″,Reason);
```

Figure 55. Open Queue Manager for Inquiry

The MQOPEN requires that the queue manager name in the object descriptor (od) is blank. Therefore, we move 48 blanks into this field. In the open flags we indicate that we want to inquire. What to inquire we specify in the object type.

Note: The default object type is a queue (MQOT_Q).

MQINQ requires us to define arrays for "selectors" and for the integer and character attributes we want to obtain. Selectors specify which attributes of the object (queue manager, queues of different types and process definitions) we want to get. For a complete listing refer to the *MQSeries Application Programming Reference*, SC33-1673.

1 In the example shown in Figure 56 on page 129 we want to get the CCSID which is an integer, and the name of the queue manager which is a character string. Therefore, we need to define three arrays, each of them large enough to hold the information:

Selectors — This is an array for the selectors. It must contain at least as many elements as selectors used.

IntArray — MQINQ returns in this array all integer attributes. The example shows two elements even though we request only one.

CharArray — MQINQ returns in this array all character attributes, concatenated. It must be at least as long as the sum of all attributes. We inquire about the queue manager name which is 48 bytes long and will fit in the 100-byte array.

2 Next we have to specify the two selectors in any order. All integer selectors start with MQIA and all character selectors with MQCA.

3 In the MQINQ you specify the usual connection handle from the MQCONN and the object handle from the MQOPEN. We also specify how many of the specified selectors the MQINQ shall use (2) and the array where they are specified. We inquire about one integer attribute and one 48-byte character attribute. We also know that the arrays are large enough for this API call (two integers and 100 characters).

4 When the MQINQ call is successful we save the queue manager name and the CCSID then close the object.

6.3.1.3 Building the Reference Message

Figure 57 on page 130 shows the statements that build a reference message. To make the program easier to understand all variables are hard coded.

1 We define two variables to hold the names and paths of the source and target files.

```
MQLONG  Selectors[4];       1
MQLONG  IntArray[2];
MQCHAR  CharArray[100];
⋮
MQLONG  QMgrCCSID = -1;                       /* QMgr CCSID        */
char    QMName[MQ_Q_MGR_NAME_LENGTH+1] = ""; /* queue manager name*/
⋮
Selectors[0] = MQIA_CODED_CHAR_SET_ID;    2
Selectors[1] = MQCA_Q_MGR_NAME;
⋮
MQINQ(Hcon, Hobj,                         3
      2L,                  /* number of selectors               */
      Selectors,           /* selector array                    */
      1L,                  /* number of integer selectors       */
      IntArray,            /* integer attributes                */
      48L,                 /* length of character attributes    */
      CharArray,           /* character attributes              */
      &CC,&Reason);        /* completion and reason codes       */

if (CC == MQCC_FAILED)
    printf("MQINQ failed with reason code %d\n", Reason);
else {
      QMgrCCSID = IntArray[0];            4
      memcpy(QMName,CharArray,MQ_Q_MGR_NAME_LENGTH);
}
```

Figure 56. Inquire Queue Manager Name and CCSID

2 The hard coded file names are moved into the variables. There are two backslashes. The C compiler assumes an expression with one backslash to be an escape character, such as "\n". The source file is in the directory \test and the receiving channel exit puts the file under a different name into the same directory.

3 The structure length includes the reference message header and the two 256-byte fields for the filenames. Refer to the definition in Figure 54 on page 127.

4 Encoding identifies the representation used for numeric data in the file. It is set to MQENC_NATIVE. This constant is environment-specific.

5 We obtained the CCSID with the MQINQ call explained above.

6 This flag indicates that the reference message represents the last part of the referenced object.

```
 char    infile [MAX_FILENAME_LENGTH+1];    1
 char    outfile[MAX_FILENAME_LENGTH+1];
:
strcpy (infile,"c:\\test\\dw.fil");      2
strcpy (outfile,"c:\\test\\dw1.txt");
:
refx.ref.StrucLength    = sizeof(refx);  3
refx.ref.Encoding       = MQENC_NATIVE;  4
refx.ref.CodedCharSetId = QMgrCCSID;     5
refx.ref.Flags          = MQRMHF_LAST;   6
memcpy(refx.ref.Format, MQFMT_STRING, (size_t)MQ_FORMAT_LENGTH);  7

memcpy(refx.ref.ObjectType,"FLATFILE",sizeof(refx.ref.ObjectType)); 8

memset(refx.SrcName,' ',sizeof(refx.SrcName)+sizeof(refx.DestName));
memcpy(refx.SrcName,infile,strlen(infile));               9
memcpy(refx.DestName,outfile,strlen(outfile));

refx.ref.SrcNameLength = strlen(infile);                  10
refx.ref.SrcNameOffset = offsetof(MQRMHX,SrcName);

refx.ref.DestNameLength = strlen(outfile);
refx.ref.DestNameOffset = offsetof(MQRMHX,DestName);
```

Figure 57. Building a Reference Message

7 Since the file to be sent contains plain text, we specify MQFMT_STRING.

8 The object type must match the MsgData specification in the channel definition. The channel exit program amqsxrma displays that a message of the object type in the message does not match the object type in the channel definition.

9 Before we move the source and target filenames into the reference message we initialize the two fields with blanks. The filenames are not null-terminated.

10 At the end we store the length and offset of the filenames into the appropriate fields in the RMH.

6.3.1.4 Sending the Reference Message

Figure 58 on page 131 shows the statements to send the reference message and file to the queue manager QMGR2. Queue manager name and queue name are hard coded.

```
:
 od.ObjectType = MQOT_Q;                           1
 strncpy(od.ObjectName, "REFMSG", sizeof(od.ObjectName));
 strncpy(od.ObjectQMgrName, "QMGR2", sizeof(od.ObjectQMgrName));

 pmo.Options = MQPMO_FAIL_IF_QUIESCING;            2

 md.MsgType = MQMT_DATAGRAM;                       3
 memcpy(md.Format,MQFMT_REF_MSG_HEADER,(size_t)MQ_FORMAT_LENGTH);

MQPUT1(Hcon,                      4    /* connection handle           */
       &od,                            /* object descriptor for queue */
       &md,                            /* message descriptor          */
       &pmo,                           /* options                     */
       sizeof(refx),                   /* buffer length               */
       &refx,                          /* buffer                      */
       &CC, &Reason);                  /* completion and reason codes */

if (Reason != MQRC_NONE)
    printf("MQPUT1 ended with reason code %d\n", Reason);
```

Figure 58. Sending a Reference Message

1 Since we used the od structure for the MQINQ above, we have to change the object type from queue manager to queue (MQOT_Q). The object name, that is the name of the remote queue, is REFMSG. The target queue manager is QMGR2.

2 The put message option specifies that the MQPUT1 shall fail when it is issued while the queue manager is shutting down.

3 Since we do not expect a reply, we define the reference message as a datagram. We also tell the receiving queue manager that the (beginning of the) message is a reference message header. The header gets converted when MQGMO_CONVERT is specified in the MQGET call.

4 MQPUT1 is used to send the message. The buffer length is the length of the reference message only. It does not include the length of the file.

Note: The reference message as it appears in the transmission queue is described in 6.4, "The Reference Message" on page 135.

6.3.2 Writing the GETREF Program

This program receives the reference message and checks if the file is present. The program reads one message from the queue REFMSG and ends. A message is printed if the message is not a reference message. It contains the following functions:

- Connect to the queue manager QMGR2.
- Open the queue REFMSG.
- Get a message from the queue.
- If the message is a reference message:
  - Extract name and path of the file from the message.
  - Check if the file exists.
- Close the queue.
- Disconnect from the queue manager.

The complete listing is in E.2, "Source of GETREF" on page 241. The interesting parts of the program are described below.

1 Message and correlation ID are set to nulls so that the queue manager gets the first message from the queue.

2 In the get message options we direct the queue manager to convert the data in the reference message to be converted. The queue manager converts when the CCSID and Encoding values specified in the message descriptor (md) differ from the values in the message header of the message on the queue.

Note: CCSID and Encoding are input/output fields. If the conversion cannot be performed, the message data is returned not converted and CCSID and Encoding in the message descriptor (md) are set to the values for the not converted message.

3 The Encoding field identifies how numeric values are represented in the application message data. It is set to MQENC_NATIVE, the default for the programming language and machine. This field is used when MQGMO_CONVERT is specified.

4 This field is also used when MQGMO_CONVERT is specified. It applies to character data in the application message data. MQCCSI_Q_MGR causes the CCSID of the queue manager to be used.

```
MQGMO    gmo = {MQGMO_DEFAULT};  /* get message options          */
char     Buffer[1000];
MQLONG   DataLen;                /* length of message            */
:
 memcpy(md.MsgId,MQMI_NONE,sizeof(md.MsgId));      1
 memcpy(md.CorrelId,MQCI_NONE,sizeof(md.CorrelId));

 gmo.Options = MQGMO_WAIT +
              MQGMO_CONVERT +                     2
              MQGMO_ACCEPT_TRUNCATED_MSG;
 gmo.WaitInterval = 5000;          /*  5 seconds wait interval   */

 md.Encoding       = MQENC_NATIVE;                 3
 md.CodedCharSetId = MQCCSI_Q_MGR;                 4

 MQGET(Hcon,Hobj,              /* connection and queue handle   */
       &md,                    /* message descriptor            */
       &gmo,                   /* get options                   */
       sizeof(Buffer),         /* buffer size                   */
       &Buffer,                /* buffer address                */
       &DataLen,               /* data length (output)          */
       &CC,&Reason);
```

Figure 59. Get a Reference Message

```
char     Filename[256];
FILE    *File;                   /* file structure               */
MQRMH   *pMQRMH;                 /* Pointer to MQRMH structure   */
char    *pObjectName;            /* Object name                  */
:
 pMQRMH = (MQRMH*)&Buffer;      /* overlay MQRMH on MQGET buffer  */
:
 pObjectName = (char*)&Buffer + pMQRMH -> DestNameOffset;
 memset(Filename,0,sizeof(Filename));
 strncpy(Filename,pObjectName,
         ((size_t)(pMQRMH->DestNameLength) >= sizeof(Filename))
              ? (size_t)(sizeof(Filename) -1)
              : (size_t)(pMQRMH -> DestNameLength));
```

Figure 60. Extract Filename from Reference Message

Figure 60 shows how you can extract the target filename from the reference message. In our example, we put first the source filename and then the target filename after the header.

Note: The reference message is described, in detail, in 6.4, "The Reference Message" on page 135.

6.3.3 Compiling and Testing

Compile the programs using one of the compilers in Table 22.

Table 22. Commands to Compile Programs for Reference Message

Compiler	Command
Microsoft Visual C/C++	cl putref.c mqm.lib cl getref.c mqm.lib
IBM Visual Age C/C++	icc putref.c mqm.lib icc getref.c mqm.lib
CSet++ for AIX	xlc putref.c -l mqm -o putref xlc getref.c -l mqm -o getref

Make sure that you have created the objects in Table 23 on page 135. If not, create two queue managers and the objects with the following commands:

QMGR1
```
crtmqm /q /u SYSTEM.DEAD.LETTER.QUEUE QMGR1
strmqm
runmqsc < qmgr1.in > qmgr1.out
```

QMGR2
```
crtmqm /q /u SYSTEM.DEAD.LETTER.QUEUE QMGR2
strmqm QMGR2
runmqsc QMGR2 < qmgr2.in > qmgr2.out
```

Check for errors in the output files.

Copy the exit program amqsxrm.dll from the \mqm\tools\c\samples\bin directory into \mqm\exits.

To execute the programs, enter the commands below:

QMGR1
```
start runmqchi -q SYSTEM.CHANNEL.INITQ
start runmqlsr -t tcp -p 1414
putref
```

QMGR2
```
start runmqchi -q SYSTEM.CHANNEL.INITQ -m QMGR2
start runmqlsr -t tcp -p 1414 -m QMGR2
getref
```

Table 23. Objects for Reference Message

QMGR1	QMGR2
define qremote (REFMSG) + rname(REFMSG) rqmname(QMGR2) + xmitq(QMGR2) + replace	define ql(REFMSG) + replace
define qlocal (QMGR2) + usage(xmitq) + trigger trigtype(first) + initq(SYSTEM.CHANNEL.INITQ) + replace	define qlocal (QMGR1) + usage(xmitq) + trigger trigtype(first) + initq(SYSTEM.CHANNEL.INITQ) + replace
define chl (QMGR1.QMGR2) + chltype(sdr) + conname('127.0.0.1(1415)') + trptype(tcp) xmitq(QMGR2) + descr('send reference message') + msgexit('c:\mqm\exits\amqsxrm.dll(MsgExit)') + msgdata(FLATFILE) + replace	define chl(QMGR1.QMGR2) + chltype(rcvr) + trptype(tcp) + descr('receive reference message') + msgexit('c:\mqm\exits\amqsxrm.dll(MsgExit)') + msgdata(FLATFILE) + replace
alter qmgr chad(enabled)	alter qmgr chad(enabled) define chl (QMGR2.QMGR1) + chltype(sdr) + conname('127.0.0.1(1414)') + trptype(tcp) xmitq(QMGR1) + replace
File Name: qmgr1.in	**File Name:** qmgr2.in

6.4 The Reference Message

If you want to use reference messages in your own programs you will have to include a new structure called MQRMH, or the MQ Reference Message Header. The structure is shown in Figure 61 on page 136. You can find details concerning this structure in the *MQSeries Application Programming Reference*, SC33-1673.

Figure 62 on page 138 and Figure 63 on page 139 show a message that is put on the transmission queue (printed with AMQSBCG).

The first part is the transmission header (XQH). In the message, you recognize the name of the destination queue (REFMSG) and the destination queue manager (QMGR2).

The xmit header is followed by the message header (MD). AMQSBCG formatted the contents of the message descriptor at the beginning of the output. Note the format "MQHREF" in line x'80' in the header.

```
/*****************************************************************/
/*  MQRMH Structure -- Reference Message Header                  */
/*****************************************************************/

typedef struct tagMQRMH {
  MQCHAR4   StrucId;            /* Structure identifier */
  MQLONG    Version;            /* Structure version number */
  MQLONG    StrucLength;        /* Total length of MQRMH, including
                                   strings at end of fixed fields,
                                   but not the bulk data */
  MQLONG    Encoding;           /* Data encoding */
  MQLONG    CodedCharSetId;     /* Coded character set identifier */
  MQCHAR8   Format;             /* Format name */
  MQLONG    Flags;              /* Reference message flags */
  MQCHAR8   ObjectType;         /* Object type */
  MQBYTE24  ObjectInstanceId;   /* Object instance identifier */
  MQLONG    SrcEnvLength;       /* Length of source environment
                                   data */
  MQLONG    SrcEnvOffset;       /* Offset of source environment
                                   data */
  MQLONG    SrcNameLength;      /* Length of source object name */
  MQLONG    SrcNameOffset;      /* Offset of source object name */
  MQLONG    DestEnvLength;      /* Length of destination environment
                                   data */
  MQLONG    DestEnvOffset;      /* Offset of destination environment
                                   data */
  MQLONG    DestNameLength;     /* Length of destination object
                                   name */
  MQLONG    DestNameOffset;     /* Offset of destination object
                                     name */
  MQLONG    DataLogicalLength; /* Length of bulk data */
  MQLONG    DataLogicalOffset; /* Low offset of bulk data */
  MQLONG    DataLogicalOffset2;/* High offset of bulk data */
 } MQRMH;
```

Figure 61. Reference Message Header

The reference message header (RMH) starts at x′01AC′. Table 24 on page 137 shows the fields and their contents. The RMH is followed by the 256-byte fields that contain path and name of the source and target files.

Table 24. Reference Message Contents		
Length	**Name**	**Contents**
MQCHAR4	StrucId	″RMH ″
MQLONG	Version	1
MQLONG	StrucLength	x26C = 620
MQLONG	Encoding	x222 = 546
MQLONG	CCSID	x1B5 = 437
MQCHAR8	Format	″MQSTR ″
MQLONG	Flags	1 = MQRMHF_LAST
MQCHAR8	ObjectType	″FLATFILE″
MQBYTE24	ObjectInstanceId	nulls
MQLONG	SrcEnvLength	
MQLONG	SrcEnvOffset	
MQLONG	SrcNameLength	x0E = 14 is length of ″c:\test\dw.fil″
MQLONG	SrcNameOffset	x6C = 108 (RMH + 1)
MQLONG	DestEnvLength	
MQLONG	DestEnvOffset	
MQLONG	DestNameLength	x0F = 15 is length of ″c:\test\dw1.txt″
MQLONG	DestNameOffset	x16C = 364 (RMH + 1 + 256)
MQLONG	DataLogicalLength	
MQLONG	DataLogicalOffset	
MQLONG	DataLogicalOffset2	

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

  StrucId  : 'MD  '  Version : 2
  Report   : 0  MsgType : 8
  Expiry   : -1  Feedback : 0
  Encoding : 546  CodedCharSetId : 437
  Format : 'MQXMIT  '
  Priority : 0  Persistence : 0
  MsgId : X'414D5120514D4752312020202020202020A9C2C93523100000'
  CorrelId : X'414D5120514D4752312020202020202020A9C2C93513100000'
  BackoutCount : 0
  ReplyToQ       : '                                                '
  ReplyToQMgr    : 'QMGR1                                           '
  ** Identity Context
  UserIdentifier : 'wackerow    '
  AccountingToken :
  X'0131000000000000000000000000000000000000000000000000000000000000'
  ApplIdentityData : '                                '
  ** Origin Context
  PutApplType    : '7'
  PutApplName    : 'QMGR1                       '
  PutDate  : '19980806'    PutTime  : '20353201'
  ApplOriginData : '    '

  GroupId : X'000000000000000000000000000000000000000000000000'
  MsgSeqNumber   : '1'
  Offset         : '0'
  MsgFlags       : '0'
  OriginalLength : '-1'

****   Message      ****  length = 1048 bytes

-------->  Transmission Header (104 bytes)

00000000:  5851 4820 0100 0000 5245 464D 5347 2020 'XQH ....REFMSG  '
00000010:  2020 2020 2020 2020 2020 2020 2020 2020 '                '
00000020:  2020 2020 2020 2020 2020 2020 2020 2020 '                '
00000030:  2020 2020 2020 2020 514D 4752 3220 2020 '        QMGR2   '
00000040:  2020 2020 2020 2020 2020 2020 2020 2020 '                '
00000050:  2020 2020 2020 2020 2020 2020 2020 2020 '                '
00000060:  2020 2020 2020 2020 ------------------- '        MD  ....'
```

Figure 62. Reference Message (Part 1)

```
-------->  Message Header  (324 bytes)

00000060:  ------------------- 4D44 2020 0100 0000 '        MD  ....'
00000070:  0000 0000 0800 0000 FFFF FFFF 0000 0000 '............'
00000080:  2202 0000 B501 0000 4D51 4852 4546 2020 '"...Á...MQHREF  '
00000090:  0000 0000 0000 0000 414D 5120 514D 4752 '........AMQ QMGR'
000000A0:  3120 2020 2020 2020 A9C2 C935 1310 0000 '1       Ì5....'
000000B0:  0000 0000 0000 0000 0000 0000 0000 0000 '................'
000000C0:  0000 0000 0000 0000 0000 0000 2020 2020 '............    '
000000D0:  2020 2020 2020 2020 2020 2020 2020 2020 '                '
000000E0:  2020 2020 2020 2020 2020 2020 2020 2020 '                '
000000F0:  2020 2020 2020 2020 2020 2020 514D 4752 '            QMGR'
00000100:  3120 2020 2020 2020 2020 2020 2020 2020 '1               '
00000110:  2020 2020 2020 2020 2020 2020 2020 2020 '                '
00000120:  2020 2020 2020 2020 2020 2020 7761 636B '            wack'
00000130:  6572 6F77 2020 2020 0131 0000 0000 0000 'erow    .1......'
00000140:  0000 0000 0000 0000 0000 0000 0000 0000 '................'
00000150:  0000 0000 0000 0000 2020 2020 2020 2020 '........        '
00000160:  2020 2020 2020 2020 2020 2020 2020 2020 '                '
00000170:  2020 2020 2020 2020 0B00 0000 433A 5C74 '        ....C:\t'
00000180:  6573 745C 7075 7472 6566 2E65 7865 2020 'est\putref.exe  '
00000190:  2020 2020 2020 2020 3139 3938 3038 3036 '        19980806'
000001A0:  3230 3335 3332 3031 2020 2020 --------- '20353201    RMH '

-------->  Reference Message Header (108 bytes)

000001A0:  ----------------------------- 524D 4820 '20353201    RMH '
000001B0:  0100 0000 6C02 0000 2202 0000 B501 0000 '....l..."...Á...'
000001C0:  4D51 5354 5220 2020 0100 0000 464C 4154 'MQSTR   ....FLAT'
000001D0:  4649 4C45 0000 0000 0000 0000 0000 0000 'FILE............'
000001E0:  0000 0000 0000 0000 0000 0000 0000 0000 '................'
000001F0:  0000 0000 0E00 0000 6C00 0000 0000 0000 '........l.......'
00000200:  0000 0000 0F00 0000 6C01 0000 0000 0000 '........l.......'
00000210:  0000 0000 0000 0000 ------------------- '........c:\test\'

-------->  Reference Message Data (2 x 256 bytes)

00000210:  ------------------- 633A 5C74 6573 745C '........c:\test\'
00000220:  6477 2E66 696C 2020 2020 2020 2020 2020 'dw.fil          '
           all blanks
00000310:  2020 2020 2020 2020 633A 5C74 6573 745C '        c:\test\'
00000320:  6477 312E 7478 7420 2020 2020 2020 2020 'dw1.txt         '
           all blanks
00000410:  2020 2020 2020 2020                     '                '
```

Figure 63. Reference Message (Part 2)

Chapter 7. Distribution Lists

MQSeries Version 5 adds a distribution list feature. This means that instead of opening one queue to PUT a message, you can now open a list of queues. The queue manager will put messages on each of the queues in the list and will distribute the messages in an intelligent manner.

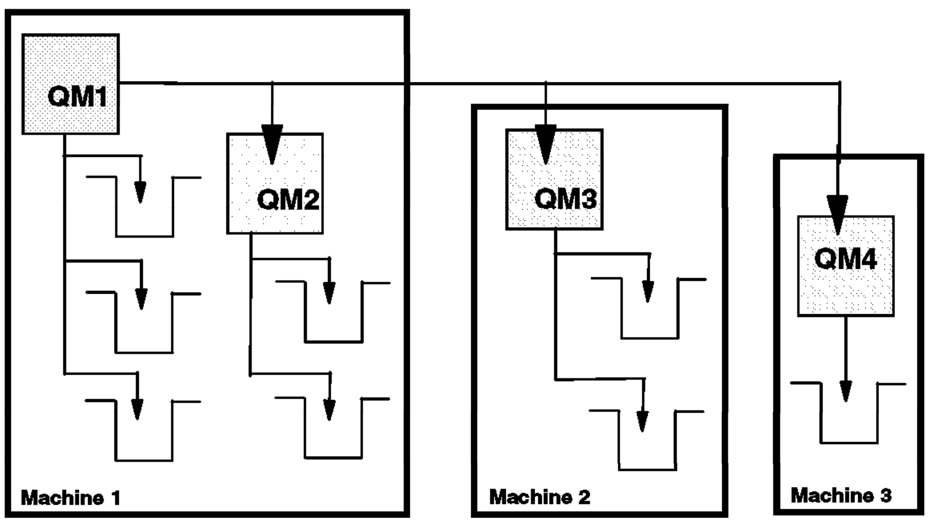


Figure 64. Distribution List

Looking at Figure 64, we see four queue managers that reside in three different machines. Each of the queue managers owns several queues. Let us assume that an application running in the first machine wants to put the same message in all of the queues in all three machines. The application can put a message to multiple destinations with a single MQPUT or MQPUT1.

The MQSeries distribution list facility provides a means where the application can optimize the performance of putting a message to multiple queues. The major performance benefit comes when the messages are put to multiple remote destinations of which two or more resolve to the same transmission queue. Under these conditions multiple logical messages can be compressed into a single physical message. The single physical message is then sent across the channel and expanded into multiple physical messages by the receiving channel. This is known as *late fan out*.

Distribution lists are dynamic. The lists are managed by the application.

7.1 Structures that Support Distribution Lists

To support distribution lists, three structures are provided. They are:

MQOR — The *MQ object record* is a structure that is used to specify a single destination queue in the form of a queue/queue manager pair. An array of these structures is called a *distribution list*. It is addressed via the Version 2 object descriptor.

MQRR — The *MQ response record* structure is used to receive the completion code and reason code resulting from the open or put operation for a single destination queue. By providing an array of these structures on the MQOPEN and MQPUT calls it is possible to determine the completion codes and reason codes for all queues in a distribution list. The array of these structures should have the same number of elements as the MQOR array. It is addressed via the Version 2 object descriptor and message options.

MQPMR — The *MQ put message record* structure is used to override certain properties in the message header. The array should contain as many elements as there are destinations. It is addressed via the Version 2 message descriptor or the Version 2 MQPMO. The MQPMR allows you to specify different values for each destination in a distribution list.

This stucture does not have a fixed layout. The fields in this structure are optional, and the presence or absence within each field is indicated by the flags in the *PutMsgRecFields* in the MQPMO. The *PutMsgRecField* can contain one or more values of:

1. MQPMRF_MSG_ID
2. MQPMRF_CORREL_ID
3. MQPMRF_GROUP_ID
4. MQPMRF_FEEDBACK
5. MQPMRF_ACCOUNTING_TOKEN

The application is expected to define its own PMR structure and then set the bits in the *PutMsgRecFields* to indicate which fields the stucture contains.

Figure 66 on page 143 through Figure 68 on page 144 show the new structures (in C) and Figure 65 on page 143 shows how they are tied together. The new fields in the MQOD and MQPMR are shown in Figure 69 on page 146 and Figure 70 on page 147.

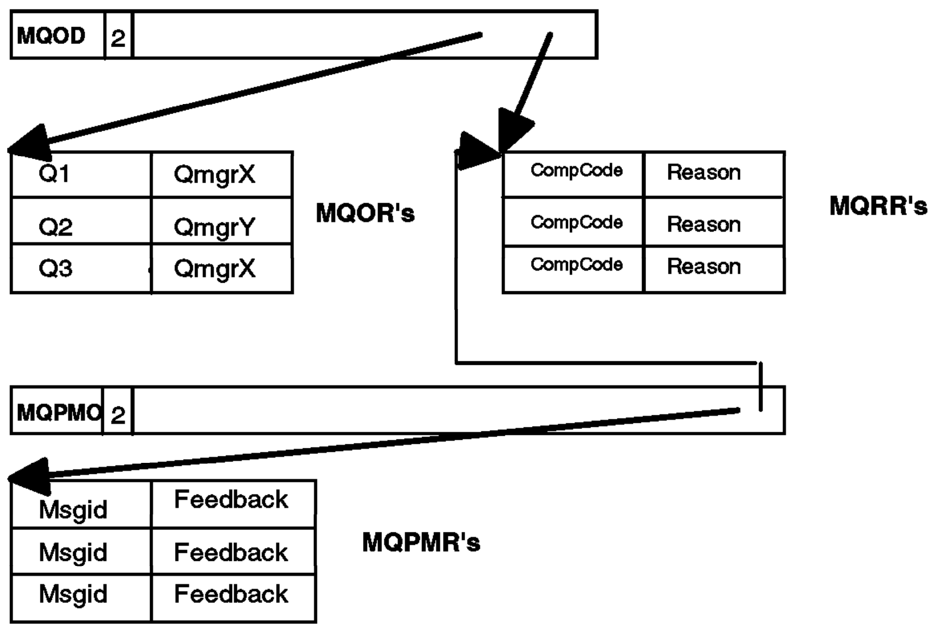


Figure 65. Stuctures for Distribution Lists

```
/*****************************************************************/
/*  MQOR Structure -- Object Record                              */
/*****************************************************************/
typedef struct tagMQOR {
   MQCHAR48  ObjectName;      /* Object name */
   MQCHAR48  ObjectQMgrName;  /* Object queue manager name */
 } MQOR;
typedef MQOR MQPOINTER PMQOR;
```

Figure 66. Object Record Structure MQOR

```
/*****************************************************************/
/*  MQRR Structure -- Response Record                            */
/*****************************************************************/
typedef struct tagMQRR {
   MQLONG  CompCode;  /* Completion code for queue */
   MQLONG  Reason;    /* Reason code for queue */
 } MQRR;
typedef MQRR MQPOINTER PMQRR;
```

Figure 67. Response Record Structure MQRR

```
/******************************************************************/
/*  MQPMR Structure -- Put Message Record                         */
/******************************************************************/
typedef struct tagMQPMR {
   MQBYTE24  MsgId;           /* Message ID                  */
   MQBYTE24  CorrelId         /* Correlation ID              */
   MQBYTE24  GroupId          /* Group ID                    */
   MQLONG    Feedback;        /* Feedback or reason code     */
   MQBYTE32  AccountingToken; /* Accounting token            */
 } MQPMR;
```

Figure 68. Sample Put Message Record Structure MQPMR

If you want to give each message a different message ID and feedback you would define this structure:

```
typedef struct tagMQPMR {
   MQBYTE24  MsgId;
   MQLONG    Feedback;
} MQPMR;
```

To tell the queue manager what the fields are in the structure specify in the put message options:

```
pmo.PutMsgRecFields = MQPRMF_MSG_ID | MQORMF_FEEDBACK;
```

7.2 MQI Extensions to Support Distribution Lists

This section describes what changes have been made to the APIs to support distribution lists. Programming examples are in 7.6, "Exercise 13: Distribution List" on page 150.

MQOPEN MQOPEN is extended to allow an array of queue/queue manager names (MQORs) to be passed and an array of completion/reason codes to be returned.

MQPUT MQPUT is extended to allow an array of message attributes (MQPMRs) to be passed and an array of reason codes to be returned.

MQPUT1 MQPUT1 is extended to allow an array of MQORs and an array of MQPMRs to be passed and an array of MQRRs to be returned.

MQCLOSE MQCLOSE does not allow an array of completion/reason codes to be returned. If one of the destinations fails to close then MQCLOSE will return one of the failing responses.

New fields have been added to the object descriptor MQOD and the put message options MQPMO. Both structures have a Version 2. The extensions to these structures allow the information in the MQOR, MQRR and MQPMR structures to be passed to the MQI. The structures can be addressed either by pointer or offset. Normally programs written in languages with good pointer support, such as C would use the pointer field. Languages with poor pointer support, such as COBOL would use the offset field. The offset is the offset from the start of the structure in which the offset field is defined.

Figure 69 on page 146 and Figure 70 on page 147 show the MQOD and MQPMO structures. The new fields are shown in **bold.** For the queue manager to recognize the new fields, the version number of the structures must be 2, for example:

```
pmo.Version = MQPMO_VERSION_2;
od.Version = MQOD_VERSION_2;
```

```
/****************************************************************/
/*  MQPMO Structure -- Put Message Options                      */
/****************************************************************/

typedef struct tagMQPMO {
   MQCHAR4   StrucId;          /* Structure identifier */
   MQLONG    Version;          /* Structure version number */
   MQLONG    Options;          /* Options that control the action
                                  of MQPUT or MQPUT1 */
   MQLONG    Timeout;          /* Reserved */
   MQHOBJ    Context;          /* Object handle of input queue */
   MQLONG    KnownDestCount;   /* Number of messages sent
                                  successfully to local queues */
   MQLONG    UnknownDestCount; /* Number of messages sent
                                  sucessfully to remote queues */
   MQLONG    InvalidDestCount; /* Number of messages that could
                                  not be sent */
   MQCHAR48  ResolvedQName;    /* Resolved name of destination
                                  queue */
   MQCHAR48  ResolvedQMgrName; /* Resolved name of destination
                                  queue manager */
   MQLONG    RecsPresent;      /* Number of put message records
                                  or response records present */
   MQLONG    PutMsgRecFields;  /* Flags indicating which MQPMR
                                  fields are present */
   MQLONG    PutMsgRecOffset;  /* Offset of first put message record
                                  from start of MQPMO */
   MQLONG    ResponseRecOffset;/* Offset of first response record
                                  from start of MQPMO */
   MQPTR     PutMsgRecPtr;     /* Address of first put message
                                  record */
   MQPTR     ResponseRecPtr;   /* Address of first response record */
 } MQPMO;
typedef MQPMO MQPOINTER PMQPMO;
```

Figure 69. Extensions to the Put Message Options MQPMO

```
/******************************************************************/
/*  MQOD Structure -- Object Descriptor                           */
/******************************************************************/
typedef struct tagMQOD {
   MQCHAR4   StrucId;          /* Structure identifier */
   MQLONG    Version;          /* Structure version number */
   MQLONG    ObjectType;       /* Object type */
   MQCHAR48  ObjectName;       /* Object name */
   MQCHAR48  ObjectQMgrName;   /* Object queue manager name */
   MQCHAR48  DynamicQName;     /* Dynamic queue name */
   MQCHAR12  AlternateUserId;  /* Alternate user identifier */
   MQLONG    RecsPresent;      /* Number of object records present */
   MQLONG    KnownDestCount;   /* Number of local queues opened
                                  successfully */
   MQLONG    UnknownDestCount; /* Number of remote queues opened
                                  successfully */
   MQLONG    InvalidDestCount; /* Number of queues that failed to
                                  open */
   MQLONG    ObjectRecOffset;  /* Offset of first object record
                                  from start of MQOD */
   MQLONG    ResponseRecOffset;/* Offset of first response record
                                  from start of MQOD */
   MQPTR     ObjectRecPtr;     /* Address of first object record */
   MQPTR     ResponseRecPtr;   /* Address of first response record */
 } MQOD;
typedef MQOD MQPOINTER PMQOD;
```

Figure 70. Extensions to the Object Descriptor MQOD

7.3 Error Handling

APIs that use distribution lists can have one of three results:

1. The request succeeds.

   `CompCode = MQCC_OK,  Reason = MQRC_OK`

2. The request is partially successful.

   `CompCode = MQCC_WARNING,  Reason = MQRC_MULTIPLE_REASONS`

3. The request fails.

   `CompCode = MQCC_FAILED,  Reason = MQRC_*`

The MQRR structure is optional on MQOPEN, MQPUT and MQPUT1. If the application wishes to handle errors then it should use MQRRs.

When an MQI verb operates against multiple destinations then it can fail on a per destination basis. The MQRR structure is provided to allow a list of completion/reason codes to be returned.

If the operation is successful for all the destinations in the list, or if the operations fails for the same reason for all the destinations in the list then a single completion/reason code is returned in the usual manner.

If the operation should work for some destinations in the list, but not for others, a warning with the reason MQRC_MULTIPLE_REASONS is returned. The MQRR array must be interrogated to determine if and why the operation failed for each destination. If the operation fails for all destinations, but not for the same reason, then the completion code is MQRC_FAILED and the reason code is MQRC_MULTIPLE_REASONS.

7.4 Late Fan Out

The example in Figure 71 on page 149 shows that an application puts a message to Q1, Q2 and Q3 using a single MQPUT to a distribution list. Two messages will be created:

- One on the transmission queue for QmgrX (XQX)
- One on the transmission queue for QmgrY (YQY)

The message in transmission queue XQX will be prepended by both an MQXQH and an MQDH. Since the message on transmission queue YQY is destined for only one queue, it is prepended by an MQXQH only.

The physical message on the transmission queue XQX consists of:

1. MQXQH
2. MQDH
3. MQOR array
4. MQPRM array
5. Message data

When a message is sent to multiple destinations using a distribution list the queue manager attempts to condense the messages into the minimum number of physical messages.

In order for two or more messages to be combined into a single physical message the following must be possible:

- The messages must resolve to the same xmit queue.

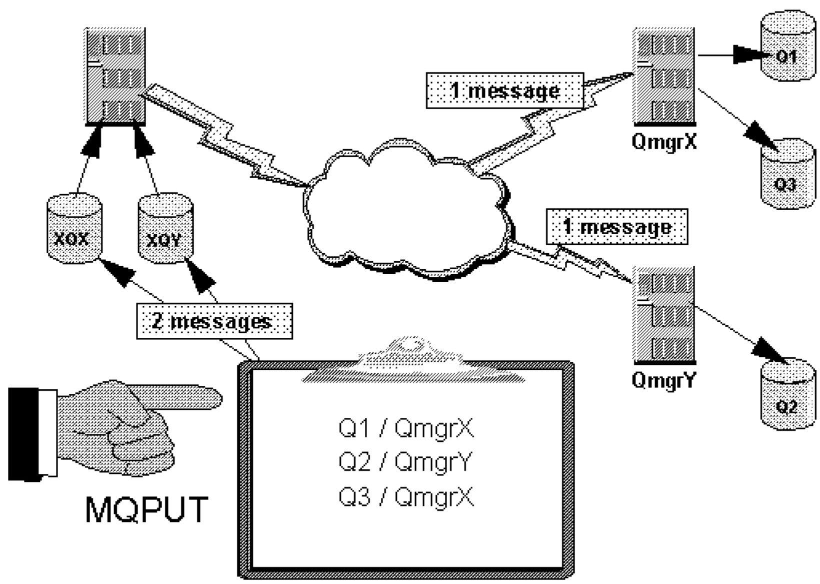


Figure 71. Late Fan Out

- The message attributes must be identical, such as priority and persistence (which may be specified ″as queue definition″).
- The size of the message must allow for the destribution list header and MQOR/MQPMR arrays to be prepended.

7.5 Configuration

Before a queue manager can send a distribution list format message over a channel it must be sure that the receiving queue manager understands this type of message. When the MCAs bind they exchange information which includes whether or not the receiving MCA supports distribution lists. The receiving MCA uses the queue manager′s DISTL attribute to determine if distribution list format messages are supported.

A queue manager passes messages to an MCA by putting them on a transmission queue. A queue manager will only put a distribution list format message on a transmission queue if that transmission queue has the DISTL attribute set to indicate that the partner to the MCA servicing that transmission queue supports distribution lists. When the MCAs bind, the sending MCA determines if the DISTL attribute is set correctly (resetting if necessary).

Attention

When a transmission queue is created the DISTL attribute defaults to NO, which means that any messages put before the channel first binds will not be in distribution list format.

If you reconfigure your system, for example, to reroute messages, then it is possible for messages in distribution list format to exist on a transmission queue which is being serviced by an MCA whose partner does not support such messages. In this case the sending MCA detects that it has read a distribution list format message that would not be understood by the partner MCA and it expands the message into multiple messages on the transmission queue. The ordering of the messages on the transmission queue is lost. There is also a performance penalty.

7.6 Exercise 13: Distribution List

The sample program DISTL puts messages from the standard input device and puts these messages to a list of message queues stored in a predefined text file called distlist.txt. The first line in the file is for the queue name and the second is for the queue manager name. With one MQPUT call, a message is put into all target queues defined in the distribution list (distlist.text). The distribution list is dynamic; you can add or remove queues.

For this application, you need the following:

1. The program DISTL that distributes the messages
2. The file distlist.txt that contains the distribution list in the form of queue/queue manager pairs
3. The file DISTL.TST that contains the queue definitions

7.6.1 Program Logic

- Read file distlist.txt that contains the distribution list and place the entries in an array.
- Put the distribution list into an MQOR structure.
- Connect to a queue manager.
- MQOPEN target queues for OUTPUT.
- Get messages from StdIn until NULL line is read.
  - Add each line to each target queue.
  - Each text line becomes a datagram message.

- The ″new line″ characters are removed.
- If a line is longer than 99 characters it is broken up into 99-character pieces. Each piece becomes the content of a datagram message.
- If the length of a line is a multiple of 99 plus 1 (for example, 199), the last piece will only contain a new-line character and so it will terminate the input.
- The program displays a message if there is a reason code other than MQRC_NONE.
- It stops if there is an MQI completion code of MQCC_FAILED.

- MQClose target queues.
- Disconnect from queue manager.
- Free up resources.

7.6.2 Setup for Distribution List Example

This example uses a text file called distlist.txt which contains names of queues and queue managers. Each name must be written in a separate line; the queue name must be first. Here is an example:

```
DISTQ.1
QMGR1
DISTQ.2
QMGR1
DISTQ.3
QMGR2
DISTQ.4
QMGR2
```

Of course, the queues specified in the above distribution list must be defined. In this example, we create two script files, one for QMGR1 and another for QMGR2:

Table 25. Queues for Distribution List

QMGR1	QMGR2
def ql(DISTQ.1) def ql(DISTQ.2)	def ql(DISTQ.3) def ql(DISTQ.4)

Define the queues with runmqsc using the two script files as input:

QMGR1 `runmqsc QMGR1 < distl1.mqs`

QMGR2 `runmqsc QMGR2 < distl2.mqs`

Note: The example program attempts to connect to the queue manager associated with the first queue specified in the file distlist.txt. If that fails then it tries to reach the next queue manager(s) it finds in the file.

7.6.3 Writing a Distribution List Program

This distribution list program DISTL.C is in Appendix F, "Distribution List Example" on page 245. The code necessary to put messages to a distribution list are outlined below.

```
static struct ObjectInfoType{
    char ObjName[40];
    char ObjQMgrName[40];
};
struct ObjectInfoType DistList[10];
:
 FILE *dl;                            /* distribution list file  */
 int i=0;                             /* number of entries       */

if (NULL == (dl = fopen("DistList.txt","r")))
   printf("\n Unable to open the data file!");
else
{
  while (!feof(dl)) {                /* read queue name         */
    fgets(DistList[i].ObjName,40,dl);
    DistList[i].ObjName[strlen(DistList[i].ObjName)-1] = '\0';
                                     /* read queue manager name */
    fgets(DistList[i].ObjQMgrName,40,dl);
    DistList[i].ObjName[strlen(DistList[i].ObjQmgrName)-1] = '\0';
    i += 1;
  }
  i -= 1;                         /* number of items in distlist */
  fclose(dl);
  }
```

Figure 72. Reading a Distribution List File

Figure 72 shows how the file distlist.txt is read and a list of queue/queue manager pairs is built and put into an array. The index variable "i" will contain the number of entries in the distribution list.

Limitations:

- Queue and queue manager names can be up to 40 characters long.
- The distribution list is limited to up to 10 entries.

In the example in Appendix F, “Distribution List Example” on page 245 this code is a procedure that is called with the statement:

```
NumQueues = ReadDistList();
```

Figure 73 shows how to copy a distribution list into an MQOR (object record) structure.

```
PMQRR   pRR=NULL;              /* Pointer to response records   */
pOR  = (PMQOR)malloc( NumQueues * sizeof(MQOR));
:
 for( Index = 0 ; Index < NumQueues ; Index ++) {
      strncpy( (pOR+Index)->ObjectName,
               DistList[Index].ObjName,
               (size_t)MQ_Q_NAME_LENGTH);
      strncpy( (pOR+Index)->ObjectQMgrName,
               DistList[Index].ObjQMgrName,
               (size_t)MQ_Q_MGR_NAME_LENGTH);
 }
```

Figure 73. Creating Object Records

Notes:

1. The field NumQueues contains the number of entries in the distribution list. The maximum is 10.
2. The structure DistList is defined as follows:

   ```
   static struct ObjectInfoType{
       char ObjName[40];
       char ObjQMgrName[40];
   };
   struct ObjectInfoType DistList[10];
   ```

3. At the end of the program, free the pOR structure with the statement:

   ```
   if (NULL != pOR)  free (pOR);
   ```

In this example we use both object records (MQOR) and response records (MQRR) for the return codes and put message records (MQPRM) to specify separate message and correlation IDs. You have to define and allocate those structures and free the memory before you end the program. These are new MQSeries Version 5 structures. This is done with the following statements:

```
PMQRR    pRR=NULL;              /* Pointer to response records   */
PMQOR    pOR=NULL;              /* Pointer to object records     */
:
pRR  = (PMQRR)malloc( NumQueues * sizeof(MQRR));
pOR  = (PMQOR)malloc( NumQueues * sizeof(MQOR));
pPMR = (pPutMsgRec)malloc( NumQueues * sizeof(PutMsgRec));
:
if (NULL != pOR)  free (pOR);
if (NULL != pRR)  free (pRR);
if (NULL != pPMR) free (pPMR);
```

How to define the put message records is shown in Figure 74.

The use of put message records (PMRs) allows some message attributes to be specified on a per destination basis. These attributes then override the values in the MD for a particular destination.

```
typedef struct
   {
     MQBYTE24 MsgId;              1
     MQBYTE24 CorrelId;
   } PutMsgRec, *pPutMsgRec;
pPutMsgRec pPMR=NULL;           /* Pointer to put msg records    */

MQLONG PutMsgRecFields=MQPMRF_MSG_ID | MQPMRF_CORREL_ID;    2
:
pPMR = (pPutMsgRec)malloc( NumQueues * sizeof(PutMsgRec)); 3
```

Figure 74. Creating Put Message Records

The function provided by this example program does not require the use of PMR′s but they are used by the program simply to demonstrate their use.

1 The program chooses to provide values for MsgId and CorrelId on a per destination basis.

2 The PutMsgRecFields in the PMO indicates what fields are in the array addressed by PutMsgRecPtr in the PMO. In our example we have provided the MsgId and CorrelId and so we must set the corresponding MQPMRF_ bits.

3 The program allocates memory for the PMRs for all destinations. Don′t forget to free them.

After connecting to a queue manager we have to open the queues defined in the distribution list. Figure 75 on page 155 shows the statements to do that.

```
:
od.Version = MQOD_VERSION_2 ;       /* must be version 2 MQOD      */
od.RecsPresent = NumQueues ;        /* number of object/resp recs  */
od.ObjectRecPtr = pOR;              /* address of object records   */
od.ResponseRecPtr = pRR ;           /* number of object records    */

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

MQOPEN(Hcon,                        /* connection handle           */
       &od,                         /* object descriptor for queue */
       O_options,                   /* open options                */
       &Hobj,                       /* object handle               */
       &OpenCode, &Reason);         /* return codes                */

if (OpenCode == MQCC_FAILED) {
    printf("Unable to open any queue for output\n");
}
else
if (Reason == MQRC_MULTIPLE_REASONS) {
    print_responses("MQOPEN", pRR, NumQueues, pOR);
}
else
if (Reason != MQRC_NONE) {
    printf("MQOPEN  returned CompCode=%ld, Reason=%ld\n",
               OpenCode, Reason);
}
```

Figure 75. Open Target Queues in Distribution List

The MQOPEN call returns the completion and reason codes and in the MQRR structure a reason code for each of the queues in the distribution list.

If the completion code is MQCC_FAILED then all of the destinations in the list failed to open.

If some destinations opened and others failed to open then the completion code will be set to MQCC_WARNING.

The reasons in the response records are only valid if the reason code returned is MQRC_MULTIPLE_REASONS. If any other reason is reported then opening all destination queues in the list completed or failed with the same reason.

You can print the reason codes in the MQRR structure with the routine shown in Figure 77 on page 157.

```
pmo.RecsPresent = NumQueues ;          /* number of queues         */
pmo.Version = MQPMO_VERSION_2 ;        /* V2 of put message options*/
pmo.PutMsgRecPtr = pPMR ;              /* PMR structure            */
pmo.PutMsgRecFields = PutMsgRecFields; /* fields in PMR            */
pmo.ResponseRecPtr = pRR ;             /* RR structure             */
:
                                      /* fill PMR structures       */
   for( Index = 0 ; Index < NumQueues ; Index ++) {     1
        memcpy( (pPMR+Index)->MsgId,MQMI_NONE,
                 sizeof((pPMR+Index)->MsgId));
        memcpy( (pPMR+Index)->CorrelId, MQCI_NONE,
                 sizeof((pPMR+Index)->CorrelId));
   }
   memcpy(md.Format, MQFMT_STRING,     /* character string format  */
          (size_t)MQ_FORMAT_LENGTH);                     2

MQPUT(Hcon, Hobj,                /* connection and object handles  */
      &md,                       /* message descriptor             */
      &pmo,                      /* default options (datagram)     */
      buflen,                    /* buffer length                  */
      buffer,                    /* message buffer                 */
      &CompCode, Reason);        /* completion amd reason codes    */
```

Figure 76. Put Message to Distribution List

Figure 76 shows the statements that put a message on multiple queues. The example in Appendix E, “Reference Message Example” on page 237 uses a loop to put several messages. The following code fragment, however, puts only one message. The message is in the variable ″buffer″, its length in ″buflen″.

1 The purpose of these instructions is to show how to put data into the PMR structure. You probably would use something other than MQMI_NONE and MQCI_NONE.

2 The messages in the example are character strings.

The function in Figure 77 on page 157 is usually called when a reason of MQRC_MULTIPLE_REASONS is received. The reasons relate to the queue at the equivalent ordinal position in the MQOR array.

```
static void print_responses( char * comment,
                             PMQRR  pRR,
                             MQLONG NumQueues,
                             PMQOR  pOR);
:
print_responses("MQCONN", pRR, Index, pOR);    /* call */
:
static void print_responses( char * comment,
                      PMQRR  pRR,
                      MQLONG NumQueues,
                      PMQOR  pOR)
{
 MQLONG Index;
 for( Index = 0 ; Index < NumQueues ; Index ++ ) {
   if( MQCC_OK != (pRR+Index)->CompCode ) {
    printf("%s for %.48s( %.48s) returned CompCode=%ld, Reason=%ld\n"
              comment,
              (pOR+Index)->ObjectName,
              (pOR+Index)->ObjectQMgrName,
              (pRR+Index)->CompCode,
              (pRR+Index)->Reason);
     }
 }
}
```

Figure 77. Display Response Record

7.6.4 Executing the Distribution List Example

First compile the program with one of the commands in Table 25 on page 151

Table 26. Commands to Compile DISTL.C	
Compiler	**Command**
Microsoft Visual C/C++	cl distl.c mqm.lib
IBM Visual Age C/C++	icc distl.c mqm.lib
CSet++ for AIX	xlc distl.c -l mqm -o distl

Then run it and check the result in two of our destination queues as follows. Call the program and enter three messages to be sent to all queues:

```
C:\dist1>distl
This is message 1
This is message 2
This is message 3


C:\dist1>amqsget DISTQ.4 QMGR2
Sample AMQSGET0 start
message <This is message 1>
message <This is message 2>
message <This is message 3>
no more messages
Sample AMQSGET0 end

C:\dist>amqsget DISTQ.2 QMGR1
Sample AMQSGET0 start
message <This is message 1>
message <This is message 2>
message <This is message 3>
no more messages
```

Chapter 8. FastPath Bindings

MQSeries Version 5 adds a new way for an application or a message channel agent to connect to the queue manager: the API MQCONNX. When you use MQCONNX you have the option of choosing either standard bindings or fast path bindings. Fast path bindings means that your application becomes a part of the queue manager; there is no boundary set up between your application and the queue manager. This means that the performance of a GET or PUT call is greatly enhanced. It also means that the integrity of the queue manager could be comprimised if your application is not well behaved. There are restrictions placed on a fast path application that are detailed in the *MQSeries Application Programming Guide*, SC33-0807.

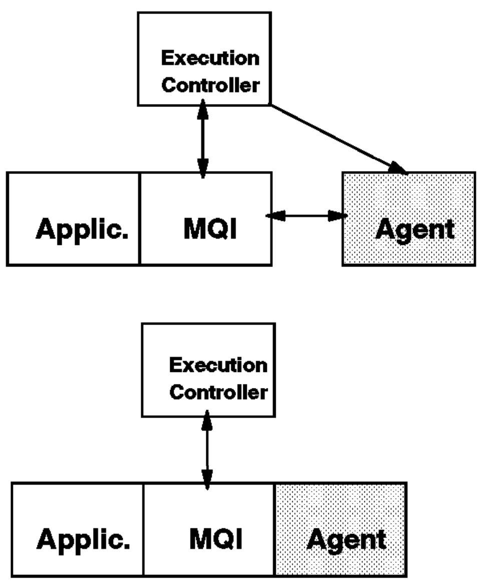


Figure 78. Standard and Fastpath Bindings

Figure 78 shows the local queue manager agent running as a separate thread and then running in the same process as the application. The first case is the default: standard binding. The application runs faster using fastpath bindings because the agent process does not need an interface to access the queue manager.

Note: Fastpath applications must be well behaved, that is, thoroughly tested. If you encounter a problem with your application run it with standard bindings (MQCONN instead of MQCONNX) before you call support.

If you use fastpath applications do not:

- Send asynchronous signals, such as sigkill
- Schedule timer interrupts
- Call ″abort()″
- Force stop a fastpath channel
- On Windows NT, never terminate from the task list

Otherwise, the queue manager may be left in an undefined state and should be recycled.

If a fastpath application ends without closing an object (MQCLOSE), the object will remain open until the queue manager is ended. If the application ends without an MQDISC, the queue manager cannot free resources.

The user of a fastpath application must be ″mqm″. On UNIX, uid and gid must be ″mqm″. Fastpath applications cannot be CICS applications.

End fastpath applications before ending the queue manager or shutting down the operating system. ″endmqm -p″ will attempt to kill fastpath applications which are still running.

8.1 Exercise 14: Using Fastpath Bindings

In this chapter we will see two programs that are identical except that one will use the MQCONN call with standard bindings and the other will use the MQCONNX call with fast path bindings to show the difference in performance as well as what you need to do in a program to use fast path bindings. When you use MQCONNX you will need to add a connection options structure to the call which specifies what type of connection you will use.

The two programs measure CPU time used by the MQPUT to put ″n″ non-persistent messages on the queue.

MCA′s can also use fastpath bindings. This is done on a global basis in the QM.INI file in the MQBINDTYPE parameter of the CHANNELS stanza. It is not possible to select fast path bindings for some channels and standard bindings for others. You should remember that if you are using fast path bindings on a channel that any exits you have will have the same integrity exposures mentioned above.

8.1.1 Program Logic

The logic of both progams is exactly the same, except that they differ in the way they connect to the queue manager:

- Take the number of messages to be put from the command line parameter.
- Connect to the queue manager using either MQCONN or MQCONNX.
- Open the queue INPUT.QUEUE for output.
- Get the CPU time.
- Put ″n″ fixed hard coded messages to the queue using MQPUT.
- Get the CPU time again.
- Calculate the actual time taken to put these n messages.
- Close the queue and disconnect from the queue manager.

Note: To remove the messages from the queue you can use amqsget. Alternatively, you may start runmqsc and use the command ″clear ql(INPUT.QUEUE)″.

8.1.2 The MQCNO Structure

The MQ connect option structure is shown below:

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

There are two connect options:

- MQCNO_STANDARD_BINDING or MQCNO_NONE
- MQCNO_FASTPATH_BINDING

8.1.3 Writing the Program

The source of the program CONNX.C is in Appendix G, “Fastpath Bindings Example” on page 255. The code for CONN.C is only on the diskette. The only difference between the two programs is the way they connect to the queue manager. Figure 79 on page 162 shows how to use CONNX. Some interesting fragments of the code are described below.

```
MQCNO    ConnectOpt;            /* Options to control the CONNX  */
:
   strcpy(ConnectOpt.StrucId, MQCNO_STRUC_ID);
   ConnectOpt.Version = MQCNO_VERSION_1;
   ConnectOpt.Options = MQCNO_FASTPATH_BINDING ;

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

Figure 79. Using MQCONNX

Figure 79 shows how the connect options are set to use fastpath bindings.

Figure 80 shows the code that measures the time for "n" puts to the queue INPUT.QUEUE.

```
double   Time1, Time2, Timediff;
:
   Time1 = (double) clock();
   Time1 = Time1/CLOCKS_PER_SEC;

   Insert code to put some messages

   Time2 = (double) clock();
   Time2 = Time2/CLOCKS_PER_SEC;

   Timediff = Time2-Time1;
   printf("\nThe elapsed time = %f seconds.", Timediff);
```

Figure 80. Measureing Elapsed Time

8.1.4 Comparing Standard and Fastpath Bindings

First, compile the two programs using one of the compilers shown in Table 27 on page 163.

In order to test the two programs we have to create a queue. The queue name INPUT.QUEUE is hard coded. You can use runmqsc and execute the command:

```
def ql(INPUT.QUEUE)
```

Table 27. Commands to Compile CONN.C and CONNX.C	
Compiler	**Command**
Microsoft Visual C/C++	cl conn.c mqm.lib cl connx mqm.lib
IBM Visual Age C/C++	icc conn.c mqm.lib icc connx.c mqm.lib
CSet++ for AIX	xlc conn.c -l mqm -o conn xlc connx.c -l mqm -o connx

To get accurate measurements make sure that the queue is cleared between runs.

Then execute CONN and CONNX, each with 100 and 1000 messages and compare the differences in CPU time.

```
C:\test>conn 100
Target queue is INPUT.QUEUE
The elapsed time = 0.310000 seconds

======> clear the queue

C:\test>connx 100
Target queue is INPUT.QUEUE

The elapsed time = 0.070000 seconds.
C:\redbook>connx 100
Target queue is INPUT.QUEUE

The elapsed time = 0.070000 seconds.
```

Table 28. Comparison between MQCONN and MQCONNX			
Program	**Message count**	**Elapsed time**	**Difference**
CONN	100	.31	
CONNX	100	.07	77%
CONN	1000	2.133	
CONNX	1000	.821	62%

As you can see the savings in elapsed time when processing non-persistent messages can be substantial. Your performance improvement will obviously vary from these depending on what your application, hardware, and software mix is. The important fact here is that using fastpath bindings is an option which can substantially affect performance. Once again, be well aware of the possible integrity issues presented in the *MQSeries Application Programming Guide*, SC33-0807 prior to using fastpath bindings.

Chapter 9. Multithreading

Multithreading is the paradigm of programming which exploits the existence of pockets of concurrency in applications. Threads or lightweight processes attempt to use the inherent parallelism of applications to provide better handling of the system. Threads are new concepts and are widely getting accepted as de facto standards used in medium-sized and large applications where the possibility of concurrency is greater. However, this is dependent on the nature of the application and the algorithms used. GUI-based applications, specifically use multithreading effectively to simulate parallelism and event-based programs find it useful to employ threads. The user finds better response time and does not really have to wait for one event to finish before getting to other tasks. In this scenario it is essential that a middle-tier product like MQSeries provide multithreading support which would mean that developers can build more responsive applications for users. Also, the advent of kernel multithreading support in almost all operating systems give applications low-level support for multithreaded programs.

MQSeries supports the use of its APIs in multithreaded applications. The MQSeries V5 release includes thread safe libraries that can be used to develop concurrent applications. This could increase the effectiveness and performance of your system utilizing all the advantages of multithreading. In fact, MQSeries internally uses multithreading to perform its functions. In this chapter we discuss the various considerations that we need to take while using MQSeries calls in multithreaded applications. We also discuss what cannot be achieved with multithreading.

9.1 MQSeries Support

MQSeries Version 5 is available on five platforms: AIX, Windows NT, HP-UX, Sun Solaris and OS/2. All libraries in these platforms are thread safe, so they are ready for multithreaded applications.

Let us now discuss the support per platform and look at what is provided and how to use it.

Table 29 on page 166 summarizes the support by platform in MQSeries Version 5 products.

Table 29. Thread Implementations by Platform	
Platform	**Thread Implementation**
AIX	DCE threads
Windows NT	Windows NT native threads
HP-UX	DCE threads
Sun Solaris	Solaris native threads and POSIX threads
OS/2	OS/2 native threads
Note: Third party POSIX libraries are available for Windows NT, and OS/2 comes with a POSIX library. They could be used, too.	

UNIX came out with the POSIX standards implementation. Windows NT and OS/2 have user-level thread implementation of the POSIX standard but these are third party software.

To make a multithreaded MQSeries application truly portable, we need to use the POSIX threads or Pthreads library with MQSeries calls. Strictly speaking, the underlying thread environment should really not affect the operation of MQSeries. But the field of multithreading is still very nascent. So it would be advisable to use products which are supported by IBM so that problems can be handed over to IBM Support for their advice.

We now look at the compilation steps used to build multithreaded applications. The step is different for each platform. This is due to the inconsistency in the implementation of threads across the platforms and the MQSeries model for that platform.

Table 30 summarizes the compilation steps for the various platforms.

Table 30 (Page 1 of 2). Compilation Steps for Multithreaded Applications	
Platform	**Compilation Steps**
AIX	Use cc_r compiler which automatically sets on multithreading switches and link with libmqm_r.a for server applications and libmqic _r.a for client applications. xlc_r appname.c -o appname -lmqm_r for server applications xlc_r appname.c -o appname -lmqic_r for client applications
Windows NT	Use Microsoft Visual C++ compiler and link with mqm.lib for server applications and mqic32.lib for client applications. cl appname.c /link mqm.lib for server applications cl appname.c /link mqic32.lib for client applications

Table 30 (Page 2 of 2). Compilation Steps for Multithreaded Applications	
Platform	**Compilation Steps**
HP-UX	Use the supported ANSI C compiler and link with libmqm_r.sl for server applications and libmqic_r.sl for client applications. cc -Aa -D_HPUX_SOURCE -o appname appname.c -lmqm_r for server applications cc -Aa -D_HPUX_SOURCE -o appname appname.c -lmqic_r for client applications
Solaris	Use Sun Workshop′s C compiler, use the -mt switch indicating to the compiler that you wish to create a multithreaded application. The MQSeries libraries to link to do not change. cc -o appname appname.c -mt -lmqm -lmqmcs -lmqmzse -lsocket -ldl -lnsl for server applications cc -o appname appname.c -mt -lmqic -lmqmcs -lmqmzse -lsocket -ldl -lnsl for client applications
OS/2	Use IBM Visual Age for C++ compiler. Use the switch /Gm+ to indicate a multithreaded application. The MQSeries libraries to link to do not change. icc /Gm+ appname.c mqm.lib for server applications icc /Gm+ appname.c mqic.lib for client applications
Note: In AIX and HP-UX, the multithreaded libraries of MQSeries are postfixed with an _r standing for re-entrant. This is from DCE terminology. MQSeries for Sun Solaris has only multithreaded libraries. So if you are not using threads and use MQI calls, you still need to exercise the -mt option for compilation. If you don′t do so you should get a core dump in MQCONN.	

9.2 The Scope of MQCONN

The connection to the queue manager in an application is through an HCONN data type. Any call made to MQSeries following an MQCONN should use the valid HCONN generated by the MQCONN call. This HCONN value represents the key to accessing MQSeries objects required for conducting transactions. So it is essential and worthwhile to have a precise knowledge of the scope of the HCONN value to enable us to use MQSeries in multithreaded applications.

Table 31 on page 168 summarizes the scope of the HCONN variable.

Table 31. Scope of MQCONN in Various Platforms	
Platform	**Scope for MQCONN**
UNIX (AIX, HP-UX and Solaris)	Thread
Windows NT	Thread
OS/2	Thread
Java	Application
Note: Java being platform independent merits a separate column for its uniqueness.	

The table leads us to the following conclusion: *The applications cannot use one HCONN across threads.* So each thread in an application which needs to talk to a queue manager object will have to perform an MQCONN and get a valid HCONN handle before progressing on its correspondence with the queue manager. HCONN used across threads will be rendered useless and the MQI call which uses another thread′s HCONN will return an MQRC_HCONN_ERROR.

In Java, due to the inherent multithreading nature of the Java Virtual Machine the code actually executes in different threads depending on invocation, usage and design. So the scope for the handle has been relaxed in Java and you can share handles across threads. This means that at initialization you could connect to the queue manager and use this handle in multiple threads concurrently. A typical design would be to use the Java applet′s init() function to achieve the connection to the queue manager and use individual threads to achieve the application objectives.

Note: MQSeries on UNIX systems cannot connect to different queue managers on different threads of an application. So an application can connect to only one queue manager at a given instance of execution. If an MQCONN is attempted while the application (any thread) is connected to a queue manager already, MQI returns MQRC_ANOTHER_Q_MGR_CONNECTED. However, during the life cycle of the application, it can connect to as many queue managers as it pleases.

9.3 Signals

MQI sets handlers for SIGSEGV, SIGBUS and SIGALRM during every MQI call. User handlers are suspended for the duration of the MQI call. So if the application wishes to set handlers for these signals, they should be set process wide before an MQI call is made. This will enable the possibility of MQSeries giving control to the application′s handler when possible. In the case of specific conditions like crashes MQSeries will terminate and may not allow cleanup of the application′s resources. In this situation the problem will have to be resolved with IBM Support.

Another point that should be kept in mind is the time of establishing the appropriate signal handlers. If a thread is doing the job of establishing the signal handler, there should not be any other threads issuing MQI calls. In such a situation, the MQSeries handler will be overridden and the handlers may pass these signals into the MQSeries code leading to unpredictable results depending on the status of the MQI call and the state of the various application threads.

If an MQSeries signal handler gets a signal for a thread that is not currently in MQSeries code, it attempts to find the handler established by the application (before it called MQSeries) and pass the signal to that handler. However, it may not be possible to pass all aspects of the call to the signal handler.

Signal handlers cannot make MQI calls. This could fail in two ways depending on other threads and the operating system restrictions:

- If another MQI function is active, MQRC_CALL_IN_PROGRESS is returned.
- If no other MQI function is active, it is likely to fail because of the operating system restrictions on which calls can be issued from within a handler.

Note: If a signal handler calls exit(), MQSeries backs out uncommitted messages under syncpoint as usual and closes any open queues.

9.4 Exercise 15: A Multithreaded Program

In this section, we present a C example of a simple multithreaded application for a UNIX (AIX) system.

UNIX platforms provide a similar interface to threads and for this reason the example should hold good for proprietary thread implementations. For this example, we use the POSIX threads library popularly known as Pthreads.

Notes:

1. DCE thread implementation came out of the ongoing POSIX standard which is not yet completed, so the interface is the same for both DCE and POSIX threads. For a detailed description of the differences refer to C programming books available in book stores.
2. Solaris native threads are slightly different in their interface. For example, pthread_create in the POSIX notation corresponds to

thr_create. The manuals suggest that the difference between the two implementations in terms of the interface only involves extra features.

The following UNIX example is the simplest one possible. Consider the scenario to send messages to two queues belonging to one queue manager. This is normally done with the following algorithm:

- Connect to the queue manager.
- Open queue 1.
- Put messages into queue 1.
- Close queue 1.
- Open queue 2.
- Put messages into queue 2.
- Close queue 2.
- Disconnect from the queue manager.

It might be possible that the application which serves queue 1 may be down and the application servicing queue 2 may be up and running. In this scenario it would make sense to issue the MQPUT to the second queue first. But the application will not want the overhead of finding out which applications are up and running (in whatever way). In this scenario, threads could be used to provide concurrent MQPUTs to the queues. The threaded application will resemble this algorithm:

- Start thread 1 to put messages in queue 1.
- Start thread 2 to put messages in queue 2.
- Wait for the threads to complete their operations or go ahead with other tasks.

Each thread would execute the following functions:

- Connect to the queue manager.
- Open queue.
- Put messages into queue.
- Close queue.
- Disconnect from the queue manager.
- End thread.

Concurrency of operations ensures that MQPUTs take place simultaneously. So the overall system′s performance comes up in the situation when one of the queue service providers is down. Normal operations where both service providers are up is not affected in any way.

The C program code is presented in the following files:

main.c
: Figure 82 on page 172

  This is the main driver function which calls threads for its operations.

mqput.c
: Figure 83 on page 173

  This is a slightly modified version of the MQSeries sample amqsput0.c used to:

  - Perform a connection to a queue manager (the default queue manager in this case).
  - Open the queue whose name is specified in the arguments to the function.
  - Put the messages.
  - Close the queue and disconnect from the queue manager.

globals.h
: Figure 81

  This is a header file containing a declaration of the functions and debug hooks.

```
#ifndef GLOBALS_H_
#define GLOBALS_H_

#ifdef _DEBUG
#define debug(a) printf(a)
#else
#define debug(a)
#endif

void * mqput(void *);
#endif
```

Figure 81. The Header File globals.h

```
#include <stdio.h>
#include <pthread.h>
#include "globals.h"

main()
{
        pthread_t thread1, thread2;
        pthread_attr_t thread1_attr, thread2_attr;
        char queue1[40]="One.Queue";
        char queue2[40]="Another.Queue";
        void * returnval;

        debug("In main\n");

        pthread_attr_init(&thread1_attr);
        pthread_attr_setdetachstate(&thread1_attr,PTHREAD_CREATE_UNDETACHED);
        pthread_attr_init(&thread2_attr);
        pthread_attr_setdetachstate(&thread2_attr,PTHREAD_CREATE_UNDETACHED);

        debug("Creating Thread 1\n");
        if(pthread_create(&thread1,&thread1_attr,mqput,(void *)queue2))
         {
          perror("pthread_create");
          exit(2);
          }
        debug("Creating Thread 2\n");
        if(pthread_create(&thread2,&thread2_attr,mqput,(void *)queue2))
         {
          perror("pthread_create");
          exit(2);
          }
        debug("Threads Created\n");

        debug("Wait for the Threads to complete\n");
        if (pthread_join(thread1,&returnval))
        {
        perror("pthread_join");
        exit(2);
        }
        printf("The Thread 1 returned with code : %d\n",(int)returnval);
        if (pthread_join(thread2,&returnval))
        {
        perror("pthread_join");
        exit(2);
        }
        printf("The Thread 2 returned with code : %d\n",(int)returnval);
        debug("Threads are joined\n");
        exit(0);
}
```

Figure 82. The Driver Function main.c

```
#include <stdio.h>
#include <pthread.h>
#include <stdlib.h>
#include <string.h>
#include <cmqc.h>            /* includes for MQI */
#include "globals.h"         /* thread globals   */

void*  mqput(void *queue)
{
   /*   Declare MQI structures needed                          */
   MQOD     od = {MQOD_DEFAULT};    /* Object descriptor       */
   MQMD     md = {MQMD_DEFAULT};    /* Message descriptor      */
   MQPMO   pmo = {MQPMO_DEFAULT};   /* Put message options     */

   MQHCONN  Hcon;                   /* Connection handle       */
   MQHOBJ   Hobj;                   /* Object handle           */
   MQLONG   O_options;              /* MQOPEN options          */
   MQLONG   C_options;              /* MQCLOSE options         */
   MQLONG   CompCode;               /* Completion code         */
   MQLONG   OpenCode;               /* MQOPEN completion code  */
   MQLONG   Reason;                 /* Reason code             */
   MQLONG   CReason;                /* Reason code for MQCONN  */
   MQLONG   buflen;                 /* Buffer length           */
   char     buffer[100];            /* Message buffer          */
   char QMName[40]="";              /* Default queue manager   */

   debug("MQPUT start\n");

   MQCONN(QMName, &Hcon, &CompCode, &CReason);   /* Connect    */
   if (CompCode == MQCC_FAILED)                  /* Failed ?   */
   {
     printf("MQCONN ended with reason code %ld\n", CReason);
     pthread_exit( (void **) CReason);
   }
   /*   Use parameter as the name of the target queue          */
   strncpy(od.ObjectName, (char *)queue, (size_t)MQ_Q_NAME_LENGTH);
   printf("target queue is %s\n", od.ObjectName);

   /*   Open the target message queue for output               */
   O_options = MQOO_OUTPUT + MQOO_FAIL_IF_QUIESCING;
   MQOPEN(Hcon, &od, O_options, &Hobj, &OpenCode, &Reason);
   if (Reason != MQRC_NONE)                      /* Failed ?   */
       printf("MQOPEN ended with reason code %ld\n", Reason);
   if (OpenCode == MQCC_FAILED)
   {
     printf("unable to open queue for output\n");
     pthread_exit((void**) Reason);
   }
   CompCode = OpenCode;  /* use MQOPEN result for initial test */
```

Figure 83 (Part 1 of 2). Function which Constitutes a Thread: mqput.c

```
/*   Put message into the required Queue                        */
   memcpy(md.Format, MQFMT_STRING, (size_t)MQ_FORMAT_LENGTH);

   memcpy(md.MsgId, MQMI_NONE, sizeof(md.MsgId) );

   memcpy(md.CorrelId, MQCI_NONE, sizeof(md.CorrelId) );

   strcpy(buffer,"This message should make sense to the other application");
   buflen=strlen(buffer);

   MQPUT(Hcon, Hobj, &md, &pmo, buflen, buffer, &CompCode, &Reason);

   if (Reason != MQRC_NONE)                    /* Failed ?    */
   {
      printf("MQPUT ended with reason code %ld\n", Reason);
   }

   /*   Close the target queue (if it was opened)                  */
   if (OpenCode != MQCC_FAILED)
   {
     C_options = 0;

     MQCLOSE(Hcon, &Hobj, C_options, &CompCode, &Reason);

     if (Reason != MQRC_NONE)
     {
       printf("MQCLOSE ended with reason code %ld\n", Reason);
     }
   }

   /*   Disconnect from MQM if not already connected               */
   if (CReason != MQRC_ALREADY_CONNECTED)
   {
     MQDISC(&Hcon, &CompCode, &Reason);

     if (Reason != MQRC_NONE)
     {
       printf("MQDISC ended with reason code %ld\n", Reason);
     }
   }

   debug("Sample MQPUT end\n");
   pthread_exit( (void **) 0);
 }
```

Figure 83 (Part 2 of 2). Function which Constitutes a Thread: mqput.c

Appendix A. Example Using One XA Resource

```
/****************************************************************************/
/*                                                                          */
/* Program name: AMQSXAS0.SQC                                          */
/*               CHANGED 10/09/1997 M.Schuette, IBM Germany                 */
/* Description:  Sample SQC program for MQ coordinating XA-compliant database */
/*               managers.                                                  */
/*                                                                          */
/* Changes :    - Unlimited wait for new messages                           */
/*              - New command message BYE ends the programm                 */
/*              - If invalid commands are sent user may delete the messages */
/*              - User is asked whether to commit successful changes or not */
/*                                                                          */
/* Parameters:  - Name of the message queue (required)                      */
/*              - Queue manager name (optional)                             */
/*                                                                          */
/****************************************************************************/
/* Includes                                                                 */
/****************************************************************************/
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <ctype.h>

#include <sqlca.h>                          /* SQL Communication Area     */
#include <sqlenv.h>                         /* SQL DB environment API     */
#include "util.h"                           /* SQL error checking utils   */

#include <cmqc.h>                           /* MQI                        */

/****************************************************************************/
/* Defines                                                                  */
/****************************************************************************/

#define OK      0                           /* define OK as zero          */
#define NOT_OK  1                           /* define NOT_OK as one       */

/****************************************************************************/
/* Define macro for checking if SQL call resulted in an error               */
/****************************************************************************/

#define  CHECKERR(CE_STR)   rc = check_error(CE_STR, &sqlca)
```

Exercise 3: amqsxas0.sqc

```
/******************************************************************************/
/* Define and declare an SQLCA (SQL Communication Area) structure          */
/******************************************************************************/
EXEC SQL INCLUDE SQLCA;
                                                /****************************/
int main(int argc, char *argv[])                /* ----- START OF MAIN ----- */
                                                /****************************/
{
   /***************************************************************************/
   /* MQI structures                                                          */
   /***************************************************************************/
   MQOD od = {MQOD_DEFAULT};                    /* object descriptor        */
   MQMD md = {MQMD_DEFAULT};                    /* message descriptor       */
   MQGMO gmo = {MQGMO_DEFAULT};                 /* get message options      */
   MQBO bo = {MQBO_DEFAULT};                    /* begin options            */
   /***************************************************************************/
   /* MQI variables                                                           */
   /***************************************************************************/
   MQLONG rc=OK;                                /* return code              */
   MQHCONN hCon;                                /* handle to connection     */
   MQHOBJ hObj;                                 /* handle to object         */
   char QMName[50]="";                          /* default QM name          */
   MQLONG options;                              /* options                  */
   MQLONG reason;                               /* reason code              */
   MQLONG connReason;                           /* MQCONN reason code       */
   MQLONG compCode;                             /* completion code          */
   MQLONG openCompCode;                         /* MQOPEN completion code   */
   char msgBuf[100];                            /* message buffer           */
   MQLONG msgBufLen;                            /* message buffer length    */
   MQLONG msgLen;                               /* message length received  */
   /***************************************************************************/
   /* Other variables                                                         */
   /***************************************************************************/
   char *pStr;                                  /* ptr to string            */
   int gotMsg;                                  /* got message from queue   */
   int committedUpdate;                         /* committed update         */
   long balanceChange;                          /* balance change           */
   int invCmd;                                  /* Invalid Message          */
   int bye;                                     /* bye command received, quit*/
   char ch[10];                                 /* for keycheck             */
   /***************************************************************************/
   /* SQL host declarations                                                   */
   /***************************************************************************/
   EXEC SQL BEGIN DECLARE SECTION;
   char name[40];                               /* name                     */
   long account;                                /* account number           */
   long balance;                                /* balance                  */
   EXEC SQL END DECLARE SECTION;
```

```
/**************************************************************************/
/* First check we have been given correct arguments                      */
/**************************************************************************/
if (argc != 2 && argc != 3){
   printf("Input is: %s 'queue name' 'queue manager name'.\n"
          "Note the queue manager name is optional\n", argv[0]);
   exit(99);
} /* endif */

if (argc == 3)                              /* use the queue manager     */
   strcpy(QMName, argv[2]);                 /* name supplied             */

/**************************************************************************/
/* Declare the cursor for locking of reads from database                 */
/**************************************************************************/
EXEC SQL DECLARE cur CURSOR FOR
         SELECT Name, Balance
         FROM MQBankT
         WHERE Account = :account
         FOR UPDATE OF Balance;

CHECKERR ("DECLARE CURSOR");
if (rc != OK)
   exit(0);                                 /* no point in going on      */

/**************************************************************************/
/* Connect to queue manager                                              */
/**************************************************************************/
MQCONN(QMName, &hCon, &compCode, &connReason);

if (compCode == MQCC_FAILED){
  printf("MQCONN ended with reason code %li\n", connReason);
  exit((int) connReason);
}

/**************************************************************************/
/* Use input parameter as the name of the target queue                   */
/**************************************************************************/
strncpy(od.ObjectName, argv[1], (size_t) MQ_Q_NAME_LENGTH);
printf("Target queue is %s\n", od.ObjectName);
```

Exercise 3: amqsxas0.sqc

```
/**************************************************************************/
/* Open the target message queue for input                                */
/**************************************************************************/
options = MQOO_INPUT_AS_Q_DEF + MQOO_FAIL_IF_QUIESCING;

MQOPEN(hCon, &od, options, &hObj, &openCompCode, &reason);

if (reason != MQRC_NONE)
   printf("MQOPEN ended with reason code %li\n", reason);

if (openCompCode == MQCC_FAILED){
   printf("Unable to open queue for input\n");
   rc = openCompCode;                       /* stop further action       */
} /* endif */

/**************************************************************************/
/* Get messages from the message queue, loop until there is a failure     */
/**************************************************************************/
while (compCode != MQCC_FAILED && rc == OK){

   /***********************************************************************/
   /* Set flags so that we can back out if something goes wrong and not   */
   /* lose the message.                                                   */
   /***********************************************************************/
   gotMsg = 0;                              /* set flag to FALSE         */
   committedUpdate = 0;                     /* set flag to FALSE         */
   invCmd=0;                  /* clear flag for invalid command received */
   bye=0;                  /*  BYE - Command received; let's quit progr.  */
   /***********************************************************************/
   /* Start a unit of work                                                */
   /***********************************************************************/
   MQBEGIN (hCon, &bo, &compCode, &reason);

   if (reason == MQRC_NONE){
      printf("Unit of work started\n"); }
   else{
      printf("MQBEGIN ended with reason code %li\n", reason);
      /********************************************************************/
      /* If we get a reason code and only a warning on the compCode, there */
      /* is something wrong with one or more of the resource managers so   */
      /* stop looping and sort it out, whatever the compCode.             */
      /********************************************************************/
      rc = NOT_OK;                          /* stop looping              */
   }

   if (compCode == MQCC_FAILED)
      printf("Unable to start a unit of work\n");
```

```
/*********************************************************************/
/* Get message off queue                                             */
/*********************************************************************/
if (rc == OK){

   /******************************************************************/
   /* In order to read the messages in sequence, MsgId and CorrelID    */
   /* must have the default value. MQGET sets them to the values for  */
   /* the message it returns, so re-initialise them before every call */
   /******************************************************************/
   memcpy(md.MsgId, MQMI_NONE, sizeof(md.MsgId));
   memcpy(md.CorrelId, MQCI_NONE, sizeof(md.CorrelId));

   /******************************************************************/
   /* Setup for the MQGET                                             */
   /******************************************************************/
   msgBufLen = sizeof(msgBuf) - 1;
   gmo.Options = MQGMO_WAIT + MQGMO_CONVERT + MQGMO_SYNCPOINT;
   gmo.WaitInterval = MQWI_UNLIMITED;

   MQGET(hCon, hObj, &md, &gmo, msgBufLen, msgBuf, &msgLen,
         &compCode, &reason);

   if (reason != MQRC_NONE){
      if (reason == MQRC_NO_MSG_AVAILABLE)
         printf("No more messages\n");
      else{
         printf("MQGET ended with reason code %li\n", reason);

         if (reason == MQRC_TRUNCATED_MSG_FAILED)
            compCode = MQCC_FAILED;        /* stop looping             */
      }
   }
   else gotMsg = 1;                  /* set flag to TRUE, message read */*/

} /* endif */
```

Exercise 3: amqsxas0.sqc

```
/*********************************************************************/
/* Process the message received                                      */
/*********************************************************************/
if (compCode != MQCC_FAILED && rc == OK){

   /******************************************************************/
   /* Check if message has been backed out more than 2 times         */
   /******************************************************************/
   if (md.BackoutCount > 2) {
      printf("The following message has been backed out %li times.\n",
              md.BackoutCount);
      printf("%s\n",msgBuf);
      rc = NOT_OK;                 /* Bypass database update */
      invCmd = 1;                  /* Ask whether to delete message */
   }

   msgBuf[msgLen] = '\0';                    /* add string terminator    */
   pStr = strstr(msgBuf, "UPDATE Balance change=");
   if (pStr != NULL){
      pStr += sizeof("UPDATE Balance change=") -1;
      sscanf(pStr, "%li", &balanceChange);
   }
   else{
      pStr = strstr(msgBuf, "BYE");
      if (strstr(msgBuf, "BYE") != NULL){
          printf("BYE-Message received...");
          bye=1;
      }
      else
          printf("Invalid Command received: %s\n", msgBuf);
      invCmd=1;
      rc = NOT_OK;                           /* stop looping anyway      */
   } /* endif */

   if (rc == OK){
      pStr = strstr(msgBuf, "Account=");
      if (pStr != NULL){
         pStr += sizeof("Account=") -1;
         sscanf(pStr, "%li", &account);
      }
      else{
         printf("Invalid command received: %s\n", msgBuf);
         invCmd=1;
         rc = NOT_OK;                        /* stop looping             */
      }
   } /* endif */
```

```
   /****************************************************************/
   /* Get details from database                                    */
   /****************************************************************/
   if (rc == OK){
      EXEC SQL OPEN cur;
      CHECKERR ("OPEN CURSOR");
   }
   if (rc == OK){
      EXEC SQL FETCH cur INTO :name, :balance;
      CHECKERR ("FETCH");
   }

   /****************************************************************/
   /* Update the bank balance                                      */
   /****************************************************************/
   if (rc == OK){
      balance += balanceChange;           /* alter balance            */
      EXEC SQL UPDATE MQBankT SET Balance = :balance
                        WHERE CURRENT OF cur;
      CHECKERR ("UPDATE MQBankT");

      if (rc == OK){
         printf("Account No %li Balance updated from %li to %li %s\n",
                account, balance - balanceChange, balance, name);
         /*************************************************************/
         /* We are going to commit the update so even if something goes */
         /* wrong now, the message has been used so don't back out.     */
         /*************************************************************/
         printf("Do you want to commit this Update [Yes|No] ?\n");
         gets(ch);
         if (ch[0]=='Y'||ch[0]=='y'){
            committedUpdate = 1;             /* set flag to TRUE         */
            /**********************************************************/
            /* Note: the cursor will be implicitly closed by the MQCMIT.   */
            /**********************************************************/
            MQCMIT(hCon, &compCode, &reason);
            if (reason == MQRC_NONE)
               printf("Unit of work successfully completed\n");
            else{
               printf("MQCMIT ended with reason code %li completion code "
                      "%li\n", reason, compCode);
               rc = NOT_OK;                  /* stop looping             */
            }
         } /* endif (Y || y) */
      } /* endif rc == OK */
   } /* endif rc == OK -> update balance */
} /* endif process message received */
```

Exercise 3: amqsxas0.sqc

```
/*********************************************************************/
/* If we got the message but something went wrong, back out so that we  */
/* don't lose the message (valid but not committed, invalid and BYE)    */
/*********************************************************************/
if (gotMsg && !committedUpdate){
   MQBACK(hCon, &compCode, &reason);
   if (reason == MQRC_NONE)
      printf("MQBACK successfully issued\n");
   else
      printf("MQBACK ended with reason code %li\n", reason);
} /* endif */

/*********************************************************************/
/* If reason for backout was an invalid command or a BYE message ask    */
/* whether the message shall remain in the queue or not.                */
/*********************************************************************/
if (invCmd){
   printf("Remove BACKOUTed-Message [Yes]No] ? /n");
   gets(ch);
   if (ch[0]=='Y'||ch[0]=='y'){
      gmo.Options = MQGMO_WAIT + MQGMO_CONVERT + MQGMO_NO_SYNCPOINT;
      MQGET(hCon, hObj, &md, &gmo, msgBufLen, msgBuf, &msgLen,
            &compCode, &reason);
      if (reason != MQRC_NONE){
         if (reason == MQRC_NO_MSG_AVAILABLE)
            printf("\nMessage NOT FOUND !\n");
         else{
            printf("\nRemove failed, MQGET ended with reason code %li\n", reason);
            if (reason == MQRC_TRUNCATED_MSG_FAILED)
              compCode = MQCC_FAILED;        /* stop looping */
         }
      }
      else{
        printf("\nMessage successful removed.\n");
        compCode = MQCC_OK;
        /* if (!bye) rc=OK; moved */
      }
   } /* endif Y || y */
   if (!bye) rc=OK;  /* Only a BYE message ends the program */
} /* endif  invCmd */
} /* endwhile */
```

```
  /**************************************************************************/
  /* Close queue if opened                                                  */
  /**************************************************************************/
  if (openCompCode != MQCC_FAILED){
     options = 0;                              /* no close options          */
     MQCLOSE(hCon, &hObj, options, &compCode, &reason);
     if (reason != MQRC_NONE)
        printf("MQCLOSE ended with reason code %li\n", reason);
  } /* endif */

  /**************************************************************************/
  /* Disconnect from queue manager if not already connected                 */
  /**************************************************************************/
  if (connReason != MQRC_ALREADY_CONNECTED){
     MQDISC(&hCon, &compCode, &reason);
     if (reason != MQRC_NONE)
        printf("MQDISC ended with reason code %li\n", reason);
  }
  return 0;
}   /* end of MAIN */
```

Exercise 3: amqsxas0.sqc

Appendix B. Example Using Two XA Resources

This example consists of three programs and make files for AIX, IBM C and Microsoft C compilers. The programs are:

- The main program AMQSXAG0.C
- The program AMQSXAB0.SQL to access MQBankDB
- The program AMQSXAF0.SQL to access MQFeeDB

Note: Since the two sql files are identical with the exception of the database name we include only amqsxab0.sql in this appendix.

B.1 Main Program AMQSXAG0.C (Modified)

```
static char *sccsid = "@(#) samples/c/xatm/amqsxag0.c, tranmgr;
/******************************************************************************/
/*                                                                            */
/* Program name: AMQSXAG0                                                     */
/* Description:  Sample C program for MQ coordinating XA-compliant database   */
/*               managers.                                                    */
/* Statement:    Licensed Materials - Property of IBM                         */
/*               (C) Copyright IBM Corp. 1997                                 */
/*Modified by M.Schuette                                                      */
/******************************************************************************/
/* Includes                                                                   */
/******************************************************************************/
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <ctype.h>
#include <cmqc.h>                              /* MQI                         */
/******************************************************************************/
/* Defines                                                                    */
/******************************************************************************/
#define OK      0                              /* define OK as zero           */
#define NOT_OK  1                              /* define NOT_OK as one        */
/******************************************************************************/
/* Function prototypes for SQL routines in AMQSXAB0.SQC and AMQSXAF0.SQC      */
/******************************************************************************/
int DeclareMQBankDBCursor(void);
int DeclareMQFeeDBCursor(void);
int ConnectToMQBankDB(void);
int ConnectToMQFeeDB(void);
int GetMQBankTBDetails(char *name, long account, long *balance,
```

Exercise 4: amqsxag0.c

```
                       long *transactions);
int GetMQFeeTBDetails(long account, long *feeDue, long *tranFee,
                      long *transactions);
int UpdateMQBankTBBalance(long balance, long transactions);
int UpdateMQFeeTBFeeDue(long feeDue, long transactions);


                                                 /*****************************/
int main(int argc, char *argv[])                 /* ----- START OF MAIN ----- */
                                                 /*****************************/
{
   /**********************************************************************/
   /* MQI structures                                                     */
   /**********************************************************************/
   MQOD od = {MQOD_DEFAULT};                     /* object descriptor        */
   MQMD md = {MQMD_DEFAULT};                     /* message descriptor       */
   MQGMO gmo = {MQGMO_DEFAULT};                  /* get message options      */
   MQBO bo = {MQBO_DEFAULT};                     /* begin options            */

   /**********************************************************************/
   /* MQI variables                                                      */
   /**********************************************************************/
   MQLONG rc=OK;                                 /* return code              */
   MQHCONN hCon;                                 /* handle to connection     */
   MQHOBJ hObj;                                  /* handle to object         */
   char QMName[50]="";                           /* default QM name          */
   MQLONG options;                               /* options                  */
   MQLONG reason;                                /* reason code              */
   MQLONG connReason;                            /* MQCONN reason code       */
   MQLONG compCode;                              /* completion code          */
   MQLONG openCompCode;                          /* MQOPEN completion code   */
   char msgBuf[100];                             /* message buffer           */
   MQLONG msgBufLen;                             /* message buffer length    */
   MQLONG msgLen;                                /* message length received  */

   /**********************************************************************/
   /* Other variables                                                    */
   /**********************************************************************/
   char *pStr;                                   /* ptr to string            */
   long balanceChange;                           /* balance change           */
   char name[40];                                /* name                     */
   long account;                                 /* account number           */
   long balance;                                 /* balance                  */
   long transactions;                            /* transactions             */
   long temp;                                    /* temporary variable       */
   long feeDue;                                  /* fee due                  */
   long tranFee;                                 /* transaction fee          */
   int gotMsg;                                   /* got message from queue   */
```

```
int committedUpdate;                          /* committed update        */
int invCmd;                                   /* Invalid Message         */
int bye;                                      /* bye command received; quit*/
char ch[10];                                  /* for keycheck            */

/************************************************************************/
/* First check we have been given correct arguments                     */
/************************************************************************/
if (argc != 2 && argc != 3)
{
   printf("Input is: %s 'queue name' 'queue manager name'.\n"
          "Note the queue manager name is optional\n", argv[0]);
   exit(99);
} /* endif */

if (argc == 3)                                /* use the queue manager   */
   strcpy(QMName, argv[2]);                   /* name supplied           */

/************************************************************************/
/* Connect to queue manager                                             */
/************************************************************************/
MQCONN(QMName, &hCon, &compCode, &connReason);

if (compCode == MQCC_FAILED) {
  printf("MQCONN ended with reason code %li\n", connReason);
  exit((int) connReason);
} /* endif */
/************************************************************************/
/* Use input parameter as the name of the target queue                  */
/************************************************************************/
strncpy(od.ObjectName, argv[1], (size_t) MQ_Q_NAME_LENGTH);
printf("Target queue is %s\n", od.ObjectName);

/************************************************************************/
/* Open the target message queue for input                              */
/************************************************************************/
options = MQOO_INPUT_AS_Q_DEF + MQOO_FAIL_IF_QUIESCING;

MQOPEN(hCon, &od, options, &hObj, &openCompCode, &reason);

if (reason != MQRC_NONE)
   printf("MQOPEN ended with reason code %li\n", reason);
if (openCompCode == MQCC_FAILED) {
   printf("Unable to open queue for input\n");
   rc = openCompCode;                         /* stop further action     */
} /* endif */
```

Exercise 4: amqsxag0.c

```
/**************************************************************************/
/* Declare the cursors for locking of reads from databases               */
/**************************************************************************/
if (rc == OK)
   rc = DeclareMQBankDBCursor();
if (rc == OK)
   rc = DeclareMQFeeDBCursor();

/**************************************************************************/
/* Get messages from the message queue, loop until there is a failure    */
/**************************************************************************/
while (compCode != MQCC_FAILED && rc == OK)
{
   /***********************************************************************/
   /* Set flags so that we can back out if something goes wrong and not   */
   /* lose the message.                                                   */
   /***********************************************************************/
   gotMsg = 0;                           /* set flag to FALSE             */
   committedUpdate = 0;                  /* set flag to FALSE             */
   invCmd=0;                             /* set flag to FALSE             */

   /***********************************************************************/
   /* Start a unit of work                                                */
   /***********************************************************************/
   MQBEGIN (hCon, &bo, &compCode, &reason);

   if (reason == MQRC_NONE)
   {
      printf("Unit of work started\n");
   }
   else
   {
      printf("MQBEGIN ended with reason code %li\n", reason);
      /********************************************************************/
      /* If we get a reason code and only a warning on the compCode, there */
      /* is something wrong with one or more of the resource managers so   */
      /* stop looping and sort it out, whatever the compCode.              */
      /********************************************************************/
      rc = NOT_OK;                       /* stop looping                  */
   } /* endif */

   if (compCode == MQCC_FAILED)
      printf("Unable to start a unit of work\n");
```

```
/*********************************************************************/
/* Get message off queue                                            */
/*********************************************************************/
if (rc == OK)
{
   /****************************************************************/
   /* In order to read the messages in sequence, MsgId and CorrelID    */
   /* must have the default value. MQGET sets them to the values for   */
   /* the message it returns, so re-initialise them before every call  */
   /****************************************************************/
   memcpy(md.MsgId, MQMI_NONE, sizeof(md.MsgId));
   memcpy(md.CorrelId, MQCI_NONE, sizeof(md.CorrelId));
   /*********************************************************************/
   /* Set up some things for the MQGET                                  */
   /*********************************************************************/
   msgBufLen = sizeof(msgBuf) - 1;
   gmo.Options = MQGMO_WAIT + MQGMO_CONVERT + MQGMO_SYNCPOINT;
   gmo.WaitInterval = MQWI_UNLIMITED;        /* old=15 sec limit for waiting  */

   MQGET(hCon, hObj, &md, &gmo, msgBufLen, msgBuf, &msgLen,
         &compCode, &reason);

   if (reason != MQRC_NONE)
   {
      if (reason == MQRC_NO_MSG_AVAILABLE)
      {
         printf("No more messages\n");
      }
      else
      {
         printf("MQGET ended with reason code %li\n", reason);

         if (reason == MQRC_TRUNCATED_MSG_FAILED)
            compCode = MQCC_FAILED;        /* stop looping              */
      } /* endif */
   }
   else
   {
      gotMsg = 1;                          /* set flag to TRUE          */
   } /* endif */
} /* endif */
```

Exercise 4: amqsxag0.c

```
/*******************************************************************/
/* Process the message received                                    */
/*******************************************************************/
if (compCode != MQCC_FAILED && rc == OK){
   msgBuf[msgLen] = '\0';                /* add string terminator     */
   pStr = strstr(msgBuf, "UPDATE Balance change=");
   if (pStr != NULL){
      pStr += sizeof("UPDATE Balance change=") -1;
      sscanf(pStr, "%li", &balanceChange);
   }else{
      pStr = strstr(msgBuf, "BYE");
      if (strstr(msgBuf, "BYE") != NULL){
          printf("BYE-Message received...");
          bye=1;
      }else{
          printf("Invalid Command received: %s\n", msgBuf);
       }
      invCmd=1;
      rc = NOT_OK;                /* stop looping anyway  */
   } /* endif */
   if (rc == OK){
      pStr = strstr(msgBuf, "Account=");
      if (pStr != NULL){
         pStr += sizeof("Account=") -1;
         sscanf(pStr, "%li", &account);
      }else{
         printf("Invalid command received: %s\n", msgBuf);
         invCmd=1;
         rc = NOT_OK;                    /* stop looping              */
      } /* endif */
   } /* endif */

   /*****************************************************************/
   /* Note only actively connected to one database at a time        */
   /*****************************************************************/
   /*****************************************************************/
   /* Get details from databases                                    */
   /*****************************************************************/
   if (rc == OK)
      rc = ConnectToMQBankDB();
   if (rc == OK)
      rc = GetMQBankTBDetails(name, account, &balance, &transactions);

   if (rc == OK)
      rc = ConnectToMQFeeDB();
   if (rc == OK)
      rc = GetMQFeeTBDetails(account, &feeDue, &tranFee, &temp);
```

```
if (rc == OK)
{
   /****************************************************************/
   /* The number of transactions to the two databases should be    */
   /* identical, stop if not.                                      */
   /****************************************************************/
   if (temp != transactions)
   {
      printf("Databases are out of step !\n");
      rc = NOT_OK;                     /* stop looping              */
   } /* endif */
} /* endif */

/*******************************************************************/
/* Update the bank balance                                         */
/*******************************************************************/
if (rc == OK)
{
   transactions++;                     /* bump no of transactions   */
   balance += balanceChange;           /* alter balance             */
   feeDue += tranFee;                  /* alter fee due             */

   rc = UpdateMQFeeTBFeeDue(feeDue, transactions);

   if (rc == OK)                       /* must now connect back to  */
      rc = ConnectToMQBankDB();        /* other database            */
   if (rc == OK)
      rc = UpdateMQBankTBBalance(balance, transactions);

   if (rc == OK)
   {
      printf("Account No %li Balance updated from %li to %li %s\n",
             account, balance - balanceChange, balance, name);
      printf("Fee Due updated from %li to %li\n",
             feeDue - tranFee, feeDue);

      /*************************************************************/
      /* We are going to commit the update so even if something goes */
      /* wrong now, the message has been used so don't back out.   */
      /*************************************************************/
      committedUpdate = 1;             /* set flag to TRUE          */
```

Exercise 4: amqsxag0.c

```
          /*************************************************************/
          /* Note: the cursor will be implicitly closed by the MQCMIT.   */
          /*************************************************************/
          MQCMIT(hCon, &compCode, &reason);

          if (reason == MQRC_NONE)
          {
             printf("Unit of work successfully completed\n");
          }
          else
          {
             printf("MQCMIT ended with reason code %li completion code "
                    "%li\n", reason, compCode);
             rc = NOT_OK;                   /* stop looping              */
          } /* endif */
       } /* endif */
    } /* endif */
 } /* endif */

 /**********************************************************************/
 /* If we got the message but something went wrong, back out so that we  */
 /* don't lose the message.                                              */
 /**********************************************************************/
 if (gotMsg && !committedUpdate)
 {
    MQBACK(hCon, &compCode, &reason);

    if (reason == MQRC_NONE)
       printf("MQBACK successfully issued\n");
    else
       printf("MQBACK ended with reason code %li\n", reason);
 } /* endif */

 if (invCmd){ /* if the reason for backout was a wrong message, we may clear it */
    printf("Remove BACKOUTed-Message [Yes]No] ?");
    gets(ch);
    if (ch[0]=='Y']]ch[0]=='y'){
        /* Delete the message, anyway. */
        /* therfore, a simple get will do it, options are : */
        gmo.Options = MQGMO_WAIT + MQGMO_CONVERT + MQGMO_NO_SYNCPOINT;
        MQGET(hCon, hObj, &md, &gmo, msgBufLen, msgBuf, &msgLen,
              &compCode, &reason);
        if (reason != MQRC_NONE){
           if (reason == MQRC_NO_MSG_AVAILABLE){
              printf("\nMessage NOT FOUND !\n");
         }else{
            printf("\nRemove failed, MQGET ended with reason code %li\n", reason);
            if (reason == MQRC_TRUNCATED_MSG_FAILED)
```

```
                compCode = MQCC_FAILED;        /* stop looping */
             } /* endif */
         }else{
            printf("\nMessage successful removed.\n");
            compCode=MQCC_OK;
            if (!bye) rc=OK;
         } /* endif */
       } /* endif */
    } /* endif */
 } /* endwhile */

 /**************************************************************************/
 /* Close queue if opened                                                  */
 /**************************************************************************/
 if (openCompCode != MQCC_FAILED)
 {
    options = 0;                              /* no close options          */

    MQCLOSE(hCon, &hObj, options, &compCode, &reason);

    if (reason != MQRC_NONE)
       printf("MQCLOSE ended with reason code %li\n", reason);
 } /* endif */

 /**************************************************************************/
 /* Disconnect from queue manager if not already connected                 */
 /**************************************************************************/
 if (connReason != MQRC_ALREADY_CONNECTED)
 {
    MQDISC(&hCon, &compCode, &reason);

    if (reason != MQRC_NONE)
       printf("MQDISC ended with reason code %li\n", reason);
 } /* endif */

 return 0;
                                              /****************************/
}                                             /* ------ END OF MAIN ------ */
                                              /****************************/
```

B.2 AMQSXAB0.SQC Source Code

```
static char *sccsid = "@(#) samples/c/xatm/amqsxab0.sqc, tranmgr, p000, p000-L970806;
/******************************************************************************/
/*                                                                            */
/* Program name: AMQSXAG0                                                     */
/* Description:  Sample C program for MQ coordinating XA-compliant database   */
/*               managers.                                                    */
/* Statement:    Licensed Materials - Property of IBM                         */
/*               (C) Copyright IBM Corp. 1997                                 */
/*                                                                            */
/* Module Name: AMQSXAB0.SQC                                                  */
/* Description: Functions to access MQBankTB table in MQBankDB database       */
/* Function:    These functions provide access to MQBankTB table in MQBankDB  */
/*              database, they are called from AMQSXAG0.C                     */
/*                                                                            */
/******************************************************************************/
/* Includes                                                                   */
/******************************************************************************/
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <ctype.h>
#include <sqlca.h>                               /* SQL Communication Area    */
#include <sqlenv.h>                              /* SQL DB environment API    */
#include "util.h"                                /* SQL error checking utils  */

/******************************************************************************/
/* Defines                                                                    */
/******************************************************************************/
#define OK      0                                /* define OK as zero         */

/******************************************************************************/
/* Define macro for checking if SQL call resulted in an error                 */
/******************************************************************************/
#define  CHECKERR(CE_STR)   rc = check_error(CE_STR, &sqlca)

/******************************************************************************/
/* Define and declare an SQLCA (SQL Communication Area) structure             */
/******************************************************************************/
EXEC SQL INCLUDE SQLCA;
```

```
/****************************************************************************/
/* SQL host declarations                                                    */
/****************************************************************************/
EXEC SQL BEGIN DECLARE SECTION;
static char hName[40];                          /* name                     */
static long hAccount;                           /* account number           */
static long hBalance;                           /* balance                  */
static long hTransactions;                      /* transactions             */
EXEC SQL END DECLARE SECTION;

/****************************************************************************/
/*   Function: DeclareMQBankDBCursor                                        */
/*   Description: Declare MQBankDB cursor                                   */
/*                                                                          */
/*   Input Parameters:  none                                                */
/*   Output Parameters: None                                                */
/*   Returns:           int   rc          - return code from SQL command    */
/****************************************************************************/
int DeclareMQBankDBCursor(void)
{
   long rc=OK;                                  /* return code              */
   /*************************************************************************/
   /* Declare the cursor for locking of reads from database                 */
   /*************************************************************************/
   EXEC SQL DECLARE curBank CURSOR FOR
            SELECT Name, Balance, Transactions
            FROM MQBankTB
            WHERE Account = :hAccount
            FOR UPDATE OF Balance, Transactions;
   CHECKERR ("DECLARE CURSOR");
   return(rc);
}
/****************************************************************************/
/*   Function: ConnectToMQBankDB                                            */
/*   Description: Connect to MQBankDB database                              */
/*                                                                          */
/*   Input Parameters:  none                                                */
/*   Output Parameters: None                                                */
/*   Returns:           int   rc          - return code from SQL command    */
/****************************************************************************/
int ConnectToMQBankDB(void)
{
   long rc=OK;                                  /* return code              */
   EXEC SQL CONNECT TO MQBankDB;
   CHECKERR ("CONNECT TO MQBankDB");
   return(rc);
}
```

Exercise 4: amqsxab0.sqc

```
/****************************************************************************/
/*   Function: GetMQBankTBDetails                                           */
/*   Description: Get MQBankTB table details for supplied account number    */
/*                                                                          */
/*   Input Parameters:  long  account      - Account                        */
/*   Output Parameters: char *name         - Name                           */
/*                      long *balance      - Balance                        */
/*                      long *transactions - Transactions                   */
/*   Returns:           int   rc           - return code from SQL command   */
/****************************************************************************/
int GetMQBankTBDetails(char *name, long account, long *balance,
                       long *transactions)
{
   long rc=OK;                                  /* return code              */

   /*************************************************************************/
   /* Copy calling function variable to SQL host variable                   */
   /*************************************************************************/
   hAccount = account;

   EXEC SQL OPEN curBank;
   CHECKERR ("OPEN CURSOR");

   if (rc == OK)
   {
      EXEC SQL FETCH curBank INTO :hName, :hBalance, :hTransactions;
      CHECKERR ("FETCH");
   } /* endif */

   /*************************************************************************/
   /* Copy SQL host variables to calling function variables                 */
   /*************************************************************************/
   if (rc == OK)
   {
      strcpy(name, hName);
      *balance = hBalance;
      *transactions = hTransactions;
   } /* endif */

   return(rc);
}
```

```
/****************************************************************************/
/*   Function: UpdateMQBankTBBalance                                        */
/*   Description: Update MQBankTB table Balance for the current cursor      */
/*                                                                          */
/*   Input Parameters:  long  balance      - Balance                        */
/*                      long  transactions - Transactions                   */
/*   Output Parameters: none                                                */
/*   Returns:           int   rc           - return code from SQL command   */
/****************************************************************************/
int UpdateMQBankTBBalance(long balance, long transactions)
{
   long rc=OK;                                 /* return code              */

   /*************************************************************************/
   /* Copy calling function variables to SQL host variables                 */
   /*************************************************************************/
   hBalance = balance;
   hTransactions = transactions;

   EXEC SQL UPDATE MQBankTB SET Balance = :hBalance,
                               Transactions = :hTransactions
                           WHERE CURRENT OF curBank;
   CHECKERR ("UPDATE MQBankTB");

   return(rc);
}
```

B.3 Make Files for IBM Compiler

ibmmake.bat

```
rem -------------------------------------------------------------
rem - Build-File for C (/ C++) Programs w. IBM Vis.C++ Compiler -
rem -        (W) 10/08/1997 M.Schuette, IBM                    -
rem - Builds Progr. w. (DB/2) embedded SQL and MQSeries Calls  -
rem - For use with IBM VA/C++ 3.5+, DB/2 V2.1.2+, MQSeries V5+    -
rem -------------------------------------------------------------
rem Usage: ibmmake <prog_name> <db_name> <addlibs>

db2 connect to %2
db2 prep %1.sqc bindfile
db2 bind %1.bnd
db2 connect reset

rem Compile and link the program.
rem ????? To build a C++ program, change the source file extension to '.cxx'
rem and include the -Tp option.
rem Include other libraries at the end of the link-command !

icc /Gm /Ti- %1.c /C
ilink  %1.obj util.obj db2api.lib mqm.lib %3
```

ibmmake2.bat

```
rem -------------------------------------------------------------
rem - Build-File for C (/ C++) Programs w. IBM Vis.C++ Compiler -
rem -        (W) 10/08/1997 M.Schuette, IBM  /  Dieter          -
rem - Builds Progr. w. (DB/2) embedded SQL and MQSeries Calls  -
rem - For use with IBM VA/C++ 3.5+, DB/2 V2.1.2+, MQSeries V5+    -
rem -------------------------------------------------------------
rem Usage: ibmmake <prog_name> <db_name> <addlibs>

rem db2 connect to %2
rem db2 prep %1.sqc bindfile
rem db2 bind %1.bnd
rem db2 connect reset

rem Compile and link the program.
rem ????? To build a C++ program, change the source file extension to '.cxx'
rem and include the -Tp option.
rem Include other libraries at the end of the link-command !

icc /Gm /Ti- %1.c /C
ilink %1.obj util.obj db2api.lib mqm.lib amqsxab0.obj amqsxaf0.obj
```

B.4 Make Files for Microsoft Compiler

msmake.bat

```
rem -----------------------------------------------------------
rem - Build-File for C (/ C++) Programs w. MS Vis.C++ Compiler -
rem -         (W) 10/08/1997 M.Schuette, IBM                   -
rem - Builds Progr. w. (DB/2) embedded SQL and MQSeries Calls  -
rem - For use with MS VC++ 2.0+, DB/2 V2.1.2+, MQSeries V5+    -
rem -----------------------------------------------------------
rem Usage: msmake <prog_name> <db_name> <addlibs>

db2 connect to %2
db2 prep %1.sqc bindfile
db2 bind %1.bnd
db2 connect reset

rem Compile and link the program.
rem ????? To build a C++ program, change the source file extension to '.cxx'
rem and include the -Tp option.
rem Include other libraries at the end of the link-command !

cl -Z7 -Od -c -W2 -D_X86_=1 -DWIN32 -I%DB2PATH%\include %1.c
link -debug:full -debugtype:cv -out:%1.exe %1.obj util.obj db2api.lib mqm.lib %3
```

msmake2.bat

```
rem -----------------------------------------------------------
rem - Build-File for C (/ C++) Programs w. MS Vis.C++ Compiler -
rem -         (W) 10/08/1997 M.Schuette, IBM                   -
rem - Build-File without DB/2-Preprocessing / Binding          -
rem -----------------------------------------------------------
rem Usage: msmake <prog_name> <addobj1> <addobj2>

rem Compile and link the program.
rem ????? To build a C++ program, change the source file extension of %1 to '.cxx'
rem and include the -Tp option.
rem Include other libraries at the end of the link-command !

cl -Z7 -Od -c -W2 -D_X86_=1 -DWIN32 -I%DB2PATH%\include %1.c
link -debug:full -debugtype:cv -out:%1.exe %1.obj util.obj db2api.lib mqm.lib %2 %3
```

B.5 Make Files for AIX

Shell File amqsxag0.sh

```
#--------------------------------------------------------------------------#
#                                                                          #
#  AIX MQSeries Link DB2 Application program                               #
#  Make file for connect DB2 DataBase                                      #
#  First call db2 to preCompiler .sqc                                      #
#  Second call makefile named is amqsxas0.mak to generate executable       #
#  module                                                                  #
#                                                                          #
#--------------------------------------------------------------------------#
#
#
echo Connect to DB2 DataBase MQBANKDB
#
#
db2 connect to MQBANKDB
db2 prep amqsxab0.sqc bindfile
db2 bind amqsxab0.bnd
db2 connect reset

db2 connect to MQFEEDB
db2 prep amqsxaf0.sqc bindfile
db2 bind amqsxaf0.bnd
db2 connect reset
#
echo Call make -f amqsxag0.mak
make -f amqsxag0.mak
```

Make File amqsxag0.mak

```
#*************************************************************************
#*                                                                       *
#*  amqsxag0.mak: Source file generated by the Class Compiler            *
#*                11/29/95     20:39:48 language = C                     *
#*                                                                       *
#*************************************************************************
.SUFFIXES:
.SUFFIXES: .o .c

CC = xlc

OBJS= amqsxag0.o amqsxab0.o amqsxaf0.o  util.o
CFLAGS = -g -c  -I/usr/lpp/db2_05_00/samples/c /usr/lpp/mqm/inc
```

```
#CFLAGS = -g -c  -I/usr/lpp/mqm/inc
#CFLAGS = -g -c -Dsigned= -Dvolatile= -D_Optlink -I. -M
#LFLAGS = -L. -lXm -lXt -lX11 -L/usr/lpp/mq3t/lib -lbmqapic -e LibMain -bM:SRE
#---------------------------------------------------------------------
# MQM Library file and seraching path
#---------------------------------------------------------------------
MQMLIBS=-l mqm
MQMLIBPATH=-L /usr/lpp/mqm/lib
DB2LIBS=-l db2
DB2LIBPATH=-L /usr/lpp/db2_05_00/lib

#HEADERS = /usr/lpp/mqm/inc /usr/lpp/db2_05_00/samples/c
#HEADERDB2 = /usr/lpp/db2_05_00/samples/c

.c.o:
	$(CC) $(CFLAGS) $<

all: amqsxag0


amqsxag0.o: amqsxag0.c\
	$(HEADERS)

amqsxab0.o: amqsxab0.c\
	$(HEADERS)

amqsxaf0.o: amqsxaf0.c\
	$(HEADERS)

util.o: util.c\
	$(HEADERS)

#
# Link all Object files
#

amqsxag0:  $(OBJS)
	$(CC) -o amqsxag0 $(MQMLIBPATH) $(MQMLIBS) $(DB2LIBPATH)\
	$(DB2LIBS) $(OBJS)
```

Exercise 4: Make Files

Appendix C. Message Segmenting Examples

This appendix lists the three programs used in Exercise 6 and Exercise 7 to explain arbitrary and application segmentation:

- PUT_SEG1.C (a modification of amqsput0.c) demonstrates arbitrary segmentation.
- BCG_SEG1.C (a modification of amqsbcg0.c) demonstrates how to get only complete logical messages.
- PUT_SEG2.C (a modification of amqsput0.c) is an example for application segmentation.

C.1 PUT_SEG1 Performing Arbitrary Segmenting

```
/******************************************************************/
/* Program name: PUT_SEG1  (based on AMQSPIT0.C)                 */
/* Description:  Sample C program that puts messages to          */
/*               a message queue (example using MQPUT)           */
/*  Statement:   Licensed Materials - Property of IBM            */
/*               (C) Copyright IBM Corp. 1994, 1997              */
/******************************************************************/
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <cmqc.h>

int main(int argc, char **argv)
{
/******************************************************************/
/*    Declare file and character for sample input                */
/******************************************************************/
  FILE *fp;
                                   /* MQI structures:             */
  MQOD     od = {MQOD_DEFAULT};    /* Object Descriptor           */
  MQMD     md = {MQMD_DEFAULT};    /* Message Descriptor          */
  MQPMO   pmo = {MQPMO_DEFAULT};   /* put message options         */
  MQHCONN  Hcon;                   /* connection handle           */
  MQHOBJ   Hobj;                   /* object handle               */
  MQLONG   O_options;              /* MQOPEN options              */
  MQLONG   C_options;              /* MQCLOSE options             */
  MQLONG   CompCode;               /* completion code             */
  MQLONG   OpenCode;               /* MQOPEN completion code      */
  MQLONG   Reason;                 /* reason code                 */
  MQLONG   CReason;                /* reason code for MQCONN      */
  MQLONG   buflen;                 /* buffer length               */
  char     buffer[5000];           /* our large message buffer    */
  char     QMName[50];             /* queue manager name          */
```

Exercise 6: PUT_SEG1.C

```
printf("Sample PUT_SEG1 start\n");
if (argc < 2)
{
  printf("Required parameter missing - queue name\n");
  exit(99);
}
/*****************************************************************/
/*   Connect to queue manager                                    */
/*****************************************************************/
QMName[0] = 0;    /* default */
if (argc > 2)
  strcpy(QMName, argv[2]);
MQCONN(QMName,                /* queue manager                  */
       &Hcon,                 /* connection handle              */
       &CompCode,             /* completion code                */
       &CReason);             /* reason code                    */

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

/*****************************************************************/
/*   Use parameter as the name of the target queue               */
/*****************************************************************/
strncpy(od.ObjectName, argv[1], (size_t)MQ_Q_NAME_LENGTH);
printf("target queue is %s\n", od.ObjectName);

/*****************************************************************/
/*   Open the target message queue for output                    */
/*****************************************************************/
O_options = MQOO_OUTPUT           /* open queue for output        */
        + MQOO_FAIL_IF_QUIESCING; /* but not if MQM stopping      */
MQOPEN(Hcon,                      /* connection handle            */
       &od,                       /* object descriptor for queue  */
       O_options,                 /* open options                 */
       &Hobj,                     /* object handle                */
       &OpenCode,                 /* MQOPEN completion code       */
       &Reason);                  /* reason code                  */
                    /* report reason, if any; stop if failed      */
if (Reason != MQRC_NONE) {
  printf("MQOPEN ended with reason code %ld\n", Reason);
}
if (OpenCode == MQCC_FAILED) {
  printf("unable to open queue for output\n");
```

```
}
/****************************************************************/
/*   Read lines from the file and put them to the message queue   */
/*   Loop until null line or end of file, or there is a failure   */
/****************************************************************/
CompCode = OpenCode;        /* use MQOPEN result for initial test */
fp = stdin;

memcpy(md.Format,          /* character string format        */
       MQFMT_STRING, (size_t)MQ_FORMAT_LENGTH);

/****************************************************************/
/*  Allow segmentation by the system                             */
/****************************************************************/
md.MsgFlags = MQMF_SEGMENTATION_ALLOWED;
md.Version  = MQMD_VERSION_2;

while (CompCode != MQCC_FAILED)
{
  if (fgets(buffer, sizeof(buffer), fp) != NULL)
  {
    buflen = strlen(buffer);       /* length without null         */
    if (buffer[buflen-1] == '\n')  /* last char is a new-line     */
    {
      buffer[buflen-1]  = '\0';    /* replace new-line with null  */
      --buflen;                    /* reduce buffer length        */
    }
  }
  else buflen = 0;        /* treat EOF same as null line          */

  /**************************************************************/
  /*   Put each buffer to the message queue                      */
  /**************************************************************/
  if (buflen > 0)
  {
    memcpy(md.MsgId,           /* reset MsgId to get a new one    */
           MQMI_NONE, sizeof(md.MsgId) );
    memcpy(md.CorrelId,        /* reset CorrelId to get a new one */
           MQCI_NONE, sizeof(md.CorrelId) );

    MQPUT(Hcon,                /* connection handle               */
          Hobj,                /* object handle                   */
          &md,                 /* message descriptor              */
          &pmo,                /* default options (datagram)      */
          buflen,              /* buffer length                   */
          buffer,              /* message buffer                  */
          &CompCode,           /* completion code                 */
          &Reason);            /* reason code                     */
```

Exercise 6: BCG_SEG1.C

```
      if (Reason != MQRC_NONE)  /* report reason, if any          */
        printf("MQPUT ended with reason code %ld\n", Reason);
    }
    else   /* satisfy end condition when empty line is read */
      CompCode = MQCC_FAILED;
  }

  /******************************************************************/
  /*   Close the target queue (if it was opened)                    */
  /******************************************************************/
  if (OpenCode != MQCC_FAILED)
  {
    C_options = 0;                 /* no close options             */
    MQCLOSE(Hcon,                  /* connection handle            */
            &Hobj,                  /* object handle                */
            C_options,
            &CompCode,              /* completion code              */
            &Reason);               /* reason code                  */

    if (Reason != MQRC_NONE)        /* report reason, if any        */
      printf("MQCLOSE ended with reason code %ld\n", Reason);
  }

  /******************************************************************/
  /*   Disconnect from MQM if not already connected                 */
  /******************************************************************/
  if (CReason != MQRC_ALREADY_CONNECTED)
  {
    MQDISC(&Hcon,                  /* connection handle            */
           &CompCode,              /* completion code              */
           &Reason);               /* reason code                  */

    if (Reason != MQRC_NONE)       /* report reason, if any        */
      printf("MQDISC ended with reason code %ld\n", Reason);
  }

  /******************************************************************/
  /* END OF PUT_SEG1                                                */
  /******************************************************************/
  printf("Sample PUT_SEG1 end\n");
  return(0);
}
```

C.2 BCG_SEG1 Browsing only Logical Messages

```
/********************************************************************/
/* Program name: BCG_SEG1                                           */
/* Description : Sample program to read and output both the message */
/*               descriptor fields and the message content of all the */
/*               messages on a queue.                               */
/*               (Based on amqsbcg0.c.; modified to tell the queue  */
/*                manager to only deal with logical messages.)      */
/* Statement:    Licensed Materials - Property of IBM               */
/*               (C) Copyright IBM Corp. 1994, 1997                 */
/*                                                                  */
/*                                                                  */
/* Function    : This program is passed the name of a queue manager */
/*               and a queue. It then reads each message from the   */
/*               queue and outputs the following to the stdout      */
/*                    -  Formatted message descriptor fields        */
/*                    -  Message data (dumped in hex and, where     */
/*                       possible, character format)                */
/*                                                                  */
/* Restriction : This program is currently restricted to printing   */
/*               the first 32767 characters of the message and will */
/*               fail with reason 'truncated-msg' if a longer       */
/*               message is read                                    */
/********************************************************************/
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <time.h>
#include <ctype.h>
#include <locale.h>
#include <cmqc.h>

#define    CHARS_PER_LINE  16  /* Used in formatting the message      */
#define    BUFFERLENGTH 32767  /* Max length of message accepted      */

/********************************************************************/
/* Function name:    printMD                                        */
/* Description:      Prints the name of each field in the message   */
/*                   descriptor together with it's contents in the  */
/*                   appropriate format viz:                        */
/*                   integers as a number (%d)                      */
/*                   binary fields as a series of hex digits (%02X) */
/*                   character fields as characters (%s)            */
/********************************************************************/
void printMD(MQMD *MDin)
{
```

Exercise 6: BCG_SEG1.C

```
int i;

printf("\n****Message descriptor****\n");
printf("\n  StrucId  : '%.4s'", MDin->StrucId);
printf("  Version : %d", MDin->Version);
printf("\n  Report   : %d", MDin->Report);
printf("  MsgType : %d", MDin->MsgType);
printf("\n  Expiry   : %d", MDin->Expiry);
printf("  Feedback : %d", MDin->Feedback);
printf("\n  Encoding : %d", MDin->Encoding);
printf("  CodedCharSetId : %d", MDin->CodedCharSetId);
printf("\n  Format : '%.*s'", MQ_FORMAT_LENGTH, MDin->Format);
printf("\n  Priority : %d", MDin->Priority);
printf("  Persistence : %d", MDin->Persistence);
printf("\n  MsgId : X'");

for (i = 0 ; i < MQ_MSG_ID_LENGTH ; i++)
  printf("%02X",MDin->MsgId[i] );

printf("'");
printf("\n  CorrelId : X'");

for (i = 0 ; i < MQ_CORREL_ID_LENGTH ; i++)
  printf("%02X",MDin->CorrelId[i] );

printf("'");
printf("\n  BackoutCount : %d", MDin->BackoutCount);
printf("\n  ReplyToQ       : '%.*s'", MQ_Q_NAME_LENGTH,
       MDin->ReplyToQ);
printf("\n  ReplyToQMgr    : '%.*s'", MQ_Q_MGR_NAME_LENGTH,
       MDin->ReplyToQMgr);
printf("\n  ** Identity Context");
printf("\n  UserIdentifier : '%.*s'", MQ_USER_ID_LENGTH,
       MDin->UserIdentifier);
printf("\n  AccountingToken : \n   X'");

for (i = 0 ; i < MQ_ACCOUNTING_TOKEN_LENGTH ; i++)
  printf("%02X",MDin->AccountingToken[i] );

printf("'");
printf("\n  ApplIdentityData : '%.*s'", MQ_APPL_IDENTITY_DATA_LENGTH,
       MDin->ApplIdentityData);
printf("\n  ** Origin Context");
printf("\n  PutApplType    : '%d'", MDin->PutApplType);
printf("\n  PutApplName    : '%.*s'", MQ_PUT_APPL_NAME_LENGTH,
       MDin->PutApplName);
printf("\n  PutDate  : '%.*s'", MQ_PUT_DATE_LENGTH, MDin->PutDate);
printf("    PutTime  : '%.*s'", MQ_PUT_TIME_LENGTH, MDin->PutTime);
```

```
   printf("\n  ApplOriginData : '%.*s'\n", MQ_APPL_ORIGIN_DATA_LENGTH,
          MDin->ApplOriginData);
   printf("\n  GroupId : X'");

   for (i = 0 ; i < MQ_GROUP_ID_LENGTH ; i++)
     printf("%02X",MDin->GroupId[i] );

   printf("'");
   printf("\n  MsgSeqNumber   : '%d'", MDin->MsgSeqNumber);
   printf("\n  Offset         : '%d'", MDin->Offset);
   printf("\n  MsgFlags       : '%d'", MDin->MsgFlags);
   printf("\n  OriginalLength : '%d'", MDin->OriginalLength);
}  /* end printMD */


/**********************************************************************/
/* Function name:    main                                             */
/* Description:      Connects to the queue manager, opens the queue,  */
/*                   then gets each message from the queue in a loop  */
/*                   until an error occurs. The message descriptor    */
/*                   and message content are output to stdout for     */
/*                   each message. Any errors are output to stdout    */
/*                   and the program terminates.                      */
/**********************************************************************/
int  main(int argc, char *argv[] )
{
  /*                                                                  */
  /* variable declaration and initialisation                          */
  /*                                                                  */
  int i = 0;       /* loop counter                                    */
  int j = 0;       /* another loop counter                            */

  /* variables for MQCONN            ******/
  MQCHAR  QMmgrName[MQ_Q_MGR_NAME_LENGTH];
  MQHCONN Hconn = 0;
  MQLONG  CompCode,Reason,OpenCompCode;

  /* variables for MQOPEN            ******/
  MQCHAR  Queue[MQ_Q_NAME_LENGTH];
  MQOD    ObjDesc = { MQOD_DEFAULT };
  MQLONG  OpenOptions;
  MQHOBJ  Hobj = 0;

  /* variables for MQGET             ******/
  MQMD    MsgDesc = { MQMD_DEFAULT };
  PMQMD   pmdin ;
  MQGMO   GetMsgOpts = { MQGMO_DEFAULT };
  PMQGMO  pgmoin;
```

Exercise 6: BCG_SEG1.C

```
PMQBYTE Buffer;
MQLONG  BufferLength = BUFFERLENGTH;
MQLONG  DataLength;

/* variables for message formatting *****/
int  ch;
int  overrun;  /* used on MBCS characters */
int  mbcsmax;  /* used for MBCS characters */
int  char_len;  /* used for MBCS characters */
char line_text[CHARS_PER_LINE + 4]; /* allows for up to 3 MBCS bytes overrun */
int  chars_this_line = 0;
int  lines_printed   = 0;
int  page_number     = 1;

/*                                      */
/* Use a version 2 MQMD incase the      */
/* message is Segmented/Grouped         */
/*                                      */
MsgDesc.Version = MQMD_VERSION_2 ;

/*                                      */
/* Initialise storage ....              */
/*                                      */
pmdin  = malloc(sizeof(MQMD));
pgmoin = malloc(sizeof(MQGMO));
Buffer = malloc(BUFFERLENGTH);

/*                                      */
/* determine locale for MBCS handling   */
/*                                      */
setlocale(LC_ALL,"");  /* for mbcs charactersets */
mbcsmax = MB_CUR_MAX;  /* for mbcs charactersets */

/*                                      */
/* Handle the arguments passed          */
/*                                      */
printf("\nAMQSBCG0 - starts here\n");
printf(  "**********************\n ");

if (argc < 2)
{
  printf("Required parameter missing - queue name\n");
  printf("\n  Usage: %s QName [ QMgrName ]\n",argv[0]);
  return 4 ;
}

/*****************************************************************/
/*                                                               */
```

```
/*   Connect to queue manager                                      */
/*                                                                 */
/*****************************************************************/
QMmgrName[0] =  '\0';                /* set to null   default QM */
if (argc > 2)
  strcpy(QMmgrName, argv[2]);

strncpy(Queue,argv[1],MQ_Q_NAME_LENGTH);

/*                                        */
/* Start function here....                */
/*                                        */
MQCONN(QMmgrName,
       &Hconn,
       &CompCode,
       &Reason);

if (CompCode != MQCC_OK)
{
  printf("\n MQCONN failed with CompCode:%d, Reason:%d",
         CompCode,Reason);
  return (CompCode);
}

/*                                         */
/* Set the options for the open call       */
/*                                         */

OpenOptions = MQOO_BROWSE;

/*    @@@@ Use this for destructive read    */
/*         instead of the above.            */
/* OpenOptions = MQOO_INPUT_SHARED;         */
/*                                          */

strncpy(ObjDesc.ObjectName, Queue, MQ_Q_NAME_LENGTH);

printf("\n MQOPEN - '%.*s'", MQ_Q_NAME_LENGTH,Queue);
MQOPEN(Hconn,
       &ObjDesc,
       OpenOptions,
       &Hobj,
       &OpenCompCode,
       &Reason);

if (OpenCompCode != MQCC_OK)
{
  printf("\n MQOPEN failed with CompCode:%d, Reason:%d",
```

Exercise 6: BCG_SEG1.C

```
          OpenCompCode,Reason);

   printf("\n MQDISC");

   MQDISC(&Hconn,
          &CompCode,
          &Reason);

   if (CompCode != MQCC_OK)
   {
     printf("\n  failed with CompCode:%d, Reason:%d",
            CompCode,Reason);
   }

   return (OpenCompCode);
 }

 printf("\n ");
 /*                                         */
 /* Set the options for the get calls       */
 /*                                         */
 GetMsgOpts.Options = MQGMO_NO_WAIT ;

 /* @@@@ Comment out the next line for        */
 /*      destructive read                     */

 GetMsgOpts.Options += MQGMO_BROWSE_NEXT ;

 /******************************************************************/
 /*                                                                */
 /*   Get the system to re-assemble all segments                   */
 /*                                                                */
 /******************************************************************/

 GetMsgOpts.Options += MQGMO_COMPLETE_MSG ;

 /*                                         */
 /* Loop until MQGET unsuccessful           */
 /*                                         */
 for (j = 1; CompCode == MQCC_OK; j++)
    {
    /*                                          */
    /* Set up the output format of the report   */
    /*                                          */
    if (page_number == 1)
    {
      lines_printed = 29;
      page_number = -1;
```

```
}
else
{
  printf("\n ");
  lines_printed = 22;
}

/*                                              */
/* Reset the message descriptor to the required  */
/* defaults and initialize the buffer to blanks  */
/*                                              */

/* ????? There is a new field in the Get Message Options      */
/*       which relieves the need for resetting the MessageID  */
/*       and CorrelID each time. Do you know what it is?      */
/*       ANSWER: MatchOptions                                 */

pmdin = memcpy(pmdin, &MsgDesc, sizeof(MQMD) );
pgmoin =  memcpy(pgmoin, &GetMsgOpts, sizeof(MQGMO) );
memset(Buffer,' ',BUFFERLENGTH);

MQGET(Hconn,
      Hobj,
      pmdin,
      pgmoin,
      BufferLength,
      Buffer,
      &DataLength,
      &CompCode,
      &Reason);

if  (CompCode != MQCC_OK)
{
  if (Reason != MQRC_NO_MSG_AVAILABLE)
  {
    printf("\n MQGET %d, failed with CompCode:%d Reason:%d",
           j,CompCode,Reason);
  }
  else
  {
    printf("\n \n \n No more messages ");
  }
}
else
{
  /* Print the message             */
  /*                               */
  printf("\n ");
```

Exercise 6: BCG_SEG1.C

```
printf("\n MQGET of message number %d ", j);
/*                                */
/* first the Message Descriptor  */
printMD(pmdin);

/*                                */
/* then dump the Message         */
/*                                */
printf("\n ");
printf("\n****   Message      ****\n ");
Buffer[DataLength] = '\0';
printf("\n length - %d bytes\n ", DataLength);
ch = 0;
overrun = 0;
do
{
  chars_this_line = 0;
  printf("\n%08X: ",ch);
  for (;overrun>0; overrun--) /* for MBCS overruns */
  {
    printf("  ");            /* dummy space for characters  */
    line_text[chars_this_line] = ' ';
                         /* included in previous line */
    chars_this_line++;
    if (overrun % 2)
      printf(" ");
  }
  while ( (chars_this_line < CHARS_PER_LINE) &&
          (ch < DataLength) )
  {
    char_len = mblen((char *)&Buffer[ch],mbcsmax);
    if (char_len < 1)   /* badly formed mbcs character */
      char_len = 1;     /* or NULL treated as sbcs     */
    if (char_len > 1 )
    { /* mbcs case, assumes mbcs are all printable */
      for (;char_len >0;char_len--)
      {
        if ((chars_this_line % 2 == 0) &&
            (chars_this_line < CHARS_PER_LINE))
          printf(" ");
        printf("%02X",Buffer[ch] );
        line_text[chars_this_line] = Buffer[ch];
        chars_this_line++;
        ch++;
      }
    }
    else
    {  /* sbcs case */
```

```
          if (chars_this_line % 2 == 0)
            printf(" ");
          printf("%02X",Buffer[ch] );
          line_text[chars_this_line] =
              isprint(Buffer[ch]) ? Buffer[ch] : '.';
          chars_this_line++;
          ch++;
        }
      }

      /* has an mbcs character overun the usual end? */
      if (chars_this_line > CHARS_PER_LINE)
         overrun = chars_this_line - CHARS_PER_LINE;

      /* pad with blanks to format the last line correctly */
      if (chars_this_line < CHARS_PER_LINE)
      {
        for ( ;chars_this_line < CHARS_PER_LINE;
             chars_this_line++)
        {
          if (chars_this_line % 2 == 0) printf(" ");
          printf("  ");
          line_text[chars_this_line] = ' ';
        }
      }

      /* leave extra space between colums if MBCS characters possible */
      for (i=0;i < ((mbcsmax - overrun - 1) *2);i++)
      {
        printf(" "); /* prints space between hex represention and character */
      }

      line_text[chars_this_line] = '\0';
      printf(" '%s'",line_text);
      lines_printed += 1;
      if (lines_printed >= 60)
      {
        lines_printed = 0;
        printf("\n ");
      }
    }
    while (ch < DataLength);

  } /* end of message received 'else' */

} /* end of for loop */

printf("\n MQCLOSE");
```

Exercise 6: BCG_SEG1.C

```
  MQCLOSE(Hconn,
          &Hobj,
          MQCO_NONE,
          &CompCode,
          &Reason);

  if (CompCode != MQCC_OK)
  {
    printf("\n  failed with CompCode:%d, Reason:%d",
           CompCode,Reason);
    return (CompCode);
  }

  printf("\n MQDISC");
  MQDISC(&Hconn,
         &CompCode,
         &Reason);

  if (CompCode != MQCC_OK)
  {
    printf("\n  failed with CompCode:%d, Reason:%d",
           CompCode,Reason);
    return (CompCode);
  }

  return(0);
}
```

C.3 PUT_SEG2 Performing Application Segmenting

```
/********************************************************************/
/* Program name: PUT_SEG2                                           */
/*              (Based on AMQSPUTO.C)                               */
/* Description: Sample C program that puts messages to              */
/*              a message queue (example using MQPUT)               */
/*  Statement:     Licensed Materials - Property of IBM             */
/*                 (C) Copyright IBM Corp. 1994, 1997               */
/* Function:                                                        */
/*   PUT_SEG2 is a sample C program to put messages on a message    */
/*   queue, and is an example of the use of MQPUT.                  */
/*   Changed to appliacation-segment large messages                 */
/********************************************************************/
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <cmqc.h>

int main(int argc, char **argv)
{
  /*  Declare file and character for sample input                  */
  FILE *fp;

  /*   Declare MQI structures needed                                */
  MQOD     od = {MQOD_DEFAULT};    /* Object Descriptor             */
  MQMD     md = {MQMD_DEFAULT};    /* Message Descriptor            */
  MQPMO   pmo = {MQPMO_DEFAULT};   /* put message options           */
  MQHCONN  Hcon;                   /* connection handle             */
  MQHOBJ   Hobj;                   /* object handle                 */
  MQLONG   O_options;              /* MQOPEN options                */
  MQLONG   C_options;              /* MQCLOSE options               */
  MQLONG   CompCode;               /* completion code               */
  MQLONG   OpenCode;               /* MQOPEN completion code        */
  MQLONG   Reason;                 /* reason code                   */
  MQLONG   CReason;                /* reason code for MQCONN        */
  MQLONG   buflen;                 /* buffer length                 */
  char     buffer[1000];           /* ????? message buffer          */
  char     QMName[50];             /* queue manager name            */

  printf("Sample PUT_SEG2 start\n");
  if (argc < 2)
  {
    printf("Required parameter missing - queue name\n");
    exit(99);
  }
```

Exercise 7: PUT_SEG2.C

```
/*****************************************************************/
/*   Connect to queue manager                                    */
/*****************************************************************/
QMName[0] = 0;    /* default */
if (argc > 2)
  strcpy(QMName, argv·2‘);
MQCONN(QMName,                /* queue manager                   */
       &Hcon,                 /* connection handle               */
       &CompCode,             /* completion code                 */
       &CReason);             /* reason code                     */
if (CompCode == MQCC_FAILED)
{
  printf(”MQCONN ended with reason code %ld\n”, CReason);
  exit( (int)CReason );
}
/*****************************************************************/
/*   Use parameter as the name of the target queue               */
/*****************************************************************/
strncpy(od.ObjectName, argv[1], (size_t)MQ_Q_NAME_LENGTH);
printf(”target queue is %s\n”, od.ObjectName);

/*****************************************************************/
/*   Open the target message queue for output                    */
/*****************************************************************/
O_options = MQOO_OUTPUT         /* open queue for output         */
        + MQOO_FAIL_IF_QUIESCING; /* but not if MQM stopping     */
MQOPEN(Hcon,                    /* connection handle             */
       &od,                     /* object descriptor for queue   */
       O_options,               /* open options                  */
       &Hobj,                   /* object handle                 */
       &OpenCode,               /* MQOPEN completion code        */
       &Reason);                /* reason code                   */

if (Reason != MQRC_NONE)
  printf(”MQOPEN ended with reason code %ld\n”, Reason);

if (OpenCode == MQCC_FAILED)
  printf(”unable to open queue for output\n”);

/*****************************************************************/
/*   Read lines from the file and put them to the message queue  */
/*   Loop until null line or end of file, or there is a failure  */
/*****************************************************************/
CompCode = OpenCode;        /* use MQOPEN result for initial test */
fp = stdin;

memcpy(md.Format,          /* character string format        */
       MQFMT_STRING, (size_t)MQ_FORMAT_LENGTH);
```

```
 md.Version  = MQMD_VERSION_2 ;

/*******************************************************************/
/*    Set the MsgFlag to show this message is a segment            */
/*******************************************************************/
 md.MsgFlags = MQMF_SEGMENT ;

/*******************************************************************/
/*    This says to retrieve the messages in logical order          */
/*    Without this option segments are retrieved in physical order */
/*******************************************************************/
 pmo.Options = MQPMO_LOGICAL_ORDER ;

  while (CompCode != MQCC_FAILED)
  {
    if (fgets(buffer, sizeof(buffer), fp) != NULL)
    {
      buflen = strlen(buffer);       /* length without null        */
      if (buffer[buflen-1] == '\n')  /* last char is a new-line    */
      {
        buffer[buflen-1]  = '\0';    /* replace new-line with null */
        --buflen;                    /* reduce buffer length       */
      }
    }
    else buflen = 0;        /* treat EOF same as null line         */

    /***************************************************************/
    /*                                                             */
    /*   Put each buffer to the message queue                      */
    /*                                                             */
    /***************************************************************/
    if (buflen > 0)
    {
      memcpy(md.MsgId,           /* reset MsgId to get a new one    */
             MQMI_NONE, sizeof(md.MsgId) );
      memcpy(md.CorrelId,        /* reset CorrelId to get a new one */
             MQCI_NONE, sizeof(md.CorrelId) );

      MQPUT(Hcon,                /* connection handle               */
            Hobj,                /* object handle                   */
            &md,                 /* message descriptor              */
            &pmo,                /* default options (datagram)      */
            buflen,              /* buffer length                   */
            buffer,              /* message buffer                  */
            &CompCode,           /* completion code                 */
            &Reason);            /* reason code                     */
```

Exercise 7: PUT_SEG2.C

```
      if (Reason != MQRC_NONE)
        printf("MQPUT ended with reason code %ld\n", Reason);
    }
    else
    {

 /****************************************************************/
 /*    We add one last (empty) segment with the last_segment       */
 /*    specified in the message descriptor                         */
 /****************************************************************/
         md.MsgFlags = MQMF_LAST_SEGMENT ;   /* indicate the LAST      */

         memcpy(md.MsgId,           /* reset MsgId to get a new one    */
                MQMI_NONE, sizeof(md.MsgId) );

         memcpy(md.CorrelId,        /* reset CorrelId to get a new one */
                MQCI_NONE, sizeof(md.CorrelId) );

         MQPUT(Hcon,                /* connection handle               */
               Hobj,                /* object handle                   */
               &md,                 /* message descriptor              */
               &pmo,                /* default options (datagram)      */
               0,                   /* ????? buffer length             */
               buffer,              /* message buffer                  */
               &CompCode,           /* completion code                 */
               &Reason);            /* reason code                     */

         if (Reason != MQRC_NONE)
           printf("MQPUT ended with reason code %ld\n", Reason);
    CompCode = MQCC_FAILED; /* satisfy end condition when empty line is read */
    }
  }
  /**************************************************************/
  /*   Close the target queue (if it was opened)                 */
  /**************************************************************/
  if (OpenCode != MQCC_FAILED)
  {
    C_options = 0;                  /* no close options            */
    MQCLOSE(Hcon,                   /* connection handle           */
            &Hobj,                   /* object handle              */
            C_options,
            &CompCode,               /* completion code            */
            &Reason);                /* reason code                */

    if (Reason != MQRC_NONE)
      printf("MQCLOSE ended with reason code %ld\n", Reason);
  }
  /**************************************************************/
```

```
  /*   Disconnect from MQM if not already connected                  */
  /******************************************************************/
  if (CReason != MQRC_ALREADY_CONNECTED)
  {
    MQDISC(&Hcon,                   /* connection handle            */
           &CompCode,               /* completion code              */
           &Reason);                /* reason code                  */

    if (Reason != MQRC_NONE)
      printf("MQDISC ended with reason code %ld\n", Reason);
  }

  /******************************************************************/
  /*                                                                */
  /* END OF PUT_SEG2                                                */
  /*                                                                */
  /******************************************************************/
  printf("Sample PUT_SEG2 end\n");
  return(0);
}
```

Exercise 7: PUT_SEG2.C

Appendix D. Message Grouping Examples

This appendix lists the two programs used in Exercise 8 to explain message groups.

- PUT_GRP1.C (a modification of amqsput0.c) demonstrates how to put messages in a group.
- BCG_GRP1.C (a modification of amqsbcg0.c) shows how to ensure that no message is retrieved until all messages in a group are present.

D.1 Source of PUT_GRP1

```
/********************************************************************/
/* Program name: put_grp1                                           */
/* Description: Sample C program that puts messages to              */
/*             a message queue (example using MQPUT)                */
/*  Statement:     Licensed Materials - Property of IBM              */
/*                 (C) Copyright IBM Corp. 1994, 1997               */
/* Function:                                                        */
/*   put_grp1 is a sample C program to put messages on a message    */
/*   queue, and is an example of the use of MQPUT.                  */
/*   Changed to PUT grouped messages                                */
/********************************************************************/
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <cmqc.h>

int main(int argc, char **argv)
{
  FILE *fp;

  MQOD     od = {MQOD_DEFAULT};    /* Object Descriptor             */
  MQMD     md = {MQMD_DEFAULT};    /* Message Descriptor            */
  MQPMO   pmo = {MQPMO_DEFAULT};   /* put message options           */
  MQHCONN  Hcon;                   /* connection handle             */
  MQHOBJ   Hobj;                   /* object handle                 */
  MQLONG   O_options;              /* MQOPEN options                */
  MQLONG   C_options;              /* MQCLOSE options               */
  MQLONG   CompCode;               /* completion code               */
  MQLONG   OpenCode;               /* MQOPEN completion code        */
  MQLONG   Reason;                 /* reason code                   */
  MQLONG   CReason;                /* reason code for MQCONN        */
  MQLONG   buflen;                 /* buffer length                 */
  char     buffer[100];            /* message buffer                */
  char     QMName[50];             /* queue manager name            */
```

Exercise 8: PUT_GRP1.C

```
printf("Sample put_grp1 start\n");
if (argc < 2)
{
  printf("Required parameter missing - queue name\n");
  exit(99);
}
/******************************************************************/
/*   Connect to queue manager                                     */
/******************************************************************/
QMName[0] = 0;    /* default */
if (argc > 2)
  strcpy(QMName, argv[2]);
MQCONN(QMName,                /* queue manager                  */
       &Hcon,                 /* connection handle              */
       &CompCode,             /* completion code                */
       &CReason);             /* reason code                    */

/* report reason and stop if it failed     */
if (CompCode == MQCC_FAILED)
{
  printf("MQCONN ended with reason code %ld\n", CReason);
  exit( (int)CReason );
}
/******************************************************************/
/*   Use parameter as the name of the target queue                */
/******************************************************************/
strncpy(od.ObjectName, argv[1], (size_t)MQ_Q_NAME_LENGTH);
printf("target queue is %s\n", od.ObjectName);

/******************************************************************/
/*   Open the target message queue for output                     */
/******************************************************************/
O_options = MQOO_OUTPUT           /* open queue for output       */
        + MQOO_FAIL_IF_QUIESCING; /* but not if MQM stopping     */
MQOPEN(Hcon,                      /* connection handle           */
       &od,                       /* object descriptor for queue */
       O_options,                 /* open options                */
       &Hobj,                     /* object handle               */
       &OpenCode,                 /* MQOPEN completion code      */
       &Reason);                  /* reason code                 */

if (Reason != MQRC_NONE)
  printf("MQOPEN ended with reason code %ld\n", Reason);
if (OpenCode == MQCC_FAILED)
  printf("unable to open queue for output\n");
```

```
/****************************************************************/
/*   Read lines from the file and put them to the message queue   */
/*   Loop until null line or end of file, or there is a failure   */
/****************************************************************/
CompCode = OpenCode;        /* use MQOPEN result for initial test */
fp = stdin;
memcpy(md.Format,          /* character string format        */
       MQFMT_STRING, (size_t)MQ_FORMAT_LENGTH);

/****************************************************************/
/*   We will use MQ API Version 2 function                       */
/****************************************************************/
md.Version  = MQMD_VERSION_2 ;
/****************************************************************/
/*   Tell the queue manager that the messages that follow are    */
/*   part of a group                                             */
/****************************************************************/
md.MsgFlags = MQMF_MSG_IN_GROUP ;
/****************************************************************/
/*   Tell the queue manager that the messages that follow are    */
/*   to be kept in sequence within the group.                    */
/****************************************************************/
pmo.Options = MQPMO_LOGICAL_ORDER ;

while (CompCode != MQCC_FAILED)
{
  if (fgets(buffer, sizeof(buffer), fp) != NULL)
  {
    buflen = strlen(buffer);       /* length without null        */
    if (buffer[buflen-1] == '\n')  /* last char is a new-line    */
    {
      buffer[buflen-1]  = '\0';    /* replace new-line with null  */
      --buflen;                    /* reduce buffer length        */
    }
  }
  else buflen = 0;        /* treat EOF same as null line          */

  /**************************************************************/
  /*   Put each buffer to the message queue                      */
  /**************************************************************/
  if (buflen > 0)
  {
    memcpy(md.MsgId,           /* reset MsgId to get a new one    */
           MQMI_NONE, sizeof(md.MsgId) );
    memcpy(md.CorrelId,        /* reset CorrelId to get a new one */
           MQCI_NONE, sizeof(md.CorrelId) );
```

Exercise 8: PUT_GRP1.C

```
     MQPUT(Hcon,                /* connection handle              */
           Hobj,                /* object handle                  */
           &md,                 /* message descriptor             */
           &pmo,                /* default options (datagram)     */
           buflen,              /* buffer length                  */
           buffer,              /* message buffer                 */
           &CompCode,           /* completion code                */
           &Reason);            /* reason code                    */

     if (Reason != MQRC_NONE)
       printf("MQPUT ended with reason code %ld\n", Reason);
   }
   else
   {
 /*****************************************************************/
 /*   Put out one last (empty) message to end the group           */
 /*****************************************************************/
     md.MsgFlags = MQMF_LAST_MSG_IN_GROUP ;

     memcpy(md.MsgId,           /* reset MsgId to get a new one   */
            MQMI_NONE, sizeof(md.MsgId) );

     memcpy(md.CorrelId,        /* reset CorrelId to get a new one */
            MQCI_NONE, sizeof(md.CorrelId) );

     MQPUT(Hcon,                /* connection handle              */
           Hobj,                /* object handle                  */
           &md,                 /* message descriptor             */
           &pmo,                /* default options (datagram)     */
           0,                   /* ????? buffer length            */
           buffer,              /* message buffer                 */
           &CompCode,           /* completion code                */
           &Reason);            /* reason code                    */

     /* report reason, if any */
     if (Reason != MQRC_NONE)
     {
       printf("MQPUT ended with reason code %ld\n", Reason);
     }
     CompCode = MQCC_FAILED; /* satisfy end condition when empty line is read */
   }
 }
```

```
  /******************************************************************/
  /*   Close the target queue (if it was opened)                    */
  /******************************************************************/
  if (OpenCode != MQCC_FAILED)
  {
    C_options = 0;                  /* no close options             */
    MQCLOSE(Hcon,                   /* connection handle            */
            &Hobj,                   /* object handle                */
            C_options,
            &CompCode,               /* completion code              */
            &Reason);                /* reason code                  */

    /* report reason, if any     */
    if (Reason != MQRC_NONE)
    {
      printf("MQCLOSE ended with reason code %ld\n", Reason);
    }
  }
  /******************************************************************/
  /*   Disconnect from MQM if not already connected                 */
  /******************************************************************/
  if (CReason != MQRC_ALREADY_CONNECTED)
  {
    MQDISC(&Hcon,                   /* connection handle            */
           &CompCode,               /* completion code              */
           &Reason);                /* reason code                  */

    /* report reason, if any     */
    if (Reason != MQRC_NONE)
    {
      printf("MQDISC ended with reason code %ld\n", Reason);
    }
  }
  /******************************************************************/
  /* END OF put_grp1                                                */
  /******************************************************************/
  printf("Sample put_grp1 end\n");
  return(0);
}
```

D.2 Source of BCG_GRP1

```
/********************************************************************/
/* Program name: bcg_grp1                                           */
/* Description : Sample program to read and output both the         */
/*                message descriptor fields and the message content  */
/*                of all the messages on a queue                     */
/* Statement:     Licensed Materials - Property of IBM               */
/*                (C) Copyright IBM Corp. 1994, 1997                 */
/*                                                                   */
/*       Changed to browse message groups                            */
/********************************************************************/
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <time.h>
#include <ctype.h>
#include <locale.h>
#include <cmqc.h>

#define    CHARS_PER_LINE  16  /* Used in formatting the message */
#define    BUFFERLENGTH  32767  /* Max length of message accepted */

/********************************************************************/
/* Function name:    printMD                                         */
/* Description:      Prints the name of each field in the message    */
/*                   descriptor together with it's contents in the   */
/*                   appropriate format viz:                         */
/*                   integers as a number (%d)                       */
/*                   binary fields as a series of hex digits (%02X)  */
/*                   character fields as characters (%s)             */
/********************************************************************/
void printMD(MQMD *MDin)
{
   int i;
   printf("\n****Message descriptor****\n");
   printf("\n  StrucId  : '%.4s'", MDin->StrucId);
   printf("  Version : %d", MDin->Version);
   printf("\n  Report   : %d", MDin->Report);
   printf("  MsgType : %d", MDin->MsgType);
   printf("\n  Expiry   : %d", MDin->Expiry);
   printf("  Feedback : %d", MDin->Feedback);
   printf("\n  Encoding : %d", MDin->Encoding);
   printf("  CodedCharSetId : %d", MDin->CodedCharSetId);
   printf("\n  Format : '%.*s'", MQ_FORMAT_LENGTH, MDin->Format);
   printf("\n  Priority : %d", MDin->Priority);
   printf("  Persistence : %d", MDin->Persistence);
   printf("\n  MsgId : X'");
```

```
    for (i = 0 ; i < MQ_MSG_ID_LENGTH ; i++)
      printf("%02X",MDin->MsgId[i] );

    printf("'");
    printf("\n  CorrelId : X'");

    for (i = 0 ; i < MQ_CORREL_ID_LENGTH ; i++)
      printf("%02X",MDin->CorrelId[i] );

    printf("'");
    printf("\n  BackoutCount : %d", MDin->BackoutCount);
    printf("\n  ReplyToQ       : '%.*s'", MQ_Q_NAME_LENGTH,
           MDin->ReplyToQ);
    printf("\n  ReplyToQMgr    : '%.*s'", MQ_Q_MGR_NAME_LENGTH,
           MDin->ReplyToQMgr);
    printf("\n  ** Identity Context");
    printf("\n  UserIdentifier : '%.*s'", MQ_USER_ID_LENGTH,
           MDin->UserIdentifier);
    printf("\n  AccountingToken : \n   X'");

    for (i = 0 ; i < MQ_ACCOUNTING_TOKEN_LENGTH ; i++)
      printf("%02X",MDin->AccountingToken[i] );

    printf("'");
    printf("\n  ApplIdentityData : '%.*s'", MQ_APPL_IDENTITY_DATA_LENGTH,
           MDin->ApplIdentityData);
    printf("\n  ** Origin Context");
    printf("\n  PutApplType    : '%d'", MDin->PutApplType);
    printf("\n  PutApplName    : '%.*s'", MQ_PUT_APPL_NAME_LENGTH,
           MDin->PutApplName);
    printf("\n  PutDate  : '%.*s'", MQ_PUT_DATE_LENGTH, MDin->PutDate);
    printf("    PutTime  : '%.*s'", MQ_PUT_TIME_LENGTH, MDin->PutTime);
    printf("\n  ApplOriginData : '%.*s'\n", MQ_APPL_ORIGIN_DATA_LENGTH,
           MDin->ApplOriginData);
    printf("\n  GroupId : X'");

    for (i = 0 ; i < MQ_GROUP_ID_LENGTH ; i++)
      printf("%02X",MDin->GroupId[i] );

    printf("'");
    printf("\n  MsgSeqNumber   : '%d'", MDin->MsgSeqNumber);
    printf("\n  Offset         : '%d'", MDin->Offset);
    printf("\n  MsgFlags       : '%d'", MDin->MsgFlags);
    printf("\n  OriginalLength : '%d'", MDin->OriginalLength);
}  /* end printMD */
```

Exercise 8: BCG_GRP1.C

```
/********************************************************************/
/* Function name:    main                                          */
/* Description:      Connects to the queue manager, opens the queue, */
/*                   then gets each message from the queue in a loop */
/*                   until an error occurs. The message descriptor  */
/*                   and message content are output to stdout for   */
/*                   each message. Any errors are output to stdout  */
/*                   and the program terminates.                    */
/********************************************************************/
int  main(int argc, char *argv[] )
{
  /*                                                                */
  /* variable declaration and initialisation                        */
  int i = 0;       /* loop counter                                  */
  int j = 0;       /* another loop counter                          */

  /* variables for MQCONN            ******/
  MQCHAR  QMmgrName[MQ_Q_MGR_NAME_LENGTH];
  MQHCONN Hconn = 0;
  MQLONG  CompCode,Reason,OpenCompCode;

  /* variables for MQOPEN            ******/
  MQCHAR  Queue[MQ_Q_NAME_LENGTH];
  MQOD    ObjDesc = { MQOD_DEFAULT };
  MQLONG  OpenOptions;
  MQHOBJ  Hobj = 0;

  /* variables for MQGET             ******/
  MQMD    MsgDesc = { MQMD_DEFAULT };
  PMQMD   pmdin ;
  MQGMO   GetMsgOpts = { MQGMO_DEFAULT };
  PMQGMO  pgmoin;
  PMQBYTE Buffer;
  MQLONG  BufferLength = BUFFERLENGTH;
  MQLONG  DataLength;

  /* variables for message formatting *****/
  int  ch;
  int  overrun;  /* used on MBCS characters */
  int  mbcsmax;  /* used for MBCS characters */
  int  char_len;  /* used for MBCS characters */
  char line_text[CHARS_PER_LINE + 4];
  int  chars_this_line = 0;
  int  lines_printed   = 0;
  int  page_number     = 1;
```

```
/* Use a version 2 MQMD in case the      */
/* message is Segmented/Grouped          */
/*                                       */
MsgDesc.Version = MQMD_VERSION_2 ;

/*                                       */
/* Initialise storage ....               */
pmdin  = malloc(sizeof(MQMD));
pgmoin = malloc(sizeof(MQGMO));
Buffer = malloc(BUFFERLENGTH);

/* determine locale for MBCS handling    */
/*                                       */
setlocale(LC_ALL,"");  /* for mbcs charactersets */
mbcsmax = MB_CUR_MAX;  /* for mbcs charactersets */

/* Handle the arguments passed           */
/*                                       */
printf("\nbcg_grp1 - starts here\n");
printf(  "*********************\n ");

if (argc < 2)
{
  printf("Required parameter missing - queue name\n");
  printf("\n  Usage: %s QName [ QMgrName ]\n",argv[0]);
  return 4 ;
}
/******************************************************************/
/*   Connect to queue manager                                     */
/******************************************************************/
QMgrName[0] =  '\0';
if (argc > 2)
  strcpy(QMgrName, argv[2]);

strncpy(Queue,argv[1],MQ_Q_NAME_LENGTH);

MQCONN(QMgrName,
       &Hconn,
       &CompCode,
       &Reason);

if (CompCode != MQCC_OK)
{
  printf("\n MQCONN failed with CompCode:%d, Reason:%d",
         CompCode,Reason);
  return (CompCode);
}
```

Exercise 8: BCG_GRP1.C

```
/* Set the options for the open call      */
/*                                        */
OpenOptions = MQOO_BROWSE;

/*    @@@@ Use this for destructive read    */
/*         instead of the above.            */
/* OpenOptions = MQOO_INPUT_SHARED;         */
/*                                          */
strncpy(ObjDesc.ObjectName, Queue, MQ_Q_NAME_LENGTH);

printf("\n MQOPEN - '%.*s'", MQ_Q_NAME_LENGTH,Queue);
MQOPEN(Hconn,
       &ObjDesc,
       OpenOptions,
       &Hobj,
       &OpenCompCode,
       &Reason);

if (OpenCompCode != MQCC_OK)
{
  printf("\n MQOPEN failed with CompCode:%d, Reason:%d",
         OpenCompCode,Reason);

  printf("\n MQDISC");

  MQDISC(&Hconn,
         &CompCode,
         &Reason);

  if (CompCode != MQCC_OK)
    printf("\n  failed with CompCode:%d, Reason:%d",
           CompCode,Reason);
  return (OpenCompCode);
}
printf("\n ");
/*                                           */
/* Set the options for the get calls         */
/*                                           */
GetMsgOpts.Options = MQGMO_NO_WAIT ;

/* @@@@ Comment out the next line for          */
/*      destructive read                       */
GetMsgOpts.Options += MQGMO_BROWSE_NEXT ;
```

```
/******************************************************************/
/*   Use MQ API Version 2 Function                                */
/******************************************************************/
GetMsgOpts.Version = MQGMO_VERSION_2 ;

/******************************************************************/
/*   Create a sequence number for ALL messages                    */
/******************************************************************/
GetMsgOpts.Options += MQGMO_LOGICAL_ORDER ;
/*                                          */
/* Loop until MQGET unsuccessful            */
/*                                          */
for (j = 1; CompCode == MQCC_OK; j++)
   {
   /*                                              */
   /* Set up the output format of the report       */
   /*                                              */
   if (page_number == 1)
   {
     lines_printed = 29;
     page_number = -1;
   }
   else
   {
     printf("\n ");
     lines_printed = 22;
   }
/******************************************************************/
/*   Set ALL_MSGS_AVAILABLE for the FIRST message only            */
/******************************************************************/
if (j == 1)
   GetMsgOpts.Options += MQGMO_ALL_MSGS_AVAILABLE ;

   /*                                              */
   /* Reset the message descriptor to the required */
   /* defaults and initialize the buffer to blanks */
   /*                                              */
   pmdin = memcpy(pmdin, &MsgDesc, sizeof(MQMD) );
   pgmoin =  memcpy(pgmoin, &GetMsgOpts, sizeof(MQGMO) );
   memset(Buffer,' ',BUFFERLENGTH);

   MQGET(Hconn,
         Hobj,
         pmdin,
         pgmoin,
         BufferLength,
         Buffer,
         &DataLength,
```

Exercise 8: BCG_GRP1.C

```
          &CompCode,
          &Reason);

     if  (CompCode != MQCC_OK)
     {
       if (Reason != MQRC_NO_MSG_AVAILABLE)
       {
         printf("\n MQGET %d, failed with CompCode:%d Reason:%d",
                j,CompCode,Reason);
       }
       else
       {
         printf("\n \n \n No more messages ");
       }
     }
     else
     {
       /* Print the message              */
       /*                                */
       printf("\n ");
       printf("\n MQGET of message number %d ", j);
       /*                                */
       /* first the Message Descriptor  */
       printMD(pmdin);

       /*                                */
       /* then dump the Message         */
       /*                                */
       printf("\n ");
       printf("\n****   Message      ****\n ");
       Buffer[DataLength] = '\0';
       printf("\n length - %d bytes\n ", DataLength);
       ch = 0;
       overrun = 0;
       do
       {
         chars_this_line = 0;
         printf("\n%08X: ",ch);
         for (;overrun>0; overrun--) /* for MBCS overruns */
         {
           printf("  ");            /* dummy space for characters  */
           line_text[chars_this_line] = ' ';
                                 /* included in previous line */
           chars_this_line++;
           if (overrun % 2)
             printf(" ");
         }
         while ( (chars_this_line < CHARS_PER_LINE) &&
```

```
        (ch < DataLength) )
{
  char_len = mblen((char *)&Buffer[ch],mbcsmax);
  if (char_len < 1)   /* badly formed mbcs character */
    char_len = 1;     /* or NULL treated as sbcs     */
  if (char_len > 1 )
  { /* mbcs case, assumes mbcs are all printable */
    for (;char_len >0;char_len--)
    {
      if ((chars_this_line % 2 == 0) &&
          (chars_this_line < CHARS_PER_LINE))
        printf(" ");
      printf("%02X",Buffer[ch] );
      line_text[chars_this_line] = Buffer[ch];
      chars_this_line++;
      ch++;
    }
  }
  else
  {  /* sbcs case */
    if (chars_this_line % 2 == 0)
      printf(" ");
    printf("%02X",Buffer[ch] );
    line_text[chars_this_line] =
        isprint(Buffer[ch]) ? Buffer[ch] : '.';
    chars_this_line++;
    ch++;
  }
}

/* has an mbcs character overrun the usual end? */
if (chars_this_line > CHARS_PER_LINE)
   overrun = chars_this_line - CHARS_PER_LINE;

/* pad with blanks to format the last line correctly */
if (chars_this_line < CHARS_PER_LINE)
{
  for ( ;chars_this_line < CHARS_PER_LINE;
       chars_this_line++)
  {
    if (chars_this_line % 2 == 0) printf(" ");
    printf("  ");
    line_text[chars_this_line] = ' ';
  }
}

/* leave extra space between colums if MBCS characters possible */
for (i=0;i < ((mbcsmax - overrun - 1) *2);i++)
```

Exercise 8: BCG_GRP1.C

```
        {
          printf(" "); /* prints space between hex represention and character */
        }

        line_text[chars_this_line] = '\0';
        printf(" '%s'",line_text);
        lines_printed += 1;
        if (lines_printed >= 60)
        {
          lines_printed = 0;
          printf("\n ");
        }
      }
      while (ch < DataLength);

    } /* end of message received 'else' */

  } /* end of for loop */

  printf("\n MQCLOSE");
  MQCLOSE(Hconn,
          &Hobj,
          MQCO_NONE,
          &CompCode,
          &Reason);
  if (CompCode != MQCC_OK)
  {
    printf("\n  failed with CompCode:%d, Reason:%d",
           CompCode,Reason);
    return (CompCode);
  }

  printf("\n MQDISC");
  MQDISC(&Hconn,
         &CompCode,
         &Reason);
  if (CompCode != MQCC_OK)
  {
    printf("\n  failed with CompCode:%d, Reason:%d", CompCode,Reason);
    return (CompCode);
  }
  return(0);
}
```

Appendix E. Reference Message Example

This appendix lists the two programs used in Exercise 12.

- PUTREF.C creates and sends the reference message.
- GETREF.C reads the reference message and file.

E.1 Source of PUTREF

```
/******************************************************************/
/*                                                                */
/* PUTREF:  Put a Reference Message                               */
/*                                                                */
/******************************************************************/
#include <stdio.h>
#include <stddef.h>
#include <stdlib.h>
#include <string.h>
#include <ctype.h>
#include <cmqc.h>
/******************************************************************/
/* Constants                                                      */
/******************************************************************/
#define MAX_FILENAME_LENGTH 256

/******************************************************************/
/* typedefs                                                       */
/******************************************************************/
typedef struct tagMQRMHX{
        MQRMH  ref;
        MQCHAR SrcName[MAX_FILENAME_LENGTH];
        MQCHAR DestName[MAX_FILENAME_LENGTH];
} MQRMHX;
 MQRMHX  refx = {{MQRMH_DEFAULT}};           /* reference message  */
/******************************************************************/
/*                        MQ Variables                            */
/******************************************************************/
 MQHCONN Hcon;                               /* connection handle  */
 char    QMName[MQ_Q_MGR_NAME_LENGTH+1] = ""; /* queue manager name */
 MQLONG  CC;                                 /* completion code    */
 MQLONG  Reason;                             /* reason code        */
 MQOD    od = {MQOD_DEFAULT};                /* object descriptor  */
 MQHOBJ  Hobj = MQHO_UNUSABLE_HOBJ;          /* object handle      */
 MQMD    md = {MQMD_DEFAULT};                /* message descriptor */
 MQPMO   pmo = {MQPMO_DEFAULT};              /* put message options*/
```

Exercise 12: PUTREF.C

```
/******************************************************************/
/*                    Program Variables                           */
/******************************************************************/
 MQLONG  QMgrCCSID = -1;                       /* QMgr CCSID        */
 char    infile[MAX_FILENAME_LENGTH+1];
 char    outfile[MAX_FILENAME_LENGTH+1];
/******************************************************************/
/*                    Fields for MQINQ                            */
/******************************************************************/
 MQLONG  flags;
 MQLONG  Selectors[4];
 MQLONG  IntArray[2];
 MQCHAR  CharArray[100];
 char    BLANK48[MQ_Q_MGR_NAME_LENGTH+1] = "";
/****************************************************************/
/*                                                              */
/*                     Program                                  */
/*                                                              */
/****************************************************************/
int main(int argc, char **argv)
{
  printf("Start PUTREF\n");
  strcpy (infile,"c:\\test\\dw.fil");
  strcpy (outfile,"c:\\test\\dw1.txt");
  printf("Input=%s, output=%s\n", infile, outfile);

  /****************************************************************/
  /*   Connect to queue manager                                   */
  /****************************************************************/
  MQCONN(QMName,                  /* queue manager                */
         &Hcon,                   /* connection handle            */
         &CC,&Reason);            /* completion and reason codes  */
  if (CC == MQCC_FAILED) {
      printf("MQCONN ended with reason code %d\n", Reason);
      return(1);
  }
  /****************************************************************/
  /*   Use MQINQ to get queue manager's name and CCSID            */
  /****************************************************************/
  memcpy(od.ObjectQMgrName,BLANK48,MQ_Q_MGR_NAME_LENGTH);
  flags = MQOO_INQUIRE;
  od.ObjectType =MQOT_Q_MGR;
  MQOPEN(Hcon,                    /* connection handle            */
         &od,                     /* object descriptor            */
         flags,                   /* inquiry flags                */
         &Hobj,                   /* object handle                */
         &CC,&Reason);            /* completion and reason codes  */
```

```
if (CC == MQCC_FAILED) {
   printf("MQOPEN queue manager ended with reason code %d\n",Reason);
   goto PGM_DISC;
}
Selectors[0] = MQIA_CODED_CHAR_SET_ID;
Selectors[1] = MQCA_Q_MGR_NAME;
MQINQ(Hcon, Hobj,
      2L,                  /* number of selectors              */
      Selectors,           /* selector array                   */
      1L,                  /* number of integer selectors      */
      IntArray,            /* integer attributes               */
      48L,                 /* length of character attributes   */
      CharArray,           /* character attributes             */
      &CC,&Reason);        /* completion and reason codes      */
if (CC == MQCC_FAILED) {
    printf("MQINQ failed with reason code %d\n", Reason);
}
else {
      QMgrCCSID = IntArray[0];
      memcpy(QMName,CharArray,MQ_Q_MGR_NAME_LENGTH);
      printf ("CCSID=%ld QMGR=%s<\n",QMgrCCSID, QMName);
}

MQCLOSE(Hcon, &Hobj, MQCO_NONE, &CC, &Reason);
if (CC == MQCC_FAILED) {
    printf("MQCLOSE after MQINQ failed with %d\n", Reason);
    goto PGM_DISC;
}
if (QMgrCCSID == -1) goto PGM_DISC;

/*****************************************************************/
/*   Build the reference message                                 */
/*****************************************************************/
refx.ref.StrucLength    = sizeof(refx);
refx.ref.Encoding       = MQENC_NATIVE;
refx.ref.CodedCharSetId = QMgrCCSID;
refx.ref.Flags          = MQRMHF_LAST;
memcpy(refx.ref.Format, MQFMT_STRING, (size_t)MQ_FORMAT_LENGTH);

memcpy(refx.ref.ObjectType,"FLATFILE",sizeof(refx.ref.ObjectType));

memset(refx.SrcName,' ',sizeof(refx.SrcName)+sizeof(refx.DestName));

memcpy(refx.SrcName,infile,strlen(infile));
memcpy(refx.DestName,outfile,strlen(outfile));

refx.ref.SrcNameLength = strlen(infile);
refx.ref.SrcNameOffset = offsetof(MQRMHX,SrcName);
```

Exercise 12: PUTREF.C

```
   refx.ref.DestNameLength = strlen(outfile);
   refx.ref.DestNameOffset = offsetof(MQRMHX,DestName);
   /******************************************************************/
   /*   Put reference message on queue                               */
   /******************************************************************/
               /****************************************************/
               /* Set up object descriptor, pmo, and message header  */
               /****************************************************/
   od.ObjectType = MQOT_Q;

   strncpy(od.ObjectName, "REFMSG", sizeof(od.ObjectName));

   strncpy(od.ObjectQMgrName, "QMGR2", sizeof(od.ObjectQMgrName));

   pmo.Options = MQPMO_FAIL_IF_QUIESCING;

   memcpy(md.Format,MQFMT_REF_MSG_HEADER,(size_t)MQ_FORMAT_LENGTH);

   md.MsgType = MQMT_DATAGRAM;
               /****************************************************/
               /* Use MQPUT1 to put the message to the xmitq        */
               /****************************************************/
   MQPUT1(Hcon,                      /* connection handle           */
          &od,                       /* object descriptor for queue  */
          &md,                       /* message descriptor          */
          &pmo,                      /* options                     */
          sizeof(refx),              /* buffer length               */
          &refx,                     /* buffer                      */
          &CC, &Reason);             /* completion and reason codes  */

   if (Reason != MQRC_NONE)
       printf("MQPUT1 ended with reason code %d\n", Reason);

   /******************************************************************/
   /*   Disconnect from queue manager and end program                */
   /******************************************************************/
PGM_DISC:
   MQDISC(&Hcon,                    /* connection handle            */
          &CC,&Reason);             /* completion and reason codes   */
   if (Reason != MQRC_NONE) {
       printf("MQDISC ended with reason code %d\n", Reason);
   }
   printf("End of PUTMSG\n");
   return(0);
 }
```

E.2 Source of GETREF

```
/******************************************************************/
/*                                                                */
/* GETREF: Get a Reference Message                                */
/*                                                                */
/******************************************************************/
#include <stdio.h>
#include <stdlib.h>
#include <stddef.h>
#include <string.h>
#include <ctype.h>
#include <cmqc.h>
/******************************************************************/
/*                         Variables                              */
/******************************************************************/
  MQHCONN  Hcon;                 /* connection handle            */
  MQHOBJ   Hobj;                 /* object handle                */
  MQLONG   CC = MQCC_OK;         /* completion code              */
  MQLONG   Reason;               /* reason code                  */
  MQLONG   CompCode = MQCC_OK;   /* completion code              */
  MQLONG   oo;                   /* MQOPEN options               */
  MQOD     od = {MQOD_DEFAULT};  /* Object Descriptor            */
  MQMD     md = {MQMD_DEFAULT};  /* Message Descriptor           */
  MQGMO    gmo = {MQGMO_DEFAULT};/* get message options          */
  char     Buffer[1000];
  MQLONG   DataLen;              /* length of message            */
  char     Filename[256];
  FILE    *File;                 /* file structure               */
  MQRMH   *pMQRMH;               /* Pointer to MQRMH structure   */
  char    *pObjectName;          /* Object name                  */
  char     ObjectType[sizeof(MQCHAR8)+1];
  /****************************************************************/
  /*                                                              */
  /*                         Program                              */
  /*                                                              */
  /****************************************************************/
int main(int argc, char **argv)
{
    printf("Start GETREF\n");

  /****************************************************************/
  /*   Connect to queue manager QMGR2                             */
  /****************************************************************/
    MQCONN("QMGR2",              /* queue manager                */
           &Hcon,                /* connection handle            */
           &CC, &Reason);        /* completion and reason codes  */
    if (CC == MQCC_FAILED) {
```

```
      printf("MQCONN ended with %d\n", Reason);
      return(1);
  }
/****************************************************************/
/*  Open the queue REFMSG                                       */
/****************************************************************/
  strncpy(od.ObjectName,"REFMSG",(size_t)MQ_Q_NAME_LENGTH);
  oo = MQOO_FAIL_IF_QUIESCING +
       MQOO_INPUT_AS_Q_DEF;

  MQOPEN(Hcon                     /* connection handle            */
        ,&od                      /* object descriptor for queue  */
        ,oo                       /* options                      */
        ,&Hobj                    /* object handle                */
        ,&CC, &Reason);

  if (CC == MQCC_FAILED) {
      printf("MQOPEN ended with %d\n",Reason);
      goto PGM_DISC;
  }
/****************************************************************/
/* Get one message from the queue                               */
/****************************************************************/
  gmo.Options = MQGMO_WAIT +
                MQGMO_CONVERT +
                MQGMO_ACCEPT_TRUNCATED_MSG;

  gmo.WaitInterval = 5000;          /*  5 seconds wait interval   */

  memcpy(md.MsgId,MQMI_NONE,sizeof(md.MsgId));
  memcpy(md.CorrelId,MQCI_NONE,sizeof(md.CorrelId));

  md.Encoding       = MQENC_NATIVE;
  md.CodedCharSetId = MQCCSI_Q_MGR;
  printf("Wait up to 5 seconds...\n");
  MQGET(Hcon,Hobj,               /* connection and queue handle   */
        &md,                     /* message descriptor            */
        &gmo,                    /* get options                   */
        sizeof(Buffer),          /* buffer size                   */
        &Buffer,                 /* buffer address                */
        &DataLen,                /* data length (output)          */
        &CC,&Reason);
  if (CC == MQCC_FAILED) {
     if (Reason == MQRC_NO_MSG_AVAILABLE)
        printf("No message available\n");
     else
        printf("MQGET failed with %d\n", Reason);
     goto PGM_CLOSE;
```

```
    }

    if (memcmp(md.Format,MQFMT_REF_MSG_HEADER,(size_t)MQ_FORMAT_LENGTH))
    {
       printf("Not a reference message, format=%s\n",md.Format);
       goto PGM_CLOSE;
    }

    pMQRMH = (MQRMH*)&Buffer;      /* overlay MQRMH on MQGET buffer */

          /*********************************************************/
          /* Extract fully qualified name from MQRMH structure.    */
          /*********************************************************/
    pObjectName = (char*)&Buffer + pMQRMH -> DestNameOffset;
    memset(Filename,0,sizeof(Filename));
    strncpy(Filename,pObjectName,
            ((size_t)(pMQRMH->DestNameLength) >= sizeof(Filename))
                  ? (size_t)(sizeof(Filename) -1)
                  : (size_t)(pMQRMH -> DestNameLength));

          /*********************************************************/
          /* Extract object type from MQRMH structure              */
          /*********************************************************/
    memset(ObjectType,0,sizeof(ObjectType));
    strncpy(ObjectType,pMQRMH->ObjectType,sizeof(pMQRMH->ObjectType));

          /*********************************************************/
          /* Check if file exists                                  */
          /*********************************************************/
    File = fopen(Filename,"r");

    if (File == NULL) {
        printf("File %s of type %s could not be found\n",
                Filename,ObjectType);
    }
    else {
          printf("File name is %.48s\n",Filename,ObjectType);
          fclose(File);
    }
  /******************************************************************/
  /* Close the queue                                                */
  /******************************************************************/
PGM_CLOSE:
    MQCLOSE(Hcon,&Hobj,MQCO_NONE, &CC, &Reason);

    if (CC == MQCC_FAILED)
        printf("MQCLOSE ended with %d\n", Reason);
```

Exercise 12: GETREF.C

```
   /****************************************************************/
   /*   Disconnect from queue manager and end the program          */
   /****************************************************************/
PGM_DISC:
     MQDISC(&Hcon,&CC, &Reason);

     if (CC == MQCC_FAILED)
         printf("MQDISC ended with %d\n", Reason);

     printf("GETREF ends\n");
     return (0);
 }
```

Appendix F. Distribution List Example

```
 /*-----------------------------------------------------------------*/
 /*                                                                 */
 /* Program Name : Distl.c                                          */
 /*                                                                 */
 /*-----------------------------------------------------------------*/

/* ------------------------------------------------------------------------
   Include Header Files.
   ------------------------------------------------------------------------ */
 #include <stdio.h>
 #include <stdlib.h>
 #include <string.h>
 #include <cmqc.h>                                 /* includes for MQI */

/* ------------------------------------------------------------------------
   Constant value definitions
   ------------------------------------------------------------------------ */
#define DIST_LIST_LENGTH    10
#define MAX_NAME_LENGTH     40

/* ------------------------------------------------------------------------
   Function Prototypes
   ------------------------------------------------------------------------ */
static int ReadDistList(void);
static void print_usage(void);
static void print_responses( char * comment
                           , PMQRR  pRR
                           , MQLONG NumQueues
                           , PMQOR  pOR);

/* ------------------------------------------------------------------------
   Global variable definitions.
   ------------------------------------------------------------------------ */
static struct ObjectInfoType{              /* struct to hold the queue and */
    char ObjName[MAX_NAME_LENGTH];         /* the queue manager name       */
    char ObjQMgrName[MAX_NAME_LENGTH];
};

struct ObjectInfoType DistList[DIST_LIST_LENGTH];
                                           /* array to hold the dist. list */

/*-----------------------------------------------------------------*/
/* This function is to read the target queue and queue manager      */
/* names from the distlist.txt file and place them into the array   */
/*-----------------------------------------------------------------*/
```

Exercise 13: DISTL.C

```
static int ReadDistList()
{
  int i=0;
  FILE *dl;           /* Distribution List */

  if (NULL == (dl = fopen("DistList.txt","r")))
     printf("\n Unable to open the data file !!");
  else
  {
    while (!feof(dl)) {
      fgets(DistList[i].ObjName,MAX_NAME_LENGTH,dl);
      DistList[i].ObjName[strlen(DistList[i].ObjName)-1] = '\0';

      fgets(DistList[i].ObjQMgrName,MAX_NAME_LENGTH,dl);
      DistList[i].ObjQMgrName[strlen(DistList[i].ObjQMgrName)-1] = '\0';
      i += 1;
    }
    i -= 1;
    fclose(dl);
  }

  return(i);
}
/* --------------------------------------------------------------------------
   Main C function
   -------------------------------------------------------------------------- */
 int main(int argc, char **argv)
 {
   typedef enum {False, True} Bool;

   /*   Declare file and character for sample input                  */
   FILE *fp;

   /*   Declare MQI structures needed                                */
   MQOD     od = {MQOD_DEFAULT};    /* Object Descriptor             */
   MQMD     md = {MQMD_DEFAULT};    /* Message Descriptor            */
   MQPMO   pmo = {MQPMO_DEFAULT};   /* put message options           */
      /** note, the program uses defaults where it can **/

   MQHCONN  Hcon;                   /* connection handle             */
   MQHOBJ   Hobj;                   /* object handle                 */
   MQLONG   O_options;              /* MQOPEN options                */
   MQLONG   C_options;              /* MQCLOSE options               */
   MQLONG   CompCode;               /* completion code               */
   MQLONG   OpenCode;               /* MQOPEN completion code        */
   MQLONG   Reason;                 /* reason code                   */
   MQLONG   buflen;                 /* buffer length                 */
   char     buffer[100];            /* message buffer                */
```

```
MQLONG   Index ;                  /* Index into list of queues     */
MQLONG   NumQueues ;              /* Number of queues              */

PMQRR    pRR=NULL;                /* Pointer to response records   */
PMQOR    pOR=NULL;                /* Pointer to object records     */
Bool     DisconnectRequired=False;/* Already connected switch      */
Bool     Connected=False;         /* Connect succeeded switch     */

/*----------------------------------------------------------------*/
/* The use of Put Message Records (PMR's) allows some message     */
/* attributes to be specified on a per destination basis. These   */
/* attributes then override the values in the MD for a particular */
/* destination.                                                   */
/* The function provided by this program does not require         */
/* the use of PMR's but they are used by the program simply to    */
/* demonstrate their use.                                         */
/* The program chooses to provide values for MsgId and CorrelId   */
/* on a per destination basis.                                    */
/*----------------------------------------------------------------*/
typedef struct
{
 MQBYTE24 MsgId;
 MQBYTE24 CorrelId;
} PutMsgRec, *pPutMsgRec;
pPutMsgRec pPMR=NULL;             /* Pointer to put msg records    */

/*----------------------------------------------------------------*/
/* The PutMsgRecFields in the PMO indicates what fields are in    */
/* the array addressed by PutMsgRecPtr in the PMO.                */
/* In our example we have provided the MsgId and CorrelId and so  */
/* we must set the corresponding MQPMRF_... bits.                 */
/*----------------------------------------------------------------*/
MQLONG PutMsgRecFields=MQPMRF_MSG_ID ] MQPMRF_CORREL_ID;

/* Read the targer queues from the text file. */
/* Number of Queue/QueueMgr name pairs        */

NumQueues = ReadDistList();

/*----------------------------------------------------------------*/
/*   Allocate response records, object records and put message    */
/*   records. (new MQSeries structures)                           */
/*----------------------------------------------------------------*/
pRR  = (PMQRR)malloc( NumQueues * sizeof(MQRR));
pOR  = (PMQOR)malloc( NumQueues * sizeof(MQOR));
pPMR = (pPutMsgRec)malloc( NumQueues * sizeof(PutMsgRec));
```

Exercise 13: DISTL.C

```
if((NULL == pRR) ]] (NULL == pOR) ]] (NULL == pPMR))
{
  printf("%s(%d) malloc failed\n", __FILE__, __LINE__);
  exit(99);
}
/*-----------------------------------------------------------------*/
/*                                                                 */
/*   Copy the queue list into the MQOR structure                   */
/*                                                                 */
/*-----------------------------------------------------------------*/
for( Index = 0 ; Index < NumQueues ; Index ++)
{
  strncpy( (pOR+Index)->ObjectName
         , DistList[Index].ObjName
         , (size_t)MQ_Q_NAME_LENGTH);
  strncpy( (pOR+Index)->ObjectQMgrName
         , DistList[Index].ObjQMgrName
         , (size_t)MQ_Q_MGR_NAME_LENGTH);
}
/*-----------------------------------------------------------------*/
/*   Connect to queue manager                                      */
/*                                                                 */
/*   Try to connect to the queue manager associated with the      */
/*   first queue, if that fails then try each of the other        */
/*   queue managers in turn.                                       */
/*                                                                 */
/*-----------------------------------------------------------------*/
for( Index = 0 ; Index < NumQueues ; Index ++)
{
  MQCONN((pOR+Index)->ObjectQMgrName, /* queue manager            */
         &Hcon,                   /* connection handle            */
         &((pRR+Index)->CompCode),/* completion code              */
         &((pRR+Index)->Reason)); /* reason code                  */

  if ((pRR+Index)->CompCode == MQCC_FAILED)
  {
    continue;
  }
  if ((pRR+Index)->CompCode == MQCC_OK)
  {
    DisconnectRequired = True ;
  }
  Connected = True;
  break ;
}
/*-----------------------------------------------------------------*/
/* Print any non zero responses                                    */
/*-----------------------------------------------------------------*/
```

```
print_responses("MQCONN", pRR, Index, pOR);

/*----------------------------------------------------------------*/
/* If we failed to connect to any queue manager then exit.        */
/*----------------------------------------------------------------*/
if( False == Connected  )
{
  printf("unable to connect to any queue manager\n");
  exit(99) ;
}
/*----------------------------------------------------------------*/
/*                                                                */
/*   Open the target message queue for output                     */
/*                                                                */
/*----------------------------------------------------------------*/
od.Version = MQOD_VERSION_2 ;
od.RecsPresent = NumQueues ;      /* number of object/resp recs   */
od.ObjectRecPtr = pOR;            /* address of object records    */
od.ResponseRecPtr = pRR ;         /* Number of object records     */
O_options = MQOO_OUTPUT           /* open queue for output        */
        + MQOO_FAIL_IF_QUIESCING; /* but not if MQM stopping      */
MQOPEN(Hcon,                      /* connection handle            */
       &od,                       /* object descriptor for queue  */
       O_options,                 /* open options                 */
       &Hobj,                     /* object handle                */
       &OpenCode,                 /* MQOPEN completion code       */
       &Reason);                  /* reason code                  */

/*----------------------------------------------------------------*/
/* report reason(s) if any; stop if failed.                       */
/*                                                                */
/* Note: The reasons in the response records are only valid if    */
/*       the MQI Reason is MQRC_MULTIPLE_REASONS. If any other    */
/*       reason is reported then all destinations in the list     */
/*       completed/failed with the same reason.                   */
/*       If the MQI CompCode is MQCC_FAILED then all of the       */
/*       destinations in the list failed to open. If some         */
/*       destinations opened and others failed to open then       */
/*       the response will be set to MQCC_WARNING.                */
/*                                                                */
/*----------------------------------------------------------------*/
if (Reason == MQRC_MULTIPLE_REASONS)
{
  print_responses("MQOPEN", pRR, NumQueues, pOR);
}
else
{
  if (Reason != MQRC_NONE)
```

Exercise 13: DISTL.C

```
    {
      printf("MQOPEN  returned CompCode=%ld, Reason=%ld\n"
            , OpenCode
            , Reason);
    }
  }

  if (OpenCode == MQCC_FAILED)
  {
    printf("unable to open any queue for output\n");
  }
  /*----------------------------------------------------------------*/
  /*                                                                */
  /*   Read lines from the file and put them to the message queue   */
  /*   Loop until null line or end of file, or there is a failure   */
  /*                                                                */
  /*----------------------------------------------------------------*/
  CompCode = OpenCode;        /* use MQOPEN result for initial test */

  fp = stdin;

  pmo.Version = MQPMO_VERSION_2 ;
  pmo.RecsPresent = NumQueues ;
  pmo.PutMsgRecPtr = pPMR ;
  pmo.PutMsgRecFields = PutMsgRecFields ;
  pmo.ResponseRecPtr = pRR ;
  while (CompCode != MQCC_FAILED)
  {
    if (fgets(buffer, sizeof(buffer), fp) != NULL)
    {
      buflen = strlen(buffer);       /* length without null         */
      if (buffer[buflen-1] == '\n')  /* last char is a new-line     */
      {
        buffer[buflen-1]  = '\0';    /* replace new-line with null  */
        --buflen;                    /* reduce buffer length        */
      }
    }
    else buflen = 0;        /* treat EOF same as null line          */
    /*--------------------------------------------------------------*/
    /*                                                              */
    /*   Put each buffer to the message queue                       */
    /*                                                              */
    /*--------------------------------------------------------------*/
    if (buflen > 0)
    {
      for( Index = 0 ; Index < NumQueues ; Index ++)
      {
        memcpy( (pPMR+Index)->MsgId
```

```
            , MQMI_NONE
            , sizeof((pPMR+Index)->MsgId));
      memcpy( (pPMR+Index)->CorrelId
            , MQCI_NONE
            , sizeof((pPMR+Index)->CorrelId));
    }
    memcpy(md.Format,          /* character string format         */
           MQFMT_STRING, (size_t)MQ_FORMAT_LENGTH);
    MQPUT(Hcon,                /* connection handle               */
          Hobj,                /* object handle                   */
          &md,                 /* message descriptor              */
          &pmo,                /* default options (datagram)      */
          buflen,              /* buffer length                   */
          buffer,              /* message buffer                  */
          &CompCode,           /* completion code                 */
          &Reason);            /* reason code                     */
/*-----------------------------------------------------------------*/
/* report reason(s) if any; stop if failed.                        */
/*-----------------------------------------------------------------*/
    if (Reason == MQRC_MULTIPLE_REASONS)
    {
      print_responses("MQPUT", pRR, NumQueues, pOR);
    }
    else
    {
      if (Reason != MQRC_NONE)
      {
        printf("MQPUT  returned CompCode=%ld, Reason=%ld\n"
              , OpenCode
              , Reason);
      }
    }
  }
  else   /* satisfy end condition when empty line is read */
    CompCode = MQCC_FAILED;
}
/*-----------------------------------------------------------------*/
/*                                                                 */
/*   Close the target queue (if it was opened)                     */
/*                                                                 */
/*-----------------------------------------------------------------*/
if (OpenCode != MQCC_FAILED)
{
  C_options = 0;                  /* no close options              */
  MQCLOSE(Hcon,                   /* connection handle             */
          &Hobj,                   /* object handle                */
          C_options,
          &CompCode,               /* completion code              */
```

```
             &Reason);                /* reason code                 */

      /* report reason, if any     */
      if (Reason != MQRC_NONE)
      {
        printf("MQCLOSE ended with reason code %ld\n", Reason);
      }
    }
    /*----------------------------------------------------------------*/
    /*                                                                */
    /*   Disconnect from MQM if not already connected                 */
    /*                                                                */
    /*----------------------------------------------------------------*/
    if (DisconnectRequired==True)
    {
      MQDISC(&Hcon,                   /* connection handle           */
             &CompCode,               /* completion code             */
             &Reason);                /* reason code                 */

      /* report reason, if any     */
      if (Reason != MQRC_NONE)
      {
        printf("MQDISC ended with reason code %ld\n", Reason);
      }
    }
    /*----------------------------------------------------------------*/
    /*                                                                */
    /* END OF PROGRAM                                                 */
    /*                                                                */
    /*----------------------------------------------------------------*/
    if( NULL != pOR )
    {
      free( pOR ) ;
    }
    if( NULL != pRR )
    {
      free( pRR ) ;
    }
    if( NULL != pPMR )
    {
      free( pPMR ) ;
    }
    return(0);
  }
/*----------------------------------------------------------------*/
/*                                                                */
/* Function: Print MQI responses from the ResponseRecord array.   */
/*                                                                */
```

```
/* Notes:    This function is typically called when a reason of     */
/*           MQRC_MULTIPLE_REASONS is received.                     */
/*           The reasons relate to the queue at the equivalent      */
/*           ordinal position in the MQOR array.                    */
/*------------------------------------------------------------------*/
static void print_responses( char * comment
                   , PMQRR  pRR
                   , MQLONG NumQueues
                   , PMQOR  pOR)
{
  MQLONG Index;
  for( Index = 0 ; Index < NumQueues ; Index ++ )
  {
    if( MQCC_OK != (pRR+Index)->CompCode )
    {
      printf("%s for %.48s( %.48s) returned CompCode=%ld, Reason=%ld\n"
            , comment
            , (pOR+Index)->ObjectName
            , (pOR+Index)->ObjectQMgrName
            , (pRR+Index)->CompCode
            , (pRR+Index)->Reason);
    }
  }
}
```

Exercise 13: DISTL.C

Appendix G. Fastpath Bindings Example

```
/*----------------------------------------------------------------*/
/*                                                                */
/* Program name: CONNX.C                                          */
/*                                                                */
/*----------------------------------------------------------------*/
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <time.h>
#include <cmqc.h>

int main(int argc, char **argv)
{
  /*   Declare MQI structures needed                              */
  MQOD     od = {MQOD_DEFAULT};    /* Object Descriptor           */
  MQMD     md = {MQMD_DEFAULT};    /* Message Descriptor          */
  MQPMO   pmo = {MQPMO_DEFAULT};   /* put message options         */

  MQHCONN  Hcon;                   /* connection handle           */
  MQHOBJ   Hobj;                   /* object handle               */
  MQLONG   O_options;              /* MQOPEN options              */
  MQLONG   C_options;              /* MQCLOSE options             */
  MQLONG   CompCode;               /* completion code             */
  MQLONG   OpenCode;               /* MQOPEN completion code      */
  MQLONG   Reason;                 /* reason code                 */
  MQLONG   CReason;                /* reason code for MQCONN      */
  MQLONG   buflen;                 /* buffer length               */
  char     buffer[100];            /* message buffer              */
  char     QMName[50];             /* queue manager name          */

  int      NumOfMsgs;
  int      Count;
  double   Time1, Time2, Timediff;
  char     Str[5];

  MQCNO    ConnectOpt;                 /* Options to control the CONNX */

  /*--------------------------------------------------------------*/
  /*                                                              */
  /*   Try to get the number of messages to be put                */
  /*                                                              */
  /*--------------------------------------------------------------*/
  if (argc == 2)
  {
    NumOfMsgs = atoi(argv[1]);
```

Exercise 14: CONNX.C

```
    QMName[0] = 0;
  }
  else if (argc == 3)
  {
    NumOfMsgs = atoi(argv[1]);
    strcpy (QMName, argv[2]);
  }
  else {
    printf("\nInvalid number of Parameters");
    printf("\nUsage : <Program Name> <Number of Msgs> [QMgrName]");
    exit(99);
  }
  /*----------------------------------------------------------------*/
  /*                                                                */
  /*   Connect to queue manager                                     */
  /*                                                                */
  /*----------------------------------------------------------------*/

  strcpy(ConnectOpt.StrucId, MQCNO_STRUC_ID);
  ConnectOpt.Version = MQCNO_VERSION_1;
  ConnectOpt.Options = MQCNO_FASTPATH_BINDING ;

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

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

  /*----------------------------------------------------------------*/
  /*                                                                */
  /*   Specify the target output queue                              */
  /*                                                                */
  /*----------------------------------------------------------------*/
  strncpy(od.ObjectName, "INPUT.QUEUE", (size_t)MQ_Q_NAME_LENGTH);
  printf("Target queue is %s\n", od.ObjectName);

  /*----------------------------------------------------------------*/
  /*                                                                */
  /*   Open the target message queue for output                     */
  /*                                                                */
  /*----------------------------------------------------------------*/
```

```
 O_options = MQOO_OUTPUT           /* open queue for output        */
         + MQOO_FAIL_IF_QUIESCING; /* but not if MQM stopping      */
 MQOPEN(Hcon,                      /* connection handle            */
        &od,                       /* object descriptor for queue  */
        O_options,                 /* open options                 */
        &Hobj,                     /* object handle                */
        &OpenCode,                 /* MQOPEN completion code       */
        &Reason);                  /* reason code                  */

 /* report reason, if any; stop if failed      */
 if (Reason != MQRC_NONE)
 {
   printf("MQOPEN ended with reason code %ld\n", Reason);
 }

 if (OpenCode == MQCC_FAILED)
 {
   printf("unable to open queue for output\n");
 }

 CompCode = OpenCode;        /* use MQOPEN result for initial test */

 memcpy(md.Format,           /* character string format         */
        MQFMT_STRING, (size_t)MQ_FORMAT_LENGTH);


 strcpy(buffer, "This is a test message.");
 buflen = strlen(buffer);       /* length without null         */

/*--------------------------------------------------------------*/
/*                                                              */
/* Setting the time before doing the MQPUT messages             */
/*                                                              */
/*--------------------------------------------------------------*/
 Time1 = (double) clock();
 Time1 = Time1/CLOCKS_PER_SEC;

 for (Count=1; Count<=NumOfMsgs; Count++)
 {
   /*--------------------------------------------------------------*/
   /*                                                              */
   /* Trying to put n number of messages on the queue              */
   /*                                                              */
   /*--------------------------------------------------------------*/

   memcpy(md.MsgId,           /* reset MsgId to get a new one    */
          MQMI_NONE, sizeof(md.MsgId) );
```

Exercise 14: CONNX.C

```
    memcpy(md.CorrelId,        /* reset CorrelId to get a new one */
           MQCI_NONE, sizeof(md.CorrelId) );

    MQPUT(Hcon,                /* connection handle               */
          Hobj,                /* object handle                   */
          &md,                 /* message descriptor              */
          &pmo,                /* default options (datagram)      */
          buflen,              /* buffer length                   */
          buffer,              /* message buffer                  */
          &CompCode,           /* completion code                 */
          &Reason);            /* reason code                     */

    /* report reason, if any */
    if (Reason != MQRC_NONE)
    {
      printf("MQPUT ended with reason code %ld\n", Reason);
    }
  }
 /*-----------------------------------------------------------------*/
 /*                                                                 */
 /* Getting the time after doing n MQPUT                            */
 /*                                                                 */
 /*-----------------------------------------------------------------*/
  Time2 = (double) clock();
  Time2 = Time2/CLOCKS_PER_SEC;

  Timediff = Time2-Time1;
  printf("\nThe elapsed time = %f seconds.", Timediff);

  /*------------------------------------------------------------------*/
  /*                                                                  */
  /*   Close the target queue (if it was opened)                      */
  /*                                                                  */
  /*------------------------------------------------------------------*/
  if (OpenCode != MQCC_FAILED)
  {
    C_options = 0;                 /* no close options             */
    MQCLOSE(Hcon,                  /* connection handle            */
            &Hobj,                  /* object handle                */
            C_options,
            &CompCode,              /* completion code              */
            &Reason);               /* reason code                  */

    /* report reason, if any     */
    if (Reason != MQRC_NONE)
    {
      printf("MQCLOSE ended with reason code %ld\n", Reason);
    }
```

```
  }
  /*----------------------------------------------------------------*/
  /*                                                                */
  /*   Disconnect from MQM if not already connected                 */
  /*                                                                */
  /*----------------------------------------------------------------*/
  if (CReason != MQRC_ALREADY_CONNECTED)
  {
    MQDISC(&Hcon,                   /* connection handle            */
           &CompCode,               /* completion code              */
           &Reason);                /* reason code                  */

    /* report reason, if any     */
    if (Reason != MQRC_NONE)
    {
      printf("MQDISC ended with reason code %ld\n", Reason);
    }
  }
  return(0);
}
```

Exercise 14: CONNX.C

Appendix H. Diskette Contents

The diskette contains the examples developed in this book. Table 32 lists the directories and the file names.

Table 32 (Page 1 of 3). Files on Diskette

File name	Description
\dbsetup	**2.5, "Exercise 1: Setup for XA Coordination" on page 27**
data.sql	Data to populate the databases
db.sql	Create databases (NT)
dbcreate.sql	Create databases (AIX)
dbdrop.sql	Drop the databases
db2swit.c	XA switch source code
db2swit.def	XA switch definition
db2swit.dll	XA switch for NT (4096 bytes)
grant.sql	Grant database access to other users
select.sql	Look at the contents of the database tables
tbldrop.sql	Drop the tables in the databases
util.c	From \SQLLIB\samples\c
util.h	From \SQLLIB\samples\c
xa.h	XA switch header
xaswit.mak	Make file XA switch for MS compiler
xaswiti.mak	Make file XA switch for IBM compiler
\DBex1	**2.7, "Exercise 2: Using One XA Resource" on page 40**
amqsxas0.sqc	Source for database update program
util.obj	Object file for NT
qm.ini	Sample qm.ini with one XA resource stanza
ibmmake.bat	Make file for IBM compiler
msmake.bat	Make file for Microsoft compiler
amqsxas0.sh	Shell for compile under AIX
amqsxas0.mak	Make file for compile under AIX
\DBex2	**2.8, "Exercise 3: Understanding Backout" on page 50**
amqsxas1.sqc	Source file for database update program
util.obj	Object file for NT

Table 32 (Page 2 of 3). Files on Diskette	
File name	**Description**
ibmmake.bat	Make file for IBM compiler
msmake.bat	Make file for Microsoft compiler
amqsxas1.sh	Shell for compile under AIX
amqsxas1.mak	Make file for compile under AIX
\DBex3	**2.9, "Exercise 4: Using Two XA Resources" on page 58**
amqsxab0.sqc	Contains routines to access MQBankDB
amqsxaf0.sqc	Contains routines to access MQFeeDB
amqsxag0.c	Main program
amqsxag0.exe	Executable
amqsxag0.mak	Make file for AIX
amqsxag0.sh	Shell file for AIX
ibmmake.bat	Compile the SQC files with IBM compiler (NT)
ibmmake2.bat	Compile and link C file with IBM compiler (NT)
msmake.bat	Compile the SQC files with MS compiler (NT)
msmake2.bat	Compile and link C file with MS compiler (NT)
select.sql	SQL file to view databases
util.obj	Object file for NT
\DBex4	**2.10, "Exercise 5: Configuration Issues" on page 63**
amqsxas2.sqc	Source code
ibmmake.bat	Build executable with IBM compiler for NT
msmake.bat	Build executable with MS compiler for NT
util.obj	Object file for NT
\Exer1	**3.4, "Exercise 6: Arbitrary Segmentation" on page 75**
bcg_seg1.c	Program demonstrating arbitrary segmentation
big.c	Program that creates a 'very large file'
put_seg1.c	Program that reassembles a logical message
\Exer2	**3.5, "Exercise 7: Application Segmentation" on page 82**
bcg_seg2.c	Program demonstrating application segmentation
\Exer3	**4.4, "Exercise 8: Putting Message Groups" on page 92**
put_grp1.c	Program that puts messages in a group
bcg_grp1.c	Program that reads messages of a group after all messages of the group have arrived

Table 32 (Page 3 of 3). Files on Diskette	
File name	**Description**
\Secu	**5.4, "Exercise 9: Remote Administration in One Machine" on page 107**
qmgr1.in	Objects for queue manager QMGR1
qmgr2.in	Objects for queue manager QMGR2
startup1.cmd	Startup commands for QMGR1
startup2.cmd	Startup commands for QMGR2
\Refmsg	**6.3, "Exercise 12: Building a Reference Message" on page 126**
dw.fil	The file to be transmitted
getref.c	Program that gets the reference message
getref.exe	Executable
putref.c	Program that builds and sends the reference message
putref.exe	Executable
qmgr1.in	Objects for queue manager QMGR1
qmgr2.in	Objects for queue manager QMGR2
\Distl	**7.6, "Exercise 13: Distribution List" on page 150**
distl.c	Program that uses a distribution list
distlist.txt	File that contain the distribution list
distl.tst	Queue definitions
\Connx	**8.1, "Exercise 14: Using Fastpath Bindings" on page 160**
conn.c	Program that measures time using standard bindings
conn.exe	Executable
connx.c	Program that measures time using fastpath bindings
connx.exe	Executable

Appendix I. Special Notices

This publication is intended to help application programmers to use the functions provided with the MQSeries Version 5 products. The information in this publication is not intended as the specification of any programming interfaces that are provided by MQSeries for OS/2 Version 5, MQSeries for AIX Version 5 and MQSeries for Windows NT Version 5. See the PUBLICATIONS section of the IBM Programming Announcement for MQSeries Version 5 for more information about what publications are considered to be product documentation.

References in this publication to IBM products, programs or services do not imply that IBM intends to make these available in all countries in which IBM operates. Any reference to an IBM product, program, or service is not intended to state or imply that only IBM′s product, program, or service may be used. Any functionally equivalent program that does not infringe any of IBM′s intellectual property rights may be used instead of the IBM product, program or service.

Information in this book was developed in conjunction with use of the equipment specified, and is limited in application to those specific hardware and software products and levels.

IBM may have patents or pending patent applications covering subject matter in this document. The furnishing of this document does not give you any license to these patents. You can send license inquiries, in writing, to the IBM Director of Licensing, IBM Corporation, North Castle Drive, Armonk, NY 10504-1785.

Licensees of this program who wish to have information about it for the purpose of enabling: (i) the exchange of information between independently created programs and other programs (including this one) and (ii) the mutual use of the information which has been exchanged, should contact IBM Corporation, Dept. 600A, Mail Drop 1329, Somers, NY 10589 USA.

Such information may be available, subject to appropriate terms and conditions, including in some cases, payment of a fee.

The information contained in this document has not been submitted to any formal IBM test and is distributed AS IS. 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 be obtained elsewhere. Customers attempting to adapt these techniques to their own environments do so at their own risk.

Any pointers in this publication to external Web sites are provided for convenience only and do not in any manner serve as an endorsement of these Web sites.

The following document contains examples of data and reports used in daily business operations. To illustrate them as completely as possible, the examples contain the names of individuals, companies, brands, and products. All of these names are fictitious and any similarity to the names and addresses used by an actual business enterprise is entirely coincidental.

The following terms are trademarks of the International Business Machines Corporation in the United States and/or other countries:

ADSTAR	AIX
AS/400	CICS
DB2	DB2 Connect
DB2 Universal Database	DRDA
FFST	IBM
MQ	MQSeries
MVS	MVS/ESA
OS/2	VisualAge

The following terms are trademarks of other companies:

C-bus is a trademark of Corollary, Inc.

Java and HotJava are trademarks of Sun Microsystems, Incorporated.

Microsoft, Windows, Windows NT, and the Windows 95 logo are trademarks or registered trademarks of Microsoft Corporation.

PC Direct is a trademark of Ziff Communications Company and is used by IBM Corporation under license.

Pentium, MMX, ProShare, LANDesk, and ActionMedia are trademarks or registered trademarks of Intel Corporation in the U.S. and other countries.

UNIX is a registered trademark in the United States and other countries licensed exclusively through X/Open Company Limited.

Appendix J. Related Publications

The publications listed in this section are considered particularly suitable for a more detailed discussion of the topics covered in this redbook.

J.1 International Technical Support Organization Publications

For information on ordering these ITSO publications see "How to Get ITSO Redbooks" on page 271.

- *MQSeries Backup and Recovery*, SG24-5222
- *Using MQSeries on the AS/400*, SG24-5236
- *Connecting the Enterprise to the Internet with MQSeries and VisualAge for Java*, SG24-2144
- *Examples of Using the TME 10 Module for MQSeries*, SG24-2134
- *MQSeries for Windows Version 2.1 in a Mobile Environment*, SG24-2103
- *Application Developoment with VisualAge for Smalltalk and MQSeries*, SG24-2117
- *Internet Application Development with MQSeries and Java*, SG24-4896
- *Examples of Using MQSeries on WWW*, SG24-4882

J.2 Redbooks on CD-ROMs

Redbooks are also available on CD-ROMs. **Order a subscription** and receive updates 2-4 times a year at significant savings.

CD-ROM Title	Subscription Number	Collection Kit Number
System/390 Redbooks Collection	SBOF-7201	SK2T-2177
Networking and Systems Management Redbooks Collection	SBOF-7370	SK2T-6022
Transaction Processing and Data Management Redbook	SBOF-7240	SK2T-8038
Lotus Redbooks Collection	SBOF-6899	SK2T-8039
Tivoli Redbooks Collection	SBOF-6898	SK2T-8044
AS/400 Redbooks Collection	SBOF-7270	SK2T-2849
RS/6000 Redbooks Collection (HTML, BkMgr)	SBOF-7230	SK2T-8040
RS/6000 Redbooks Collection (PostScript)	SBOF-7205	SK2T-8041
RS/6000 Redbooks Collection (PDF Format)	SBOF-8700	SK2T-8043
Application Development Redbooks Collection	SBOF-7290	SK2T-8037

J.3 Other Publications

These publications are also relevant as further information sources:

- *MQSeries Application Programming Guide*, SC33-0807

- *MQSeries Application Programming Reference*, SC33-1673
- *MQSeries Command Reference*, SC33-1369
- *MQSeries System Administration*, SC33-1873

How to Get ITSO Redbooks

This section explains how both customers and IBM employees can find out about ITSO redbooks, CD-ROMs, workshops, and residencies. A form for ordering books and CD-ROMs is also provided.

This information was current at the time of publication, but is continually subject to change. The latest information may be found at `http://www.redbooks.ibm.com/`.

How IBM Employees Can Get ITSO Redbooks

Employees may request ITSO deliverables (redbooks, BookManager BOOKs, and CD-ROMs) and information about redbooks, workshops, and residencies in the following ways:

- **Redbooks Web Site on the World Wide Web**

  `http://w3.itso.ibm.com/`

- **PUBORDER** — to order hardcopies in the United States
- **Tools Disks**

  To get LIST3820s of redbooks, type one of the following commands:

  ```
  TOOLCAT REDPRINT
  TOOLS SENDTO EHONE4 TOOLS2 REDPRINT GET SG24xxxx PACKAGE
  TOOLS SENDTO CANVM2 TOOLS REDPRINT GET SG24xxxx PACKAGE (Canadian users only)
  ```

  To get BookManager BOOKs of redbooks, type the following command:

  ```
  TOOLCAT REDBOOKS
  ```

  To get lists of redbooks, type the following command:

  ```
  TOOLS SENDTO USDIST MKTTOOLS MKTTOOLS GET ITSOCAT TXT
  ```

  To register for information on workshops, residencies, and redbooks, type the following command:

  ```
  TOOLS SENDTO WTSCPOK TOOLS ZDISK GET ITSOREGI 1998
  ```

- **REDBOOKS Category on INEWS**
- **Online** — send orders to: USIB6FPL at IBMMAIL or DKIBMBSH at IBMMAIL

Redpieces

For information so current it is still in the process of being written, look at ″Redpieces″ on the Redbooks Web Site (`http://www.redbooks.ibm.com/redpieces.html`). Redpieces are redbooks in progress; not all redbooks become redpieces, and sometimes just a few chapters will be published this way. The intent is to get the information out much quicker than the formal publishing process allows.

How Customers Can Get ITSO Redbooks

Customers may request ITSO deliverables (redbooks, BookManager BOOKs, and CD-ROMs) and information about redbooks, workshops, and residencies in the following ways:

- **Online Orders** — send orders to:

	IBMMAIL	Internet
In United States:	usib6fpl at ibmmail	usib6fpl@ibmmail.com
In Canada:	caibmbkz at ibmmail	lmannix@vnet.ibm.com
Outside North America:	dkibmbsh at ibmmail	bookshop@dk.ibm.com

- **Telephone Orders**

United States (toll free)	1-800-879-2755
Canada (toll free)	1-800-IBM-4YOU
Outside North America	(long distance charges apply)
(+45) 4810-1320 - Danish	(+45) 4810-1020 - German
(+45) 4810-1420 - Dutch	(+45) 4810-1620 - Italian
(+45) 4810-1540 - English	(+45) 4810-1270 - Norwegian
(+45) 4810-1670 - Finnish	(+45) 4810-1120 - Spanish
(+45) 4810-1220 - French	(+45) 4810-1170 - Swedish

- **Mail Orders** — send orders to:

IBM Publications	IBM Publications	IBM Direct Services
Publications Customer Support	144-4th Avenue, S.W.	Sortemosevej 21
P.O. Box 29570	Calgary, Alberta T2P 3N5	DK-3450 Allerød
Raleigh, NC 27626-0570	Canada	Denmark
USA		

- **Fax** — send orders to:

United States (toll free)	1-800-445-9269
Canada	1-403-267-4455
Outside North America	(+45) 48 14 2207 (long distance charge)

- **1-800-IBM-4FAX (United States)** or **(+1)001-408-256-5422 (Outside USA)** — ask for:

Index # 4421 Abstracts of new redbooks
Index # 4422 IBM redbooks
Index # 4420 Redbooks for last six months

- **On the World Wide Web**

Redbooks Web Site	http://www.redbooks.ibm.com/
IBM Direct Publications Catalog	http://www.elink.ibmlink.ibm.com/pbl/pbl

Redpieces

For information so current it is still in the process of being written, look at ″Redpieces″ on the Redbooks Web Site (`http://www.redbooks.ibm.com/redpieces.html`). Redpieces are redbooks in progress; not all redbooks become redpieces, and sometimes just a few chapters will be published this way. The intent is to get the information out much quicker than the formal publishing process allows.

IBM Redbook Order Form

Please send me the following:

Title	Order Number	Quantity

First name Last name

Company

Address

City Postal code Country

Telephone number Telefax number VAT number

- Invoice to customer number
- Credit card number

Credit card expiration date Card issued to Signature

We accept American Express, Diners, Eurocard, Master Card, and Visa. Payment by credit card not available in all countries. Signature mandatory for credit card payment.

Index

D

E

F

G

N

O

P

ITSO Redbook Evaluation

MQSeries Version 5 Programming Examples
SG24-5214-00

Your feedback is very important to help us maintain the quality of ITSO redbooks. **Please complete this questionnaire and return it using one of the following methods:**

- Use the online evaluation form found at http://www.redbooks.ibm.com
- Fax this form to: USA International Access Code 914 432 8264
- Send your comments in an Internet note to redbook@us.ibm.com

Which of the following best describes you?
__Customer __Business Partner __Solution Developer __IBM employee
__None of the above

Please rate your overall satisfaction with this book using the scale:
(1 = very good, 2 = good, 3 = average, 4 = poor, 5 = very poor)

Overall Satisfaction ____________

Please answer the following questions:

Was this redbook published in time for your needs? Yes____ No____

If no, please explain:

____

____

____

____

What other redbooks would you like to see published?

____

____

____

Comments/Suggestions: (THANK YOU FOR YOUR FEEDBACK!)

____

____

____

____

____

MQSeries Version 5 Programming Examples **SG24-5214-00**

SG24-5214-00
Printed in the U.S.A.