Migrating to MQSeries® for AS/400®, V5.1

John Samuel

IBM United Kingdom Limited
Hursley Park
Winchester
Hampshire
SO21 2JN
United Kingdom

Note!

Before using this information and the product it supports, be sure to read the general information under Appendix C, “Notices” on page 43.

First edition (March 2000)

This edition applies to MQSeries for AS/400 V5.1

Contents

About this document

This document applies to IBM® MQSeries for AS/400, V5.1, and is intended for you to use during a migration planning exercise.

MQSeries for AS/400, V5.1 is far more than just an upgrade from previous releases. Unlike migration to earlier versions and releases of MQSeries for AS/400, migrating to this new version will require more emphasis on planning. This document is designed to point the systems administrator towards the major migration aspects of MQSeries for AS/400, V5.1 that will require careful planning.

While preparing this document, MQSeries for AS/400, V4.2.1 and MQSeries for AS/400, V5.1 systems were used.

Who this document is for

This document is for systems administrators and systems programmers who manage the configuration and administration tasks for MQSeries. It will also be useful to application programmers who have to maintain MQSeries interfaces within applications.

What you need to know to understand this document

To use this document, you should have a good understanding of the OS/400 operating system and an understanding of previous MQSeries releases for the AS/400.

Other reading

This document refers to:

- *MQSeries for AS/400, V5.1 Quick Beginnings*, GC34-5557-00, for a summary of the new functions introduced in MQSeries for AS/400, V5.1, and for installation and migration of MQSeries.
- *MQSeries Intercommunication*, SC33-1872-03, for post-installation configuration of a distributed queuing network.
- *MQSeries for AS/400, V5.1 System Administration*, SC34-5558-00.
- *MQSeries for AS/400, V5.1 Application Programming Reference (ILE RPG)*, SC34-5559-00.
- *ILE RPG for AS/400 Reference*, SC09-2508-02, for more information about threading.

The author

The author of this document is John Samuel, who specializes in Business Integration Solutions based on MQSeries for AS/400.

If you have any comments on this document, please contact me at this address:

IBM United Kingdom Limited
Hursley Park
Winchester
Hampshire
SO21 2JN
United Kingdom

The author would like to thank the following people for their help in producing this document:

Norman Adlam

Martin Hyne

Mark Phillips

James Taylor

Bruce Wassell

Chapter 1. Library structure

First of all, let's think about the library structure of previous releases of MQSeries:

QMQM
MQSeries for AS/400 Programs

QMQMDATA
MQSeries for AS/400 Objects

QMQMPROC
MQSeries for AS/400 Processes

QMQMADM
MQSeries AS/400 Administration Application

QMQMSAMP
MQSeries for the AS/400 Sample Routines

Only one queue manager could be created on the AS/400, and each of its objects (channels and queues) was stored in QMQMDATA along with the journal receivers. From a systems administration point of view, the most significant change for MQSeries for AS/400, V5.1 is that multiple queue managers are allowed. This means that the structure of MQSeries has radically changed. MQSeries for AS/400, V5.1 now appears like this:

QMQM
MQSeries for the AS/400 Programs

QMxxxxxxxx
Queue Manager. (xxxxxxxx is replaced by the first 8 digits of the queue manger's name.) If the first 8 characters of two queue managers are the same, the last 2 characters of the second queue manager are replaced with 00. This is incremented each time a queue manager is created with the same first 8 characters. Each queue manager created has its own library.

QMQMSAMP
MQSeries for AS/400 Sample Routines

MQSeries V5.1 now utilizes the Integrated File Structure (IFS) to a greater extent. These directories are created in the IFS:

```
/QIBM/ProdData/mqm/inc
/QIBM/ProdData/mqm/lib
/QIBM/ProdData/mqm/samp

/QIBM/UserProd/mqm/errors
/QIBM/UserProd/mqm/qmgrs
/QIBM/UserProd/mqm/trace
```

The directory qmgrs have directories for each of the queue managers created on the AS/400. All the MQSeries objects for that queue manager are stored in these subsequent directories. The name of the directory reflects the name of the queue manager. Within the queue manager directory several other directories are created, in particular the errors and queue directories.

Previous releases of MQSeries for AS/400 submitted jobs to QSYSWRK. Now with V5.1, all MQSeries jobs are submitted to its own subsystem called QMQM.

For each queue manager started, five jobs are submitted:

QMQM

AMQALMPX Checkpoint processor

AMQRRMFA Repository manager for clusters

AMQZLAA0 Queue manager agents

AMQZXMA0 The execution controller is the first job started by the queue manager

RUNMQCHI Channel Initiator

Consider the following situation. If you have multiple queue managers for your production environments and a test queue manager it might be unwise to have all the jobs running under one subsystem.

With MQSeries for AS/400, V5.1 it is now possible to separate different queue managers into different subsystems. Taking the above situation, it would probably be advisable to separate the test queue manager from the production queue managers. Once you've done that, you can specify different priorities for each queue manager. For instance, the test queue manager could be given less priority for its channel control jobs. For more information on job control see chapter 4 of the *MQSeries for AS/400, V5.1 System Administration*.

Chapter 2. Journals and backups

Because the structure of MQSeries has changed, many of your automated routines must be altered. In this section, we'll look at journal management and backing up MQSeries.

Managing journals

With previous releases of MQSeries, the local and remote journal receivers were held in the QMQMDATA library. This library is no longer required with V5.1. But each queue manager has its own library with its own journal and journal receiver.

AMQAJRN
MQM Local Journal

AMQAnnnnnn
MQM Local Journal Receiver

The remote journal is no longer required because channel synchronization no longer uses a database file. This can prevent confusion. To some degree, it also simplifies management because journal receiver changing is set by the system value. The biggest task in migrating to V5.1 in respect to existing automated routines will be handling the multiple queue managers. The creation of new queue managers now affects journal management in respect to the journal's physical location.

To help journal management, previous releases of MQSeries issued two messages to the QSYSOPR message queue:

```
AMQ7460        MQSeries start-up journal information
AMQ7462        MQSeries media recover journal information
```

These two messages are still issued to denote which journal receivers are required for successful MQSeries startup and media recovery, but they are now issued to QMQMMSG in the queue manager's library.

Backing up MQSeries for AS/400

Because MQSeries for AS/400, V5.1 makes use of IFS, you'll have to revise previous backup routines. Very little apart from the journal and journal receiver is held in the queue manager libraries, but this does not mean that they no longer have to be backed up. Virtually all the MQSeries objects are now held within the IFS directory structure and you must add this directory structure to your existing backup routines.

You are recommended to include everything below `/QIBM/UserData/mqm` in your backup routines at first, but when you have reached a stable production environment `/QIBM/UserData/mqm/qmgrs` would suffice, because there should be little or nothing in the trace and errors directories.

Of course, the QMQM and QMQMSAMP libraries still exist and must remain in your backup routines. To be included in this routine you should now add the directory `/QIBM/ProdData/mqm`.

For a more detailed guide to backup and recovery of MQSeries V5.1 for AS/400, refer to chapter 8 of the *MQSeries for AS/400, V5.1 Systems Administration.*

Journals and backups

Chapter 3. Configuration scripts (Changed commands)

When you create or customize MQSeries objects, it's useful to keep a record of all MQSeries definitions created. This record can then be used for:

- Recovery purposes
- Maintenance
- Rolling out MQSeries applications

You can obtain this record by:

- Creating CL programs to generate your MQSeries definitions for the AS/400

or

- Creating MQSC text files as SRC members to generate your MQSeries definitions using the cross-platform MQSeries command language.

Because you are migrating to V5.1, you should have one of the above. Most AS/400 implementations use CL programs to create definitions. These CL programs, if allowed to remain unchanged, could create unexpected definitions or might fail, resulting in only partially-created definition lists. The reason for this is that the CL commands used have changed. By "changed", I mean more function has been added to many of the CL commands, which now have a queue manager parameter. This queue manager parameter is set to *DFT for the default queue manager.

If you have followed the migration path described in the *MQSeries for AS/400, V5.1 Quick Beginnings* manual, your previous queue manager will be the default queue manager. If you have followed the migration path or have created a new default queue manager but have not changed your CL programs, the created definitions will be placed in the default queue manager. However, depending on each command processed, the default values might have changed because of the increased function in the new release. If you have not followed the migration path and have not created a default queue manager (that is, on the CRTMQM command default queue manager is *NO) and have not changed the CL programs, they will fail with the error `MQSeries queue manager not available`.

CHGMQM does not allow you to change the default status of a queue manager. However, creating another queue manager with DFTQMGR(*YES) transfers the default status to this new queue manager.

If you have created a queue manager but forgot to set it as the default queue manager, you have a couple of options to set it as the default:

- You could delete the queue manager and start again; this is not such a big deal if you have created the appropriate definition scripts in CL or MQSC.
- If you have started using the queue manager, deleting it will not be an option. To get round this, edit the `mqs.ini` file in `/QIBM/UserData/mqm` adding the following Stanza lines:

  ```
  DefaultQueueManager:
        Name=QM.MAIN
  ```

  So mqs.ini would look originally something like this:

```
QueueManager:
   Name=QM.MAIN
   Prefix=/QIBM/UserData/mqm
   Library=QMQM.MAIN
   Directory=QM!MAIN
QueueManager:
   Name=QM.TEST
   Prefix=/QIBM/UserData/mqm
   Library=QMQM.TEST
   Directory=QM!TEST
```

and changes to something like this

```
QueueManager:
   Name=QM.MAIN
   Prefix=/QIBM/UserData/mqm
   Library=QMQM.MAIN
   Directory=QM!MAIN
DefaultQueueManager:
      Name=QM.MAIN
QueueManager:
   Name=QM.TEST
   Prefix=/QIBM/UserData/mqm
   Library=QMQM.TEST
   Directory=QM!TEST
```

Queue manager `QM.MAIN` is now the default queue manager.

Chapter 4. Handling errors

In earlier releases of MQSeries, errors were reported using job logs and spool files. MQSeries for AS/400, V5.1 follows the other V5.1 releases in its error management. This new error management makes it far easier for an MQSeries systems administrator to locate error logs without having to have an understanding of the OS/400® operating system.

MQSeries uses a number of error logs to capture messages concerning the operation of MQSeries itself. The location of the error logs depends on whether the queue manager is known and available. The physical location of the error logs is in the IFS directory of the queue manager for which it is reporting.

For example, if the queue manager name is known and the queue manager is available, the error logs are in the error directory for that queue manager:

```
/QIBM/UserData/mqm/<Queue Manager>/errors
```

If the queue manager is not available, the error logs are in the `@SYSTEM` directory.

```
/QIBM/UserData/mqm/@SYSTEM/errors
```

When a queue manager is created, three error logs are created:

- AMQERR01.LOG
- AMQERR02.LOG
- AMQERR03.LOG

As error messages are generated, they are placed in AMQERR01. When AMQERR01 is larger than 256 KB, it is copied to AMQERR02. Before the copy, AMQERR02 is copied to AMQERR03. The previous contents, if any, of AMQERR03 are discarded.

So the latest error messages can always be found in AMQERR01 while the others are used to maintain a history. It might be worthwhile to check AMQERR01 for errors from time to time as a pre-emptive way of ensuring that your channels are running effectively and the general health of your queue manger is sound.

The error logs do not diminish the need to check the system operator messages. Operator messages identify normal errors, typically caused directly by users performing illegal operations or defining parameters that are not valid for commands. Some operator messages are written to AMQERR01 in the queue manager's directory, including starting and stopping the queue manager and the important journal check point messages.

The FFST™ (First Failure Support Technology™) is recorded in the IFS in the directory `/QIBM/UserData/mqm/errors` and within the problem database, which you can access using WRKPRB. The FFST files are named `AMQnnnn.mm.FDC`; `nnnn` is the ID of he process reporting the error and `mm` is the sequence number.

When FFST reports are generated, there is little the system administrator can do apart from raise that problem with IBM using the information contained within the FFST report.

Viewing these log and FFST files can be a challenge if you are not used to the IFS. The best 'native' OS/400 utility I've found is the EDTF command. Be careful when using the EDTF utility because it does not always show what you think it does. For instance, if you wish to view `AMQERR01.LOG` in `@SYSTEM` you must provide a fully qualified path, such as:

```
EDTF '/QIBM/Userdata/mqm/qmgrs/@SYSTEM/errors/AMQERR01.LOG'
```

This will then display the contents of AMQERR01.LOG. But if you navigate to the directory where the log files are held using the command WRKLNK, like this:

```
WRKLNK OBJ('/qibm/userdata/mqm/qmgrs/@SYSTEM/errors')
```

and then issue the command EDTF 'AMQERR01.LOG' without the qualifying path, EDTF displays what appears to be an empty AMQERR01.LOG file. In fact, EDTF has taken your default path, which is set in your user profile. To an EDTF beginner, this can be very confusing and might lead you to believe that the log file is empty and errors are not being reported to the correct place.

There is an alternative to EDTF that might be of more use if you need to email log or FFST files for analysis. By mapping a network drive to your AS/400, you are able to view the IFS as if it were a local drive to your PC. But the log and FFST files are on the AS/400 and are in EBCDIC. If you use a utility like Wordpad to view these files, it will display garbage because it expects ASCII files. To get round this problem, you can set client access to automatically display EBCDIC files as ASCII thus enabling you to read them using Wordpad. To enable client access to do this, click on “client access properties” and then select the “network drives” tab. At the bottom you will see the automatic EBCDIC/ASCII conversion section. If you wish to view the log files using this method, use the file extension `.log`. You will now be able to view your log files on the AS/400 as if they were local to your PC.

For more information on problem analysis, see chapter 9 of the *MQSeries for AS/400, V5.1 System Administration*.

Chapter 5. Application development (ILE RPG)

There are two approaches that can be taken when using the MQI (message queue interface) from within an RPG program:

1. Dynamic calls to the QMQM program interface
2. Static Bound Calls to the MQI procedures

MQSeries for AS/400, V5.1 increases the level of function available to the programmer, but only if the bound calls are used. This is the recommended approach, particularly when the program is making repeated calls to the MQI, because it requires less resource. RPG-OPM is still supported, so your compiled programs can still be maintained, but these programs will not have all the function of MQSeries for AS/400, V5.1 available to them. Also, they are slower.

Components to programming the MQI

To help your application developers migrate their existing programs, I'll briefly explain the RPG-OPM and RPG-ILE components to programming the MQI.

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

1. COPY files with names ending with the letter G are for use with programs that use static bound calls. The preferred method.
2. COPY files with names ending with the letter R are for use with programs that use dynamic calls.

These COPY files can be found in `QRPGLESRC` in library `QMQM`.

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

QMQM/AMQZSTUB
This service program provides compatibility bindings for applications written prior to V5.1 that do not require access to any of the new capabilities provided in V5.1. The signature of this service program is the same as the one contained in V4.2.1.

QMQM/LIBMQM
This service program contains the single-threaded bindings for V5.1

QMQ/LIBMQM_R
This service program contains the multi-threaded bindings for V5.1

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

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

I used the term “threading” above. This term might be new to many of you and therefore needs some consideration. In general, RPG programs should not use the multi-threaded service program (LIBMQM_R). The exception arises when RPG programs are created containing the `THREAD(*SERIALIZE)` keyword in the control specification. However, even though these programs are thread-safe, careful consideration must be given to overall application design, because `THREAD(*SERIALIZE)` forces serialization of RPG procedures at the module level, and this could have an adverse affect on overall performance.

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

For more information about threading see the *AD/400 ILE RPG/400 Reference* and the *ILE RPG for AS/400 Reference*.

Examples of bound and dynamic calls

Much of what we've talked about above will be new to many of you. Therefore, the following section contains a few examples of the differences between bound and dynamic calls.

With bound calls, to use these procedures you need to:

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

In this example, the MQOPEN call requires the inclusion of the following code:

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

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

```
C                   CALLP     MQOPEN(HCONN : MQOD : OPTS : HOBJ :
C                              OCODE : REASON)
```

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

Now compare this with the dynamic call.

To use the MQI through dynamic calls to QMQM, you require the following code (for MQOPEN again):.

```
C                    Z-ADD     MQOPEN        CID
C                    CALL      'QMQM'
C                    PARM                    CID             9 0
C                    PARM                    HCONN           9 0
C                    PARM                    MQOD
C                    PARM                    OPTS            9 0
C                    PARM                    HOBJ            9 0
C                    PARM                    OCODE           9 0
C                    PARM                    REASON          9 0
```

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

Sample programs

A good starting point for migrating your applications from RPG-OPM to RPG-ILE to take full advantage of its increased capability would be to take a look at the provided RPG samples. The samples are held in `QRPGLESRC` in library `QMQMSAMP`. Examples of coding using dynamic calls start AMQ2, whereas examples of coding using the bound calls start AMQ3. If you compare, say, AMQ2GET4 to AMQ3GET4 you will see that the coding differences are slight compared to the gain in function. I'll emphasize that the samples are samples; they are not intended to be used in production code. They are intended to give you an idea of how to use the MQI, but do not, for example, contain any error handling.

New MQI calls

I've talked above about some of the basic migration issues your application programmers will face. Now I would like to talk a little more about the new MQI calls available in MQSeries for AS/400, V5.1.

For more information on the above topics, refer the *MQSeries for AS/400 Application Programming Reference (ILE RPG)* manual.

MQCONN - Connect queue manager

Although MQCONN is not a new MQI call, its usage has changed somewhat and so it requires a brief explanation.

The MQCONN call connects an application program to a queue manager. It provides a queue manager connection handle, which is used by the application on subsequent MQI calls.

Programs using the dynamic MQI calls do not have to issue this call. These applications are connected automatically to the queue manager when they issue the first MQOPEN call. However, the MQCONN can still be used in these applications if it isn't just implied.

Programs using the bound MQI calls must use the MQCONN or MQCONNX call to connect to the queue manager, and the MQDISC call to disconnect from the queue manager. This is the recommended style of programming.

```
D*****************************************************************
D**  MQCONN Call -- Connect Queue Manager                        **
D*****************************************************************
D*
D*..1....:....2....:....3....:....4....:....5....:....6....:....7..
DMQCONN           PR                  EXTPROC('MQCONN')
D* Name of queue manager
D QMNAME                        48A
D* Connection handle
D HCONN                         10I 0
D* Completion code
D CMPCOD                        10I 0
D* Reason code qualifying CMPCOD
D REASON                        10I 0
D*

C                   CALLP     MQCONN(QMNAME : HCONN : CMPCOD : REASON)
```

I think it's useful at this stage to talk through a situation that could arise. If you have not followed the migration path described in the *MQSeries for AS/400, V5.1 Quick Beginnings* and have not created a default queue manager, your dynamic MQI call programs will try to connect to the default queue manager; but, because you don't have one, they will fail to connect. All subsequent QMQM calls will also fail within that program because they don't have a queue manager connection handle. You cannot code for this error in applications using previous releases of MQSeries, so it will be worth checking for.

Another useful application check is to make sure that applications required to access a non-default queue manager have been coded to do so. Otherwise, when you migrate to MQSeries for AS/400, V5.1 and you have various application queues split across queue managers, wrong application queues might start receiving messages meant for the other queue manager, if those queues share the same names.

MQCONNX - Connect queue manager (extended)

The MQCONNX call connects an application program to a queue manager. It provides a queue manager connection handle, which is used by the application on subsequent MQI calls. The MQCONNX call is similar to the MQCONN call, except that MQCONNX allows options to be specified to control the way that the call works. RPG-OPM applications cannot use this call.

```
D*****************************************************************
D**  MQCONNX Call -- Connect Queue Manager (Extended)           **
D*****************************************************************
D*
D*..1....:....2....:....3....:....4....:....5....:....6....:....7..
DMQCONNX          PR                  EXTPROC('MQCONNX')
D* Name of queue manager
D QMNAME                        48A
D* Options that control the action of MQCONNX
D CNO                           32A
D* Connection handle
D HCONN                         10I 0
D* Completion code
D CMPCOD
                      10I 0
                      D* Reason code qualifying CMPCOD
D REASON                        10I 0
D*
D*
C                   CALLP     MQCONNX(QMNAME : CNO : HCONN : CMPCOD :
                                    REASON)
```

MQCMIT - Commit changes

With this new release of MQSeries, there are two ways to define transaction boundaries that incorporate message puts and gets from queues under syncpoint control:

1. Use COMMIT and ROLLBACK as before. This requires that commitment scope be set at *JOB level and as a consequence database as well as message operations are included in the transaction. With this release, MQSeries now registers itself with OS/400 as a 2-phase commit resource. This ensures that the message and database operations are ALL committed or ALL rolled back as one unit.
2. Use the new (to MQSeries for AS/400) API calls MQCMIT and MQBACK. On AS/400, these affect only the message operations; they do not affect any other resources, such as the database. Generally speaking, they use less system resource than COMMIT and ROLLBACK.

The MQCMIT call indicates to the queue manager that the application has reached a syncpoint, and that all of the message gets and puts that have occurred since the last syncpoint are to be made permanent. Messages put as part of a unit of work are made available to other applications, messages retrieved as part of a unit of work are deleted. RPG-OPM applications cannot use this call. COMMIT through commitment control is available for database and queue synchronization.

```
D*****************************************************************
D**  MQCMIT Call -- Commit Changes                              **
D*****************************************************************
D*
D*..1....:....2....:....3....:....4....:....5....:....6....:....7..
DMQCMIT           PR                  EXTPROC('MQCMIT')
D* Connection handle
D HCONN                         10I 0 VALUE
D* Completion code
D CMPCOD                        10I 0
D* Reason code qualifying CMPCOD
D REASON                        10I 0
D*

C                   CALLP     MQCMIT(HCONN : CMPCOD : REASON)
```

MQBACK - Back out changes

Use this new MQSeries API call in conjunction with MQCMIT. For the reasons for using this call instead of ROLBK, refer to "MQCMIT - Commit changes" on page 13. The MQBACK call indicates to the queue manager that all of the message gets and puts that have occurred since the last syncpoint are to be backed out. Messages put as part of a unit of work are deleted and messages retrieved as part of a unit of work are reinstated on the queue. RPG-OPM applications cannot use this call; ROLBK through commitment control is available for database and queue synchronization.

```
D*****************************************************************
D**  MQBACK Call -- Back Out Changes                            **
D*****************************************************************
D*
D*..1....:....2....:....3....:....4....:....5....:....6....:....7..
DMQBACK           PR                  EXTPROC('MQBACK')
D* Connection handle
D HCONN                         10I 0 VALUE
D* Completion code
D CMPCOD                        10I 0
D* Reason code qualifying CMPCOD
D REASON                        10I 0

C                   CALLP     MQBACK(HCONN : CMPCOD : REASON)
```

MQBEGIN - Begin unit of work

The MQBEGIN API call is for cross-platform compliance for code porting purposes. In short, it does not provide any function to your RPG applications and if you do put it in your code it returns a warning message. That is all it does on MQSeries for AS/400, V5.1, but on other platforms it marks the beginning of a single unit of work to be used with MQCMIT and MQBACK as database and queue synchronization. RPG-OPM applications cannot use this call.

```
D*****************************************************************
D**  MQBEGIN Call -- Begin Unit of Work                          **
D*****************************************************************
D*
D*..1....:....2....:....3....:....4....:....5....:....6....:....7..
DMQBEGIN          PR                  EXTPROC('MQBEGIN')
D* Connection handle
D HCONN                         10I 0 VALUE
D* Options that control the action of MQBEGIN
D BO                            12A
D* Completion code
D CMPCOD                        10I 0
D* Reason code qualifying CMPCOD
D REASON                        10I 0

C                   CALLP     MQBEGIN(HCONN : BO : CMPCOD : REASON)
```

MQXCNVC - Convert characters

This is a good point to talk about user exits. If you have user exits running with previous releases of MQSeries they will need recompiling after you migrate to MQSeries for AS/400, V5.1. The recompile is to make your user exit thread-safe and, because user exits are called by MQSeries, they have to be teraspace enabled. This is why AMQVSTUB must be used instead of AMQZSTUB when recompiling compatibility-mode user exits. To recompile these programs, you can use the service program LIBMQM or AMQVSTUB. LIBMQM will, of course, give your user exit program full V5.1 capability. AMQVSTUB is for user exits written before V5.1 and do not require the increased capability of V5.1; this service program has the same signature as the V4.2.1 release.

```
D*****************************************************************
D**  MQXCNVC Call -- Convert Characters                          **
D*****************************************************************
D*
D*..1....:....2....:....3....:....4....:....5....:....6....:....7..
DMQXCNVC          PR                   EXTPROC('MQXCNVC')
D* Connection handle
D HCONN                         10I 0 VALUE
D* Options that control the action of MQXCNVC
D OPTS                          10I 0 VALUE
D* Coded character set identifier of string before conversion
D SRCCSI                        10I 0 VALUE
D* Length of string before conversion
D SRCLEN                        10I 0 VALUE
D* String to be converted
D SRCBUF                          *   VALUE
D* Coded character set identifier of string after conversion
D TGTCSI                        10I 0 VALUE
D* Length of output buffer
D TGTLEN                        10I 0 VALUE
D* String after conversion
D TGTBUF                          *   VALUE
D* Length of output string
D DATLEN                        10I 0
D* Completion code
D CMPCOD                        10I 0
D* Reason code qualifying CMPCOD
D REASON                        10I 0
```

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

```
C                   CALLP     MQOPEN(HCONN : OPTS : SRCCSI : SRCLEN :
C                                    SRCBUF : TGTCSI : TGTLEN :
C                                    TGTBUF : DATLEN : CMPCOD : REASON)
```

For more information on the above topics, refer to the *MQSeries for AS/400, V5.1 Application Programming Reference (ILE RPG)* manual.

For an example of how to convert a Dynamic Call Put program to a Bound Call Put program see Appendix A, “Dynamic call MQPUT program” on page 23.

For an example of how to convert a Dynamic Call Get program to a Bound Call Get program see Appendix B, “Dynamic call MQGET program” on page 33.

Chapter 6. Performance

For MQSeries for AS/400, V5.1, the fundamental changes to the product architecture and the addition of significant new function have inevitably changed the performance profile. When you consider the performance of individual MQSeries operations, such as connecting and disconnecting from queue managers, opening and closing queues, and putting and getting messages, differences are apparent.

Performance considerations

In MQSeries for AS/400, V5.1, some APIs show an improvement over V4.2.1, in particular MQCONN and MQDISC. This will be of benefit to customers with applications that make frequent and short transactions - from clients, for example. APIs that show reduced performance are MQCLOSE for an empty queue, MQOPEN and MQCLOSE for dynamic queues, and MQPUT and MQGET. For standard-bound, single-threaded MQPUT and MQGET of non-persistent messages of 1K in size, a measurement comparing timings of calls showed an overhead of approximately 100% at V5.1 compared with the (fast-bound) V4.2.1 figures. In the case of persistent messages, the overhead was much lower: approximately 20%. This difference can be accounted for, in its entirety, by the cost of the process switch, which is described below, for the standard-bound operations, because the corresponding measurements at V5.1 for fast-bound MQPUTs and MQGETs are almost identical to the V4.2.1 figures.

Any measurements such as these should be interpreted with the greatest caution. Clearly, the absolute cost of each operation in seconds and the mix in any particular customer application is particularly significant. Some customers, especially those with MQSeries transactions involving relatively few message puts and gets, might see performance improvements. Environments where the MQI content of applications is relatively high, for example batch applications involving substantial numbers of put and get operations, might see a performance degradation.

Fast-bound and standard-bound calls

Fundamental to the understanding of the performance differences between this release and the previous release is the comparison between ‘fast-bound’ or trusted execution of MQSeries functions and ‘standard-bound’ execution. In fast-bound mode, the MQSeries queue manager functions execute as part of the user process. This includes delicate operations such as the manipulation of shared memory segments where queue data may be stored. Clearly, this places a significant responsibility on the designer of the user application both from the point of view of integrity of vital MQSeries data and also in the need to maintain adequate security.

With standard-bound execution, the user and queue manager processes are separate, and a badly-behaved user application cannot directly compromise the queue manager. The user and queue manager processes use inter-process communications - similar to remote procedure calls - to exchange data and to coordinate their behavior. In this mode, each MQSeries operation, for example putting or getting a message, involves one or more process switches between the two sides. Compared with fast-bound mode, the process switch adds an overhead.

MQSeries for AS/400 prior to V5.1 always runs in fast-bound mode, with no process switch. Certain aspects of OS/400 architecture are exploited to make this possible, and safe and additional routines are incorporated to prevent the interruption of certain critical sections of code. It is not feasible to transfer these protective constructions to the new, IFS-based architecture of the V5.1 product. Consequently, MQSeries for AS/400, V5.1 runs, by default, in standard-bound mode. Any performance comparisons with the previous versions should take this into account. It is possible, through the use of the MQCONNX MQI function, to run in fast-bound mode with V5.1, but this is not generally recommended, for two reasons:

1. As stated above, the user application must be designed very carefully to ensure the integrity of MQSeries data, such as the content of queues. Fairly advanced programming techniques would be needed to implement such a design.
2. Fast-bound mode puts a severe security constraint on the application. Although OS/400 programming interfaces may be exploited to avoid this, their use is not common practice.

Checking a slow-running application

If you have an application that is running slowly, it might be in a loop, or waiting for a resource that is not available. But you could have a real performance problem. Your problem could be caused by a system operating near the limits of its capacity, or a system that suffers peak load times, such as early morning as users log on. However, if you find that performance degradation does not depend on system loading but continues when machine loading is low, you could have a poorly designed-application program.

Other performance factors

There are other factors that might be causing poor performance of MQSeries. Here are a few points worth considering.

Effects of message length
Although MQSeries for AS/400 allows messages to hold up to 100 MB of data, the amount of data in a message affects the performance of the application that deals with the message. By sending only the required data you will improve the performance of your applications, also lightening the load on your systems and network.

Effects of message persistence
Persistent messages are logged to an AS/400 journal. Journaling messages can reduce the performance of your application because it requires disk I/O. Your applications should specify persistent messages only if they need to survive queue manager restarts. If the message is not important, use nonpersistent messages and you should see a performance improvement.

Frequency of syncpoints
Programs that issue numerous MQPUT calls within a syncpoint, without committing them, can cause performance problems. Affected queues can fill up with messages that are currently unusable, while other tasks might be waiting to get these messages. This has implications in terms of storage required, and in terms of threads tied up with tasks that are attempting to get messages.

Use of the MQPUT1 call

Use MQPUT1 only when a single message has to be put to a queue; if more than one message has to be put to a queue, MQPUT should be used because MQPUT1 does a MQCONN. MQOPEN, MQPUT, MQCLOSE and MQDISC in one operation. If multiple messages are required, the overhead of multiple MQCONN, MQOPEN, MQCLOSE and MQDISC can really mount up.

Dynamic calls

Using dynamic calls within your RPG or COBOL applications can have a major effect on performance. By converting your applications to use bound calls, you might see performance improvements. Chapter 5, “Application development (ILE RPG)” on page 9 provides information about RPG application migration.

In addition to the existing dynamic call COBOL interface, service programs are now supplied to provide ILE bound procedure calls to the MQI. These service programs are AMQ0STUB for single-threaded applications and AMQ0STUB_R for multi-threaded applications. The programming interface is identical for both dynamic and bound COBOL calls; a compiler switch (LINKLIT(*PGM) or LINKLIT(*PRC)) allows you to specify which is to be used. Generally speaking, the bound procedure interface should provide superior performance, as well as providing the COBOL programmer with access to the MQI functions new to MQSeries for AS/400, V5.1: MQCONNX, MQCMIT an MQBACK.

Chapter 7. Security

Security for MQSeries for AS/400 changes significantly with V5.1. Security with V5.1 is implemented using the MQSeries Object Authority Manager (OAM).

The OAM manages users' authorizations to manipulate MQSeries objects, including queues and process definitions. It also provides a command interface through which you can grant or revoke access authority to an object for a specific group of users. The decision to allow access to a resource is made by the OAM, and the queue manager follows that decision. If the OAM cannot make a decision, the queue manager prevents access to that resource.

With the OAM, you are able to control access to MQSeries objects through the MQI. When an application program attempts to access an object, the OAM checks that the user profile making the request has the authorization for the operation requested. This enables different groups of users different kinds of access authority to the same object. For example, for a specific queue, one group may be allowed to perform both put and get operations, while another group may be allowed only to browse the queue.

During installation the following user profiles are created:

QMQM
Primarily used for internal product-only function

QMQMADM
Intended to be used as a group profile for administrators of MQSeries, giving access to CL commands and MQSeries resources

To grant and revoke authorities to MQSeries for AS/400 objects, use the MQSeries commands GRTMQMAUT and RVKMQMAUT. These commands have changed significantly for MQSeries for AS/400, V5.1, and you must consider them carefully before migration to V5.1. You will have to manually migrate your security setups. Because MQSeries does not use OS/400 security directly, some previously supported functions are no longer supported. The most used, probably, of these previously supported functions are authorization lists and reference objects. But security groups are supported and could be used to replace authorization lists. However, this will involve a certain amount of manual work.

For more information on security issues, see chapter 5 of the *MQSeries for AS/400, V5.1 System Administration* manual.

Appendix A. Dynamic call MQPUT program

Here's an example of a dynamic call MQPUT program, followed by the same program converted into a bound call MQPUT program.

```
H
 *****************************************************************
 *                                                               *
 *   Function:                                                   *
 *                                                               *
 *                                                               *
 *     AMQPUT is a sample RPGLE program to put messages on a     *
 *     message queue, and is an example of the use of MQPUT.     *
 *                                                               *
 *        -- sample input is taken from file defined in          *
 *           the source; the program parameter identifies        *
 *           the target queue                                    *
 *                                                               *
 *        -- writes each record in the file to the message       *
 *           queue, taking each record as the content            *
 *           of a datagram message                               *
 *                                                               *
 *        -- Creates a report for each MQI reason other than     *
 *           RCNONE; stops if there is a MQI completion code     *
 *           of CCFAIL                                           *
 *                                                               *
 *      Program logic:                                           *
 *           MQOPEN target queue for OUTPUT                      *
 *           for each record in file                             *
 *           .  MQPUT datagram message with text line as data    *
 *           MQCLOSE target queue                                *
 *                                                               *
 *                                                               *
 *****************************************************************
 *File Specification
FMsgInFile IF   E             Disk
FAMQPUTP1  O    E             PRINTER OflInd(*In99)
 /EJECT
 *
 *****************************************************************
 **  Input Specifications                                        *
 *****************************************************************
 *
 ** Declare Required MQI Structures (/COPY members in QMQMSAMP)
 ** NOTE This program uses supplied defaults when it can
 *
 * MQI Named Constants
D/COPY CMQR
 * MQI Object Descriptor
D MQOD            DS
D* MQOD Structure
D/COPY CMQODR
```

```
 * MQI Message Descriptor
D MQMD            DS
D* MQMD Structure
D/COPY CMQMDR
 * MQI Put Message Options
D MQPMO           DS
D* MQPMO Structure
D/COPY CMQPMOR

 *
 ** Declare variables used in MQI calls
 ** NOTE This program uses supplied defaults when it can
 *
D  CID            S              9  0
D  QMgrHandle     S              9  0
D  QHandle        S              9  0
D  Options        S              9  0
D  OpenCode       S              9  0
D  CompCode       S              9  0
D  Reason         S              9  0
D  MsgLength      S              9  0 Inz(%Size(MsgData))
D  QueueName      S             48

 *
 ** Error Messages
 *
D ErrMsg          S             80
D OpenFail        S             36    Inz('MQOPEN failed. Unable to -
D                                     open queue.')
D OpenRpt         S             33    Inz('MQOPEN did not complete -
D                                     normally.')
D PutErr          S             32    Inz('MQPUT did not complete -
D                                     normally.')
D CloseErr        S             34    Inz('MQCLOSE did not complete -
D                                     normally.')

 ** Define DS over fields in input file
 *
D MsgData       E DS                  ExtName(MsgInFile)
 /EJECT
 *
 ****************************************************************
 * Initialization and Setup                                     *
 ****************************************************************
 *
 * Program Parameter is name of queue to Put data to
C     *Entry        PList
C                   Parm                    QueueName
 *
 ** Use parameter as name of queue
 *
C                   Eval      ODON = QueueName
 *
C                   Eval      QMgrHandle = HCDEFH
 *
 *
 ************************************
 *  MQOPEN - Open Queue for Output  *
 ************************************
 *
 ** Open queue for output (and fail if quiescing)
 ** Resulting queue handle is QHandle
 *
C                   Eval      Options = OOFIQ + OOOUT
C                   Eval      CID = MQOPEN
C                   Call      'QMQM'
C                   Parm                    CID
C                   Parm                    QMgrHandle
C                   Parm                    MQOD
C                   Parm                    Options
C                   Parm                    QHandle
C                   Parm                    OpenCode
C                   Parm                    Reason
 ** If Reason code returned report it
 *
                   If        Reason <> RCNONE

 ** If Open code is fail...
                   If        OpenCode = CCFAIL
                   Eval      ErrMsg = OpenFail
 ** Else report reason...
                   Else
                   Eval      ErrMsg = OpenRpt
                   EndIf
                   Write     PutError
                   EndIf
 ******************************************
 *  Processing to put Messages on Queue   *
 ******************************************
 *
 ** Set initial loop condition based on result on MQOPEN
 ** (i.e. if MQOPEN failed then no messages will be put)
 *
C                   Eval      CompCode = OpenCode

 *  Start of loop to MQPUT messages
C                   Read      RMsgIn                                 LR
C                   Dow       CompCode <> CCFAIL
C                             And *InLR = *Off
 *
 ************************************
 *  MQPUT - Put messages on queue   *
 ************************************
 *
 *  Set Message Format to FMSTR (so can be converted)
C                   Eval      MDFMT = FMSTR

C                   Eval      CID = MQPUT
C                   Call      'QMQM'
C                   Parm                    CID
C                   Parm                    QMgrHandle
C                   Parm                    QHandle
C                   Parm                    MQMD
C                   Parm                    MQPMO
```

```
C                   Parm                    MsgLength
C                   Parm                    MsgData
C                   Parm                    CompCode
C                   Parm                    Reason

 ** If Reason code returned report it
 *
C                   If        Reason <> RCNONE
C                   Eval      ErrMsg = PutErr
C                   Write     PutError
C                   EndIf

C                   Read      RMsgIn                                  LR
C                   EndDo
 *
 ************************************
 *  MQCLOSE Close Queue             *
 ************************************
 *
 * If Queue was opened close it with no options
C                   If        OpenCode <> CCFAIL
C                   Eval      CID = MQCLOS
C                   Eval      Options = CONONE

C                   Call      'QMQM'
C                   Parm                    CID
C                   Parm                    QMgrHandle
C                   Parm                    QHandle
C                   Parm                    Options
C                   Parm                    CompCode
C                   Parm                    Reason
 ** If Reason code returned report it
 *
C                   If        Reason <> RCNONE
C                   Eval      ErrMsg = CloseErr
C                   Write     PutError
C                   EndIf

C                   EndIf
C                   Eval      *InLR = *On
```

Here's the bound call MQPUT program:

```
H
 *****************************************************************
 *                                                               *
 *   Function:                                                   *
 *                                                               *
 *                                                               *
 *     AMQPUT2 is a sample RPGLE program to put messages on a    *
 *     message queue, and is an example of the use of MQPUT.     *
 *                                                               *
 *        -- sample input is taken from file defined in          *
 *           the source; the program parameter identifies        *
 *           the target queue                                    *
 *                                                               *
 *        -- writes each record in the file to the message       *
 *           queue, taking each record as the content            *
 *           of a datagram message                               *
 *                                                               *
 *        -- Creates a report for each MQI reason other than     *
 *           RCNONE; stops if there is a MQI completion code     *
 *           of CCFAIL                                           *
 *                                                               *
 *      Program logic:                                           *
 *           MQCONN connect to Queue Manager                     *
 *           MQOPEN target queue for OUTPUT                      *
 *           for each record in file                             *
 *           .  MQPUT datagram message with text line as data    *
 *           MQCLOSE target queue                                *
 *           MQDISC disconnect from Queue Manager                *
 *                                                               *
 *                                                               *
 *****************************************************************
 *File Specification
FMsgInFile IF   E             Disk
FAMQPUTP2  O    E             PRINTER OflInd(*In99)
 /EJECT
 *
 *****************************************************************
 **  Input Specifications                                        *
 *****************************************************************
 *
 ** Declare Required MQI Structures (/COPY members in QMQMSAMP)
 ** NOTE This program uses supplied defaults when it can
 *
 * MQI Named Constants
D/COPY CMQG
 * MQI Object Descriptor
D MQOD            DS
D* MQOD Structure
D/COPY CMQODG
```

```
 * MQI Message Descriptor
D MQMD           DS
D* MQMD Structure
D/COPY CMQMDG
 * MQI Put Message Options
D MQPMO          DS
D/COPY CMQPMOG

 *
 ** Declare variables used in MQI calls
 ** NOTE This program uses supplied defaults when it can
 *
D  QMgrHandle     S            10I 0
D  QHandle        S            10I 0
D  Options        S            10I 0
D  OpenCode       S            10I 0
D  CompCode       S            10I 0
D  Reason         S            10I 0
D  MsgLength      S            10I 0 Inz(%Size(MsgData))
D  MsgPoint       S              *   Inz(%Addr(MsgData))
D  QManager       S            48
D  QueueName      S            48

 *
 ** Error Messages
 *
D ErrMsg         S            80
D OpenFail       S            36    Inz('MQOPEN failed. Unable to -
D                                   open queue.')
D OpenRpt        S            33    Inz('MQOPEN did not complete -
D                                   normally.')
D PutErr         S            32    Inz('MQPUT did not complete -
D                                   normally.')
D CloseErr       S            34    Inz('MQCLOSE did not complete -
D                                   normally.')

 ** Define DS over fields in input file
 *
D MsgData       E DS                ExtName(MsgInFile)
 /EJECT
 *
 ****************************************************************
 * Initialization and Setup                                     *
 ****************************************************************
 *
 * Program Parameter is name of queue to Put data to
C     *Entry        PList
C                   Parm                    QManager
C                   Parm                    QueueName
 *
 ** Use parameter as name of queue
 *
C                   Eval      ODON = QueueName
 *
 *
 *****************************************************************
 * MQCONN - Connect to the Queue Manager                         *
 *****************************************************************
 *
C                   CallP     MQCONN(QManager : QMgrHandle : OpenCode
C                             : Reason)

 ** If Reason code returned report it
 *
C                   If        Reason <> RCNONE

 ** If Open code is fail...
C                   If        OpenCode = CCFAIL
C                   Eval      ErrMsg = OpenFail
 ** Else report reason...
C                   Else
C                   Eval      ErrMsg = OpenRpt
C                   EndIf
C                   Write     PutError
C                   EndIf
 *
 ************************************
 *  MQOPEN - Open Queue for Output  *
 ************************************
 *
 ** Open queue for output (and fail if quiescing)
 ** Resulting queue handle is QHandle
 *
C                   Eval      Options = OOFIQ + OOOUT

C                   CallP     MQOPEN(QMgrHandle : MQOD : Options : QHandle
C                             : OpenCode : Reason)

 ** If Reason code returned report it
 *
C                   If        Reason <> RCNONE
 ** If Open code is fail...
C                   If        OpenCode = CCFAIL
C                   Eval      ErrMsg = OpenFail
 ** Else report reason...
C                   Else
C                   Eval      ErrMsg = OpenRpt
C                   EndIf
C                   Write     PutError
C                   EndIf

 *
 ******************************************
 *  Processing to put Messages on Queue   *
 ******************************************
 *
 ** Set initial loop condition based on result on MQOPEN
 ** (i.e. if MQOPEN failed then no messages will be put)
 *
C                   Eval      CompCode = OpenCode
 *  Start of loop to MQPUT messages
C                   Read      RMsgIn                                 LR
C                   Dow       CompCode <> CCFAIL
C                             And *InLR = *Off

 *
 ************************************
 *  MQPUT - Put messages on queue   *
 ************************************
 *
 *  Set Message Format to FMSTR (so can be converted)
C                   Eval      MDFMT = FMSTR

C                   CallP     MQPUT(QMgrHandle : QHandle : MQMD : MQPMO
C                             : MsgLength : MsgPoint : CompCode : Reason)

 ** If Reason code returned report it
 *
C                   If        Reason <> RCNONE
C                   Eval      ErrMsg = PutErr
C                   Write     PutError
C                   EndIf

C                   Read      RMsgIn                                 LR
C                   EndDo

 *
 ************************************
 *  MQCLOSE Close Queue             *
 ************************************
 *
 * If Queue was opened close it with no options
C                   If        OpenCode <> CCFAIL

C                   Eval      Options = CONONE

C                   CallP     MQCLOSE(QMgrHandle : QHandle : Options :
C                             CompCode : Reason)

 ** If Reason code returned report it
 *
C                   If        Reason <> RCNONE
C                   Eval      ErrMsg = CloseErr
C                   Write     PutError
C                   EndIf

C                   EndIf
```

```
 ************************************
 *  MQDISC - Disconnect from Queue  *
 ************************************
 *

C                   CallP     MQDISC(QMgrHandle :CompCode : Reason)

 ** If Reason code returned report it
 *
C                   If        Reason <> RCNONE
C                   Eval      ErrMsg = CloseErr
C                   Write     PutError
C                   EndIf

C                   Eval      *InLR = *On
```

Dynamic call MQPUT program

Appendix B. Dynamic call MQGET program

Here's an example of a dynamic call MQGET program, followed by the same program converted into a bound call MQGET program.

```
 H
 *****************************************************************
 *                                                               *
 *   Function:                                                   *
 *                                                               *
 *                                                               *
 *     AMQGET is a sample RPGLE program to get messages from a   *
 *     message queue, and is an example of the use of MQGET.     *
 *                                                               *
 *         -- sample gets messages from the queue named in       *
 *            the parameter                                      *
 *                                                               *
 *         -- writes  the contents of the message queue          *
 *            to a file.                                         *
 *                                                               *
 *            messages are removed from the queue                *
 *                                                               *
 *         -- Writes a message for each MQI reason other than    *
 *            RCNONE; stops if there is a MQI completion code    *
 *            of CCFAIL                                          *
 *                                                               *
 *     Program logic:                                            *
 *      Use the parameter to name the input queue                *
 *       MQOPEN queue for INPUT                                  *
 *        while no MQI failures,                                 *
 *        .  MQGET next message, remove from queue               *
 *        .  write message to file                               *
 *        .  (no message available is failure, and ends loop)    *
 *       MQCLOSE the subject queue                               *
 *                                                               *
 *                                                               *
 *****************************************************************
 *
 *File Specification
MsgOutFileO    E             Disk
AmqGetp1  O    E             PRINTER OflInd(*In99)
/EJECT
 *
 ****************************************************************
 **  Input Specifications                                       *
 ****************************************************************
 *
 ** Declare Required MQI Structures (/COPY members in QMQMSAMP)
 ** NOTE This program uses supplied defaults when it can
 *
 * MQI Named Constants
D/COPY CMQR
 * MQI Object Descriptor
D MQOD            DS
D* MQOD Structure
D/COPY CMQODR
 * MQI Message Descriptor
D MQMD            DS
D* MQMD Structure
D/COPY CMQMDR
 * MQI Get Message Options
D MQGMO           DS
D* MQGMO Structure
D/COPY CMQGMOR

 *
 ** Declare variables used in MQI calls
 *
D  CID            S              9  0
D  QMgrHandle     S              9  0
D  QHandle        S              9  0
D  Options        S              9  0
D  OpenCode       S              9  0
D  CompCode       S              9  0
D  Reason         S              9  0
D  MsgLength      S              9  0
D  BufferLen      S              9  0 Inz(%Size(MsgData))
D  QueueName      S             48
D  GetWait        S              9

 *
 ** Error Messages
 *
D ErrMsg          S             80
D OpenFail        S             36    Inz('MQOPEN failed. Unable to -
D                                     open queue.')
D OpenRpt         S             33    Inz('MQOPEN did not complete -
D                                     normally.')
D GetErr          S             32    Inz('MQGET did not complete -
D                                     normally.')
D CloseErr        S             34    Inz('MQCLOSE did not complete -
D                                     normally.')

 ** Define DS over fields in input file
 *
D MsgData       E DS                  ExtName(MsgOutFile)
 /EJECT
 *
 ****************************************************************
 * Initialization and Setup                                     *
 ****************************************************************
 *
 * Program Parameters are name of queue to Put data to
 * and GetWait interval (i.e. time to wait for new message on queue)
C     *Entry        PList
C                   Parm                    QueueName
C                   Parm                    GetWait
 *
 ** Use parameter as name of queue
 *
C                   Eval      ODON = QueueName
 *
C                   Eval      QMgrHandle = HCDEFH

 *
 ****************************************************************
 *               Main Processing                                *
 ****************************************************************
 *

 *
 ************************************
 *  MQOPEN - Open Queue for Input   *
 ************************************
 *
 ** Open queue for input (and fail if quiescing)
 ** Resulting queue handle is QHandle
 ** Exclusive or shared use of the queue is
 ** controlled by the queue definition in this sample
 *
C                   Eval      Options = OOFIQ + OOINPQ
C                   Eval      CID = MQOPEN
C                   Call      'QMQM'
C                   Parm                    CID
C                   Parm                    QMgrHandle
C                   Parm                    MQOD
C                   Parm                    Options
C                   Parm                    QHandle
C                   Parm                    OpenCode
C                   Parm                    Reason

 ** If Reason code returned report it
 *
C                   If        Reason <> RCNONE

 ** If Open code is fail...
C                   If        OpenCode = CCFAIL
C                   Eval      ErrMsg = OpenFail
```

```
 ** Else report reason...
C                   Else
C                   Eval      ErrMsg = OpenRpt
C                   EndIf
C                   Write     GetError
C                   EndIf

 *
 ******************************************
 *  Processing to get Messages from Queue *
 ******************************************
 *
 ** Set initial loop condition based on result on MQOPEN
 ** (i.e. if MQOPEN failed then no messages will be got)
 *
C                   Eval      CompCode = OpenCode

 *
 ** Start of loop to MQPUT messages
 *
 ** Read Messages off Queue until MQGET completion code = CCFAIL
 ** Set get message options to convert data (if necessary)
 ** and wait for messages.
 *
C                   Dow       CompCode <> CCFAIL
C                   Eval      GMOPT = GMWT + GMCONV
C                   Move      GetWait       GMWI
 *
 ** MsgId and CorrelId are selectors that must be cleared
 ** to get messages in sequence, as they are set each MQGET
 ** set them to none before each get
 *
C                   Eval      MDMID = MINONE
C                   Eval      MDCID = CINONE
C                   Clear                   MsgData
 *  call ...
C                   Eval      CID = MQGET
C                   Call      'QMQM'
C                   Parm                    CID
C                   Parm                    QMgrHandle
C                   Parm                    QHandle
C                   Parm                    MQMD
C                   Parm                    MQGMO
C                   Parm                    BufferLen
C                   Parm                    MsgData
C                   Parm                    MsgLength
C                   Parm                    CompCode
C                   Parm                    Reason

 ** If Reason code returned report it (RC2033 = No more msgs - ignore)
 *
C                   If        Reason <> RCNONE
C                             And Reason <> RC2033
C                   Eval      ErrMsg = GetErr
C                   Write     GetError
C                   EndIf
```

```
 ** Write record to file
 *
C                   If        CompCode <> CCFAIL
C                   Write     RMsgOut
C                   EndIf

C                   EndDo

 *
 ************************************
 *  MQCLOSE Close Queue             *
 ************************************
 *
 * If Queue was opened close it with no options
C                   If        OpenCode <> CCFAIL
C                   Eval      CID = MQCLOS
C                   Eval      Options = CONONE

C                   Call      'QMQM'
C                   Parm                    CID
C                   Parm                    QMgrHandle
C                   Parm                    QHandle
C                   Parm                    Options
C                   Parm                    CompCode
C                   Parm                    Reason

 ** If Reason code returned report it
 *
C                   If        Reason <> RCNONE
C                   Eval      ErrMsg = CloseErr
C                   Write     GetError
C                   EndIf

C                   EndIf

C                   Eval      *InLR = *On
```

Here's the bound call MQGET program:

```
 H
 ****************************************************************
 *                                                              *
 *   Function:                                                  *
 *                                                              *
 *                                                              *
 *     AMQGET2 is a sample RPGLE program to get messages from a  *
 *     message queue, and is an example of the use of MQGET.    *
 *                                                              *
 *         -- sample gets messages from the queue named in      *
 *            the parameter                                     *
 *                                                              *
 *         -- writes  the contents of the message queue         *
 *            to a file.                                        *
 *                                                              *
 *            messages are removed from the queue               *
 *                                                              *
 *         -- Writes a message for each MQI reason other than   *
 *            RCNONE; stops if there is a MQI completion code   *
 *            of CCFAIL                                         *
 *                                                              *
 *     Program logic:                                           *
 *      MQCONN connect to the Queue Manager                     *
 *      Use the parameter to name the input queue               *
 *       MQOPEN queue for INPUT                                 *
 *        while no MQI failures,                                *
 *        .  MQGET next message, remove from queue              *
 *        .  write message to file                              *
 *        .  (no message available is failure, and ends loop)   *
 *       MQCLOSE the subject queue                              *
 *      MQDISC disconnect from the Queue Manager                *
 *                                                              *
 *                                                              *
 ****************************************************************
 *
 *File Specification
FMsgOutFileO    E             Disk
FAmqGetp2  O    E             PRINTER OflInd(*In99)
 /EJECT
 *
 *****************************************************************
 **  Input Specifications                                        *
 *****************************************************************
 *
 ** Declare Required MQI Structures (/COPY members in QMQMSAMP)
 ** NOTE This program uses supplied defaults when it can
 *
 * MQI Named Constants
D/COPY CMQG
 * MQI Object Descriptor
D MQOD            DS
D* MQOD Structure
D/COPY CMQODG
 * MQI Message Descriptor
D MQMD            DS
D* MQMD Structure
D/COPY CMQMDG
 * MQI Get Message Options
D MQGMO           DS
D* MQGMO Structure
D/COPY CMQGMOG

 *
 ** Declare variables used in MQI bound calls
 *
D  QMgrHandle                   10I 0
D  QHandle                      10I 0
D  Options                      10I 0
D  OpenCode                     10I 0
D  CompCode                     10I 0
D  Reason                       10I 0
D  MsgLength                    10I 0
D  BufferLen                    10I 0 Inz(%Size(MsgData))
D  MsgPoint                       *   Inz(%Addr(MsgData))
D  QueueName      S             48
D  QManager       S             48
D  GetWait        S              9

 *
 ** Error Messages
 *
D ErrMsg          S             80
D OpenFail        S             36    Inz('MQOPEN failed. Unable to -
D                                     open queue.')
D OpenRpt         S             33    Inz('MQOPEN did not complete -
D                                     normally.')
D GetErr          S             32    Inz('MQGET did not complete -
D                                     normally.')
D CloseErr        S             34    Inz('MQCLOSE did not complete -
D                                     normally.')

 ** Define DS over fields in input file
 *
D MsgData       E DS                  ExtName(MsgOutFile)
 /EJECT
 *
 ****************************************************************
 * Initialization and Setup                                     *
 ****************************************************************
 *
 * Program Parameters are name of queue to Put data to
 * and GetWait interval (i.e. time to wait for new message on queue)
C     *Entry        PList
C                   Parm                    QManager
C                   Parm                    QueueName
C                   Parm                    GetWait
 *
 ** Use parameter as name of queue
 *
C                   Eval      ODON = QueueName

 *
 ****************************************************************
 * MQCONN - Connect to the Queue Manager                        *
 ****************************************************************
 *

C                   CallP     MQCONN(QManager : QMgrHandle : OpenCode :
C                             Reason)

 ** If Reason code returned report it
 *
C                   If        Reason <> RCNONE

 ** If Open code is fail...
C                   If        OpenCode = CCFAIL
C                   Eval      ErrMsg = OpenFail
 ** Else report reason...
C                   Else
C                   Eval      ErrMsg = OpenRpt
C                   EndIf
C                   Write     GetError
C                   EndIf
 *
 ************************************
 *  MQOPEN - Open Queue for Input   *
 ************************************
 *
 ** Open queue for input (and fail if quiescing)
 ** Resulting queue handle is QHandle
 ** Exclusive or shared use of the queue is
 ** controlled by the queue definition in this sample
 *
C                   Eval      Options = OOFIQ + OOINPQ

C                   CallP     MQOPEN(QMgrHandle : MQOD : Options : QHandle
C                             : OpenCode : Reason)
```

```
 ** If Reason code returned report it
 *
C                   If        Reason <> RCNONE

 ** If Open code is fail...
C                   If        OpenCode = CCFAIL
C                   Eval      ErrMsg = OpenFail
 ** Else report reason...
C                   Else
C                   Eval      ErrMsg = OpenRpt
C                   EndIf
C                   Write     GetError
C                   EndIf

 *
 ******************************************
 *  Processing to get Messages from Queue *
 ******************************************
 *
 ** Set initial loop condition based on result on MQOPEN
 ** (i.e. if MQOPEN failed then no messages will be got)
 *
C                   Eval      CompCode = OpenCode

 *
 ** Start of loop to MQPUT messages
 *
 ** Read Messages off Queue until MQGET completion code = CCFAIL
 ** Set get message options to convert data (if necessary)
 ** and wait for messages.
 *
C                   Dow       CompCode <> CCFAIL
C                   Eval      GMOPT = GMWT + GMCONV
C                   Move      GetWait       GMWI
 *
 ** MsgId and CorrelId are selectors that must be cleared
 ** to get messages in sequence, as they are set each MQGET
 ** set them to none before each get
 *
C                   Eval      MDMID = MINONE
C                   Eval      MDCID = CINONE
C                   Clear                   MsgData
 *  call ...
C                   CallP     MQGET(QMgrHandle : QHandle : MQMD : MQGMO
C                             : Bufferlen : MsgPoint : Msglength
C                             : CompCode : Reason)
```

```
 ** If Reason code returned report it (RC2033 = No more msgs - ignore)
 *
C                   If        Reason <> RCNONE
C                             And Reason <> RC2033
C                   Eval      ErrMsg = GetErr
C                   Write     GetError
C                   EndIf


 ** Write record to file
 *
C                   If        CompCode <> CCFAIL
C                   Write     RMsgOut
C                   EndIf

C                   EndDo

 *
 ************************************
 *  MQCLOSE Close Queue             *
 ************************************
 *
 * If Queue was opened close it with no options
C                   If        OpenCode <> CCFAIL

C                   Eval      Options = CONONE

C                   CallP     MQCLOSE(QMgrHandle : QHandle : Options
C                             : CompCode : Reason)

 ** If Reason code returned report it
 *
C                   If        Reason <> RCNONE
C                   Eval      ErrMsg = CloseErr
C                   Write     GetError
C                   EndIf

C                   EndIf

 ************************************
 *  MQDISC  Disconnect from Queue   *
 ************************************
 *

C                   CallP     MQDISC(QMgrHandle : CompCode : Reason)
 ** If Reason code returned report it
 *
C                   If        Reason <> RCNONE
C                   Eval      ErrMsg = CloseErr
C                   Write     GetError
C                   EndIf

C                   Eval      *InLR = *On
```

Appendix C. Notices

Trademarks