MQ update

17

November 2000

In this issue

MQ Update

Published by

Xephon
27-35 London Road
Newbury
Berkshire RG14 1JL
England
Telephone: +44 1635 550955
e-mail: harryl@xephon.com

North American office

Xephon/QNA
Post Office Box 350100
Westminster CO 80035-0100, USA
Telephone: +1 303 410 9344
Fax: +1 303 438 0290

Contributions

Articles published in *MQ Update* are paid for at the rate of £170 ($250) per 1000 words and £90 ($140) per 100 lines of code. For more information about contributing an article, please check Xephon's Web site, where you can download *Notes for Contributors*.

MQ Update on-line

Code from *MQ Update* is available from Xephon's Web site at www.xephon.com/mqupdate.html (you'll need the user-id shown on your address label to access it). If you've a problem with your user-id or password call Xephon's subscription department on +44 1635 33886.

Editor

Harry Lewis

Disclaimer

Subscriptions and back-issues

A year's subscription to *MQ Update*, comprising twelve monthly issues, costs £255.00 in the UK; $380.00 in the USA and Canada; £261.00 in Europe; £267.00 in Australasia and Japan; and £265.50 elsewhere. In all cases the price includes postage. Individual issues, starting with the July 1999 issue, are available separately to subscribers for £22.50 ($33.50) each including postage.

MQSeries recovery for Unix

INTRODUCTION

The MQSeries recovery process outlined in this article uses only supported MQSeries recovery procedures and utilities. It is based on MQSeries Version 5.1, and the latest IBM documentation should be reviewed for any improvements since this article was written.

In preparing these procedures, it is assumed that the required back-up and log files are available. Another consideration is that, while this article applies to most Unix installations, it was tested on Sun Solaris. This means that one or two details may differ from your installation.

RECOVERY CONCEPTS

A messaging system ensures that messages entered into the system are delivered to their destination. This means that the system must include a method of tracking messages in the system and recovering messages, if the system fails for any reason.

MQSeries ensures that messages are not lost by maintaining records (logs) of the activity of queue managers that handle the receipt, transmission, and delivery of messages. It uses these logs for two basic types of recovery:

1. *Restart recovery*, when you stop MQSeries in an orderly way. This is not covered in this article, though it's discussed in the IBM documentation.

2. *Crash recovery*, when MQSeries is stopped by an unexpected failure. This is the subject of this article.

In all cases, the recovery restores the queue manager to the state it was in when the queue manager stopped, with the exception that any in-flight transactions are rolled back, removing from the queues any messages that were not committed when the queue manager stopped. Recovery restores all persistent messages, though non-persistent messages are lost.

RISKS

You should be aware of the following risks associated with recovering an MQSeries system:

1 Recovery relies on the availability of all the necessary log files. The absence of a required file will prevent MQSeries recovery from taking place.

2 Only persistent messages are recovered.

3 Circular logging supports only *restart* recovery, which means that transactions that were in progress are rolled back. Linear logging is required if *full restart* and either *media* or *forward* recovery is required; in this case, existing transactions will be completed, if possible.

4 Logging requires that sufficient disk space is available.

  For *circular* logging, the disk space required is determined when the system is configured, and no extra space is allocated as the system runs, so space is not a problem with this type of log.

  For *linear* logging, a new log file is created only when it is needed, and it is then allocated sufficient space to accommodate its maximum size. Thus the system can run out of space only when a new log file is created, and regular archiving and management of the logs will ensure that this is not a problem.

MANAGING THE LOG FILES

Circular logging is generally used to minimize the support overhead. While linear logging maximizes the likelihood of recovering from errors, this is usually not necessary on non-transactional systems, where the majority of user-generated message traffic is designed to expire if not processed within a short interval. Procedures for both methods are described in this article.

The log files are stored in the directory:

```
/var/mqm/log/queue_mgr_name
```

All files and sub-directories in this directory must be backed up together – using files from different back-ups will cause the MQSeries queue manager to fail to start.

Over time, older linear log files are no longer required to restart the queue manager or perform the media recovery of any damaged objects. Periodically, the queue manager issues a pair of messages to indicate which of the log files are required:

- Message *AMQ7467* gives the name of the oldest log file needed to restart the queue manager. This log file and all newer log files must be available during queue manager restart.
- Message *AMQ7468* gives the name of the oldest log file needed to perform media recovery.

Log files older than the ones identified by these messages can either be archived and removed from the system or deleted, if not required for any business purpose.

If any log file that is needed cannot be found, operator message *AMQ6767* is issued. Make the log file, and all subsequent log files, available to the queue manager and retry the operation.

Note that, if you're performing media recovery using linear log files, all the required log files must be available in the log file directory at the same time. Make sure that you take regular media images of any objects you may wish to recover to avoid running out of disk space necessary to hold all the required log files.

CHECK-POINTING – ENSURING COMPLETE RECOVERY

Persistent updates to message queues happen in two stages. First, the records representing the update are written to the log, then the queue file is updated. The log files can, therefore, be more up-to-date than the queue files. To ensure that restart processing begins from a consistent point, MQSeries uses checkpoints. A checkpoint is a point in time when the record described in the log is the same as the record in the queue. The checkpoint itself consists of a series of log records that are needed to restart the queue manager, such as records relating to the state of all transactions (that is, units of work) active at the time of the checkpoint. This is why it is important to back up all the files in the log directory at the same time.

Checkpoints are generated automatically by MQSeries. They are taken when the queue manager starts, when it shuts down, when logging space is running low, and after every 1,000 logged operations.

As queues handle subsequent messages, the checkpoint record becomes inconsistent with the current state of the queues.

When MQSeries is restarted, it locates the latest checkpoint record in the log. This information is held in the checkpoint file that is updated at the end of every checkpoint. The checkpoint record represents the most recent point of consistency between the log and the data. The data from this checkpoint is used to rebuild the queues as they existed at the last checkpoint time. When the queues are recreated, the log is then played forward to bring the queues back to the state they were in before system failure or shutdown.

Checkpoints are used to make recovery more efficient and to control the re-use of primary and secondary log files.

BACKING UP AND RESTORING MQSERIES

When you run MQSeries, you must periodically take a back-up of your queue manager data to protect it against possible corruption resulting from hardware failure.

If the message data used by a system is short-lived, you may be tempted not to take back-ups. In this case, you can use the following shell scripts to recreate queue managers in the event of a failure:

```
createqm
```

this recreates the queue manager, while:

```
configqm
```

configures the queue manager with its default values. Note that, if you chose not to take back-ups, this is the only option that is available for restoring the system – in many cases, you will not be able to restart the queue manager, and it must be deleted and recreated in this way. This is not recommended.

The remainder of this article describes what to back up and how to use the back-ups to restore MQSeries to a working condition.

BACKING UP MQSERIES

To take a back up of a queue manager's data, you must:

1 Ensure that the queue manager is not running.

 If your queue manager is running, stop it with either the **endmqm -w** command or the **stopqm** shell script and wait while it quiesces. Note that, if you try to take a back-up of a running queue manager, the back-up may not be consistent as a result of updates that were in progress while the files were being copied.

2 Take copies of all the queue manager's data and log file directories, including the following subdirectories:

```
/var/mqm/config

/var/mqm/conv

/var/mqm/log/q_mgr_name

/var/mqm/qmgrs/q_mgr_name

/var/mqm/mqs.ini

$APPL_HOME/
```

 Make sure that you don't miss any files – especially the log control file and the configuration files. Some of the directories may be empty, but they'll all be required when you restore the back-up at a later date, so it is advisable to save them too.

3 Ensure that you preserve the ownership of the files (you can do this, for instance, with the **tar** command).

RESTORING MQSERIES

To restore a back-up of a queue manager's data, you must:

1 Ensure that the queue manager is not running.

2 Clear out the directories in which you are going to place the data you backed up.

3 Copy the backed up queue manager data and log files to the correct places.

4 Check the resulting directory structure to ensure that you have all of the ones required.

Also make sure that you have a log control file, as well as the log files themselves, and check that the MQSeries and queue manager configuration files are consistent so that MQSeries can look in the correct places for the restored data.

If the data was backed up and restored correctly, the queue manager will now start. Note that, even though the queue manager data and log files are held in different directories, you should back up and restore the directories at the same time. If the queue manager data and log files have different ages, the queue manager is not in a valid state and will probably not start. If it does start, your data will almost certainly be corrupt.

RECOVERING FROM PROBLEMS

MQSeries can recover from both communication failures and outages resulting from loss of power. In addition, it is sometimes possible to recover from other types of problem, such as the inadvertent deletion of a file.

There are several ways that your data can be damaged. MQSeries helps you recover from:

- A power loss in the system
- A communication failure
- A damaged data object (if linear logging is used)
- A damaged log volume (again, if linear logging is used).

This section looks at how the logs are used to recover from these types of problem. In some cases, such as recovering damaged objects, only linear logging provides sufficient information to support recovery.

Loss of power

If your system goes down after a power outage, MQSeries restores the queues to their committed state at the time of the failure when the queue manager is restarted. This ensures that no persistent messages are lost. Non-persistent messages, which don't survive when MQSeries stops, are discarded.

Whenever the Unix system stops unexpectedly, the TCP/IP links that support the channels between Unix and OS/390 need re-establishing (if applicable). Starting the queue manager automatically achieves this as far as the Unix system is concerned, but the OS/390 system is generally not aware of any problem, or will be permanently in a 'retrying' state. The channels on the OS/390 must be re-started to re-establish communication.

Communications failure

In the case of a communications failure, messages remain on queues until they are removed by a receiving application. If the message is being transmitted, it remains on the transmission queue until it is successfully transmitted. To recover from a communications failure, it is normally sufficient just to restart the channels using the link that failed.

Media recovery

Media recovery is the re-creation of objects from information recorded in a linear log – this is not applicable to systems running circular logs. For example, if an object file is inadvertently deleted or becomes unusable for some other reason, media recovery can be used to recreate it. The information in the log that's required for the media recovery of an object is called a *media image*. Media images can be recorded manually using the **rcdmqimg** command, and they're also recorded automatically in some circumstances.

A media image is a sequence of log records containing an image of an object from which the object itself can be recreated.

The first log record required to recreate an object is known as its *media recovery record* – this is the start of the most recent media image of the object. An object's media recovery record is part of the information recorded during a checkpoint.

When an object is recreated from its media image, it is also necessary to replay any log records describing updates performed on the object since the last image was taken.

Consider, for example, a local queue that has an image of the queue object taken before a persistent message was put on the queue. In order to recreate the latest image of the object, it is necessary to replay the log entries that record the putting of the message on the queue, as well as replaying the image itself.

When an object is created, the log records written contain enough information to recreate the object completely. These records make up the object's first media image. Media images of the following objects are subsequently recorded automatically by the queue manager at each shutdown:

- All process objects and queues that are not local
- Empty local queues.

As stated, media images can be recorded manually with the **rcdmqimg** command. This command causes a media image of the MQSeries object to be written. Once this is done, only the logs that hold the media image, and all logs created after this, are needed to recreate damaged objects. The benefit of doing this depends on such factors as the amount of free storage available and the speed at which log files are created.

Recovering media images

MQSeries automatically recovers some objects from their media image if it finds that they are corrupt or damaged. This applies particularly to damaged objects found during normal queue manager start-up. If any transaction was incomplete at the time of the last queue manager shutdown, any queue affected is also recovered automatically in order to complete the start-up operation.

You must recover other objects manually using the **rcrmqobj** command. This command replays the relevant records in the log to recreate the MQSeries object. The object is recreated from its latest image in the log, together with all applicable log events between the time the image was taken and the recreate command was issued. Note that, if an MQSeries object becomes damaged, the only valid actions that can then be performed on it are to delete it and to recreate it by this method. Also note that non-persistent messages cannot be recovered in this way.

It is important to remember that both the log file that contains the media recovery record and all subsequent log files must be available in the log file directory when attempting the media recovery of an object. If a required file cannot be found, operator message *AMQ6767* is issued and the media recovery operation fails. If you do not take regular media images of the objects that you may need to recreate, you can get into a situation where you have insufficient disk space to hold all the log files required to recreate an object.

RECOVERING DAMAGED OBJECTS DURING START UP

If the queue manager discovers a damaged object during start-up, the action it takes depends on the type of object and whether the queue manager is configured to support media recovery.

If the queue manager object is damaged, the queue manager cannot start unless it can recover the object. If the queue manager is configured with a linear log, and thus supports media recovery, MQSeries automatically tries to recreate the MQSeries object from its media images. If the logging method selected does not support media recovery, you can either restore a back-up of the queue manager or delete the queue manager.

If any transactions were active when the queue manager stopped, then local queues that contain persistent uncommitted messages that were put on or got from the queue as part of these transactions are also needed if the queue manager is to start successfully. If any of these local queues are found to be damaged, and the queue manager supports media recovery, it automatically attempts to recreate them from their media images.

If any queues cannot be recovered, MQSeries cannot start. Any damaged local queues that contain uncommitted messages that are discovered during the start-up processing of a queue manager that does not support media recovery are marked as 'damaged objects' and the uncommitted messages on them are ignored. This is because it is not possible to perform media recovery of damaged objects on such a queue manager and the only action left is to delete them. Message *AMQ7472* is issued to report the damage.

RECOVERING DAMAGED OBJECTS AT OTHER TIMES

Media recovery of objects is automatic only during start-up. At other times, when object damage is detected, operator message *AMQ7472* is issued and most operations using the object fail. If the queue manager object is damaged at any time after the queue manager has started, the queue manager performs a pre-emptive shutdown. If an object is damaged you may delete it or, if the queue manager has a linear log, attempt to recover it from its media image using the **rcrmqobj** command.

PROTECTING MQSERIES LOG FILES

It is important that you do not remove the log files manually when an MQSeries queue manager is running. If a user inadvertently (or maliciously) deletes log files that a queue manager needs to restart, MQSeries does not issue an error message and continues to process data, including persistent messages. The queue manager will shut down normally, but will fail to restart. Media recovery of messages then becomes impossible.

Any user with sufficient authority to remove logs that are being used by an active queue manager also has the authority necessary to delete other important queue manager resources, such as authorization files, queue files, the object catalogue, and MQSeries executables. They can, therefore, damage a running or dormant queue manager, either through inexperience or intent, in a way against which MQSeries cannot protect itself.

For this reason, you should exercise caution when conferring superuser or *mqm* authority.

RECOVERY SCENARIOS

This section looks at a number of possible problems and indicates how to recover from them.

Disk drive failure

You may suffer problems with a disk drive containing the queue manager data, the log, or both. Problems can include data loss or corruption. The above three cases differ only in the part of the data that survives, if any.

In all cases you must first check the directory structure for any damage and, if necessary, repair it. If you lose queue manager data, there is a danger that the queue manager directory structure is also damaged. If so, you must recreate the directory tree manually before you try to restart the queue manager.

Having checked for structural damage, there are a number of alternative things you can do, depending on the type of logging that you use. Bear in mind that, even if the MQSeries system initially uses circular logs, this may be changed in future. This article, therefore, covers both linear and circular logging recovery methods.

Where there is major damage to the directory structure or any damage to the log, remove all the old files back to the *QMgrName* level, including configuration files, logs, and the queue manager directory. Restore the last back-up and try to restart the queue manager.

For linear logging with media recovery, ensure the directory structure is intact and try to restart the queue manager. If the queue manager does not restart, restore a back-up. If the queue manager restarts, check whether any other objects have been damaged using MQSC commands, such as **DISPLAY QUEUE**. Recover those objects that are reported damaged using the **rcrmqobj** command, for example:

```
rcrmqobj -m QMgrName -t all *
```

where *QMgrName* is the queue manager being recovered. **-t all *** indicates that all objects of any type (except channels) are to be recovered. If only one or two objects are reported damaged, you may want to specify those objects by name and type.

For linear logging with media recovery and an undamaged log, you may be able to restore a back-up of the queue manager data leaving the existing log files and log control file unchanged. Starting the queue manager applies changes from the log to bring the queue manager back to its state when the failure occurred. This method relies on two sources of data. Firstly, the checkpoint file contains the information that determines how much of the data in the log must be applied to yield a consistent queue manager, so it’s vital that this file is restored as part of the queue manager data. Secondly, you must have the oldest log file that was required to start the queue manager at the time of the back-up – and all subsequent log files – available in the log file directory.

If this is not possible, you must restore a back-up of both the queue manager data and the log, both of which were taken at the same time.

Damaged queue manager object

If the queue manager object is reported as damaged during normal operation, the queue manager performs a pre-emptive shutdown. There are two ways of recovering from this, depending on the type of logging you use:

- For linear logging only, delete the file containing the damaged object manually and restart the queue manager. (You can use the **dspmqfls** command to determine the damaged object’s name in the filesystem.) Media recovery of the damaged object is automatic.
- For either circular or linear logging, restore the last back-up of the queue manager data and log and restart the queue manager.

Damaged single object

If a single object is reported as damaged during normal operation:

- Recreate the object from its media image (this works only with linear logging)

or:

- Restore the last back-up and restart the queue manager (this works with both circular and linear logging).

Automatic media recovery failure

If a local queue that's required to start a queue manager that uses a linear log is damaged, and automatic media recovery fails, restore the last back-up of the queue manager data and log and restart the queue manager.

Saida Davies
IBM (UK)

Microsoft MSMQ to MQSeries Bridge

INTRODUCTION

In this article, I first look at the structure of MSMQ, Microsoft's message queueing product, with which MQSeries users may be unfamiliar. Next, I discuss the integration of MQSeries with MSMQ, which is Windows NT/2000-specific and communicates with heterogeneous platforms via Microsoft's MSMQ-MQSeries Bridge.

MSMQ is an integral part of Windows 2000, so it's inevitable that many MQSeries users will be confronted with the task of integrating MSMQ islands with an MQSeries framework, even in organizations where an MQSeries infrastructure is already present in Windows NT. One of the primary drivers for the use of MSMQ in organizations where MQSeries is the message-oriented middleware standard is the current shortage of MQSeries skills – this may be compared with the 'application backlog' of the nineties, which led to the development of isolated departmental systems. While the integration of those systems proved to be one of the main challenges of the past decade, proper planning and an understanding of the interoperation of MSMQ and MQSeries could make the task of integrating the two systems painless.

This article is in two part – the second and concluding part appears in next month's issue of *MQ Update*.

MESSAGE QUEUEING WITH MSMQ

In common with MQSeries, message queueing in an MSMQ environment uses a store-and-forward technique to allow applications running at different times to communicate across distributed networks. This allows programs to share data across a network without having a synchronized connection between the sending and receiving components of the distributed application.

As you'd expect, the two main parts of MSMQ are *messages* and *queues*. Messages containing data are placed on message queues by a sending program. The message queue is just the location where messages are stored, and the messages are retrieved from queues by the receiving program. Queues can be managed using the GUI-based MSMQ Explorer, which provides a high-level view of the MSMQ hierarchy.

MSMQ messages consist of fields, known as message *properties*. Any number of properties is allowed, including none, yielding a message with a dynamic data structure. This is in contrast to MQSeries, where the fields comprise a fixed data structure.

The process of sending a message is as follows:

- The sending application specifies the message fields or properties, provides the field values, and then issues an API call to MSMQ to send the message.
- The MSMQ queue manager conveys the message to the destination queue. If the destination queue is not available, the message is stored and is subsequently automatically forwarded when a connection link is established.
- A receiving application issues an API call to receive the message from the queue.

Hence, both MSMQ and MQSeries provide the following services:

- Connectionless, asynchronous messaging.
- Guaranteed delivery.
- Message prioritization, allowing the order in which the receiving application gets messages to be specified.

- A user-defined message structure consisting of anything from no data, to a single byte, a text string, or a long and complex data structure, perhaps involving an encryption mechanism that restricts access to just the communicating applications.
- 'Send message' and 'receive message' operations that can participate in a transaction that is coordinated with other database operations, so that the entire transaction can be committed, if all operations succeed, or cancelled and rolled back, if any of the operations fail.
- An Application Programming Interface (API) that operates at the Application Layer of the ISO Reference Model for Open System Interconnection, thereby providing a simple interface between an application program and the network, freeing the application programmer from concerns about how networking or communications are implemented.

MSMQ OPERATION

As previously stated, MSMQ comprises several objects and components, the two most fundamental being the message and the queue. A message may contain text or binary data in any format, as long as the sender and receiver agree on the format.

Queues hold messages, and there are two types of queue:

- *Application* queues that are used by applications to send and receive data.
- *System* queues that are created when MSMQ is installed and include the dead letter queue, transactional dead letter queue, and journal queues.

The Message Queue Information Store (MQIS) is an SQL database that contains definitions of both queues and the MSMQ topology. Note that Active Directory Services replaces the MQIS in Windows 2000. Messages themselves exist either in memory or in the file system.

Using the graphical tool, MSMQ Explorer, you can modify the following MSMQ objects:

- *The MSMQ enterprise*

  The enterprise is the top layer of the MSMQ hierarchy and all other objects are affiliated with an enterprise. Right-clicking the *Enterprise* icon brings up the *Enterprise Properties* dialogue, which contains the tabs: *General*, *MQIS Defaults*, and *Security*. The *General* tab contains the enterprise name, the enterprise server, and the default lifetime of a message on the network. The *MQIS Defaults* tab contains the replication interval (the number of seconds between updates, which may be *external inter-site*, *internal*, or *intra-site*). The *Security* tab is for setting permissions, auditing, and ownership.

- *Servers*

  The PEC (Primary Enterprise Controller) is the basis of MSMQ's infrastructure or foundation. It holds the master copy of the MQIS and can serve as a PSC (Primary Site Controller), BSC (Backup Site Controller), or Routing Server. Right-clicking the PEC computer icon allows you to choose the *Computer Properties* dialogue with the tabs: *General*, *Network*, *Events*, *Status*, *IS Status*, *Dependent Clients*, *Tracking*, and *Security*.

  The *General* tab specifies:

  – The *Pathname*, which is the name of the machine that hosts the queue and is used as a prefix for the queue's pathname, which specifies the queue's physical location.

  – The *Original Site*, which is the site with which the machine was originally affiliated.

  – The *Site*, which is a parameter that allows the machine to be relocated to another site.

  – The *ID*, which is the GUID (Globally Unique Identifier) that's automatically assigned to every object.

  – The *Service*, which is a function of the computer.

  – The parameters *Limit Message Storage* (in KB) and *Limit Journal Storage* (also in KB), which limit the number of bytes that may be allocated to all messages in all queues (this doesn't limit the size of individual messages – a message is limited to 4 MB in Windows NT and 400 KB in Windows 98).

The *Network* tab contains buttons to *Add*, *Edit*, and *Remove* connected networks (*CN*s). A *CN* is a logical grouping of computers in which any two can have a direct session with each other. The *Events* tab provides a filtered listing of MSMQ-related events from the Event Viewer and includes a button to launch the Event Viewer itself. If you need to isolate a problem more rigorously, set up auditing using the *Security* tab. The *Status* tab provides MSMQ-related MQIS resource statistics from Performance Monitor, such as the number of sessions, IP sessions, IPX sessions, incoming messages per second, MSMQ incoming messages, outgoing messages per second, MSMQ outgoing messages, total messages in all queues, and total bytes in all queues. A button is also provided to launch Performance Monitor. The *IS Status* tab provides statistics on database replication. The *Dependent Clients* tab shows all clients connected to the server (clients that have synchronous MSMQ sessions with the server via RPC). With the *Tracking* tab, you can enable tracking on a queue and specify the type of message to track as it moves through the MSMQ network. This is useful for checking that messages are reaching their destination via the best route. The *Security* tab is equivalent to the one discussed earlier.

- *Connected Networks (CNs)*

  CNs map protocol types running on your network, presenting a logical group of computers in which any two systems in the group can establish a session with each other. For example, clients using IP would be in their own CN. You have the option to create an IP, IPX, or foreign CN. It is here that you can connect to an MQSeries system by creating a foreign CN. Right-clicking on a CN brings up a *Connected Network* properties page with two tabs: *General* and *Security*. The *General* tab displays the CN name, the protocol (such as IP), and the CN's ID (its GUID).

- *Queues*

  A queue is a logical representation of the physical data storage area for messages. Right-clicking on a queue brings up the *General*, *Advanced*, *Status*, and *Security* tabs. The *General* tab displays the GUID of the queue object, an application-defined *LABEL*, a user-defined *TypeID* (typically a GUID generated with the **guidgen.exe** utility), the creation date and time of the queue, and the last modification date and time of the queue. Note that the *LABEL* and *TypeID* are used to select a queue. The *Advanced* tab allows you to limit the amount of storage for all messages (the value is specified in KB), to specify whether the queue accepts only authenticated messages, to specify whether the queue can be used for transactional messages (this is specified when the queue is created), to specify whether the queue is 'journal enabled' (allowing you to store a copy of outgoing messages), to limit the amount of storage for journal messages (the value is, again, in KB), to set the queue's 'privacy level' (this determines whether the queue accepts only private encrypted messages, non-private messages, or both), and to specify the base priority of messages sent to a public queue on a scale from –32768 to +32767 (queue priority takes precedence over message priority).

- *Messages*

  As mentioned already, the size of messages is limited to 4 MB in NT and 400 KB in Windows 98. Right-clicking on a message brings up a *Message Properties* dialogue, which contains the *General*, *Queues*, *Body*, and *Sender* tabs. The *General* tab displays the application-defined message *LABEL*, which can be used to convey (for example) the message's purpose, the message's ID or GUID (which is created automatically when the message is placed on the queue), an application-defined priority level (on a scale from *0* to *7*, the default being *0*, which stands for 'no priority', so that messages are processed 'first in, first out' or FIFO), the message's *Tracked* flag, which indicates whether the message is being tracked, the MSMQ-set *Class* or message type, such as *Normal*, *Positive*/*Negative Acknowledgement* (arrived and read), or *Report*, the date and time the message was sent (for outgoing messages), and the date and time the message arrived (for incoming messages).

The *Queues* tab displays the names of the destination queue, the response queue (for reply messages), and the administration queue (for administrative messages, such as acknowledgements). The *Body* tab displays the content of the message and shows the message size in bytes. The *Sender* tab provides information on the sending computer and application, including the sending computer's GUID, the pathname or name of the computer, the User ID and Security Identifier (SID) of the sending application, and whether the message is either authenticated or encrypted (and the algorithm used, if applicable).

MSMQ capabilities include:

- Integration with Windows NT, which means integration with NT's security infrastructure, event/performance monitoring and logging, transaction support, and clustering support.
- Easy administration through the GUI-based MSMQ Explorer.
- Building applications using ActiveX components (primarily with Visual Basic) and Active Server Pages (ASP – both C++ and J++ are supported), and a C API.
- Message and queue prioritization. Messages can be assigned a priority from *0* to *7* (*7* being the highest) by the programmer; the receiving application then receives messages in order of priority. Queues can also be assigned a priority, and messages are then routed by queue priority first and message priority second.
- Dynamic message routing, which can be set by the administrator based on site-link costs. This allows you to implement load-balancing.
- The retention of a copy of the message until successful delivery by Microsoft Transaction Server (MTS). An application can manage a Unit Of Work (UOW) to prevent data loss in the event of a failure. It's the responsibility of either MTS or DTC (the Distributed Transaction Coordinator) to coordinate the transaction. MTS can ensure safe delivery by retrieving a message under

syncpoint control and writing the message to the destination also under syncpoint control. A syncpoint is the point where outstanding updates are made available and it works like a flag on a process thread. A UOW usually controls the transaction, and MTS manages all threads in the UOW. When all the UOW threads complete successfully, they're committed. If any thread fails, then all threads are stopped and the resources rolled back to the state before the UOW was initiated.

MSMQ ARCHITECTURE

There are four main servers that comprise the MSMQ infrastructure:

- *Primary Enterprise Controller (PEC)*

  The PEC is the first server to be installed when building an MSMQ infrastructure. It holds both the enterprise configuration data and the certification keys used in authenticating messages in the MQIS. The MQIS contains the master definitions for the enterprise, site, site links, connected networks, and users, plus the master copy of the PEC site's computer and queue definitions. In addition, the PEC contains a read-only copy of MQIS data from other sites obtained by replication. The PEC can perform the jobs of the other three servers, though this isn't advisable for performance reasons.

- *Primary Site Controller (PSC)*

  A site is typically a geographical location rather than just a logical division of LANs, and a PSC should be installed at each site. For example, you could place a PEC at the company headquarters and a PSC at each branch office. The PSC MQIS database holds master data about computers and queues at the site. In addition, the PSC contains a read-only copy of MQIS data from other sites obtained by replication. The PSC can also act as a Routing Server.

- *Back-up Site Controller (BSC)*

  A BSC should be installed at each site that contains a PEC or PSC to provide load balancing and redundancy in the event of a PEC or PSC failure. The BSC holds a read-only copy of the MQIS database replicated from the PSC or PEC, so no queues can be created using this server. The BSC can also act as a Routing Server. To install a BSC, a PEC or PSC must be available.

- *Routing Server (RS)*

  You install RSs to provide more than one path for messages to reach their destination queues. As a result, RSs can distribute messages to different servers and prevent 'session concentration', where too many hits are on one particular resource. RSs provide load balancing, intermediate store-and-forward message queuing, and dynamic routing. An RS doesn't hold the MQIS database and therefore requires a previously-installed PEC.

When changes occur to the infrastructure, such as changes to queues or computers, the MQIS is replicated. By default, intrasite replication occurs every two seconds, while intersite replication occurs every ten seconds.

Two types of client are supported: independent clients (ICs) and dependent clients (DCs). The programming interface is the same for both clients and servers, though clients require fewer resources. The client software can be installed on Windows 98/95 and NT systems (IC software requires NT4 or better).

- DCs require a highly available network where the session is maintained with the MSMQ server. If the link is broken, MSMQ services are not available to the application.
- ICs can store messages in private queues, which are normally used as reply queues and are not registered with the MQIS. These queues can be addressed only by their direct format name. A client application creates a private queue and sends the queue's format name in messages so that responding applications can reply to the named queue. Unlike DCs, an IC's application will continue to run when the link is broken with the network. Any messages sent that cannot be forwarded are just stored locally.

MSMQ-MQSERIES BRIDGE OPERATION

MSMQ-MQSeries Bridge was originally developed by Level 8

Systems and marketed as the FalconMQ Bridge. The product was then sold and licensed to Microsoft in early 1998 and is now bundled with Microsoft SNA Server, Service Pack 3.

The bridge acts as an intermediary or interface between MSMQ and MQSeries, allowing MSMQ applications to send messages to MQSeries queues and vice versa. The process is fairly transparent, operating entirely in the background. For example, an MSMQ application sends a message using a standard MSMQ *MQSendMessage()* API call (or an ActiveX control that implements this functionality) and the MQSeries application receives the message from the MQSeries queue using a standard MQSeries *MQGET()* API call. Since each application deals with its native environment, the complexity of the interaction is removed. Messages are routed to and from each messaging system even if both systems are not connected to the network at the same time.

The code example below illustrates the use of the MSMQ-MQSeries Bridge for sending messages between the two messaging systems (I'd like to thank Microsoft for their assistance in creating this code). It comprises six simple C programs that you can use to send test messages between the two systems:

- *EPRecv.c* is a client application for MSMQ that retrieves a message from a specified MSMQ queue and prints out its field values. The application is also able to recognize an MQSeries MQMD (MQ Message Descriptor) structure and print it out, if present.
- *EPSend.c* is another MSMQ client application that sends a message to MQSeries via a specified MSMQ queue. The application creates an MQSeries MQMD, setting the *MQMD.MsgType* to *MQMT_REPLY*.
- *MQSRRecv.c* is an MQSeries client application that gets messages from a queue using the MQI interface. You need to specify the MQSeries queue manager and the queue name.
- *MQSRSend.c* is another MQSeries client application that sends a batch of ten messages to an MQSeries queue for forwarding to MSMQ. You need to specify the QM to which to connect and the destination queue manager and queue.

- *MSMQRecv.c* and *MSMQSend.c* are simple MSMQ clients that respectively receive and send messages from MSMQ queues (*MSMQSend.c*, like *MQSRSend.c*, sends messages in batches of ten).

EPRECV.C

```
/*//////////////////////////////////////////////////////////////////////
//  PROGRAM: EPRecv.c                                                  //
//                                                                     //
//  PURPOSE:                                                           //
//    Receives messages from MSMQ, prints extension fields values.     //
//    Recognizes and prints MQMD structure, if it's stored in          //
//    PROPID_M_EXTENSION.                                              //
//                                                                     //
//  PARAMETERS:                                                        //
//    2. MSMQ queue name to receive from.                              //
//////////////////////////////////////////////////////////////////////*/

#include <windows.h>
#include <stdio.h>
#include <stdlib.h>
#include <cmqc.h>
#include <mq.h>
#include "MSMQExt.h"
#include "MQSRExt.h"

void Error(char *szFuncName, HRESULT hRes)
{
  printf("Error in %s: %X.\n", szFuncName, hRes);
  exit(1);
}

void DumpByteField(BYTE *pField, DWORD dwSize)
{
  DWORD i;
  for (i=0; i<dwSize; i++) {
    printf("%02X", pField[i]);
    if (i%8==7 && i<dwSize-1) printf(" ");
  }
}

void DumpGUID(GUID *pGUID)
{
  printf("{%08X-%04X-%04X-%04X-%02X%02X%02X%02X%02X%02X}",
    pGUID->Data1, pGUID->Data2, pGUID->Data3, *(WORD *)pGUID->Data4,
    pGUID->Data4[2], pGUID->Data4[3], pGUID->Data4[4],
    pGUID->Data4[5], pGUID->Data4[6], pGUID->Data4[7]);
}
```

```
void DumpCharField(char *pField, DWORD dwSize)
{
  char s[80];
  memcpy(s, pField, dwSize);  s[dwSize] = '\0';
  printf("%s", s);
}

void DumpExtensionProperty(DWORD dwSize, void *pExtBuffer)
{
  HANDLE hExt;
  HANDLE hCursor;
  HRESULT hRes;
  BYTE *pField;
  MQMD *pMQMD;
  DWORD dwFieldSize;
  GUID FieldGUID;
  if((hRes = EPOpen(&hExt, pExtBuffer, dwSize)) != MQ_OK)
    Error("EPOpen", hRes);

  hCursor = NULL;
  while((hRes = EPGet(hExt, EP_NEXT_FIELD, &hCursor, NULL, NULL,
    &dwFieldSize)) == MQ_OK) {
    pField = malloc(dwFieldSize);
    if((hRes = EPGet(hExt, EP_CURRENT_FIELD, &hCursor, &FieldGUID,
      pField, &dwFieldSize)) != MQ_OK)
      Error("EPGet", hRes);

    if(!memcmp(&FieldGUID, &sg_MSMQExtMQMD, sizeof(GUID))) {
      pMQMD = (MQMD *)pField;
      printf("    MQMD Extension Field found.  Dumping values:\n"
             "        MQMD.Report = %08X   MQMD.MsgType = %08X\n"
             "        MQMD.Feedback = %08X   MQMD.Priority = %d\n",
             pMQMD->Report, pMQMD->MsgType, pMQMD->Feedback,
             pMQMD->Priority);
      printf("        MQMD.ReplyToQMgr = '");
      DumpCharField(pMQMD->ReplyToQMgr, 48);       printf("'\n");
      printf("        MQMD.ReplyToQ    = '");
      DumpCharField(pMQMD->ReplyToQ,    48);       printf("'\n");
      printf("        MQMD.UserIdentifier = '");
      DumpCharField(pMQMD->UserIdentifier, 12);    printf("'\n");
      printf("        MQMD.ApplIdentityData = '");
      DumpCharField(pMQMD->ApplIdentityData, 12);  printf("'\n");
      printf("        MQMD.PutApplName = '");
      DumpCharField(pMQMD->PutApplName, 12);       printf("'\n");
      printf("        MQMD.PutDate = '");
      DumpCharField(pMQMD->PutDate, 8);            printf("'\n");
      printf("        MQMD.PutTime = '");
      DumpCharField(pMQMD->PutTime, 8);            printf("'\n");
      printf("        MQMD.MsgId = '");
      DumpByteField(pMQMD->MsgId, 24);             printf("'\n");
      printf("        MQMD.CorrelId = '");
      DumpByteField(pMQMD->CorrelId, 24);         printf("'\n");
    } else {
      printf("    Unknown Field: ");
      DumpGUID(&FieldGUID);                       printf("\n");
      printf("        Data = '");
      DumpByteField(pField, dwFieldSize);         printf("'\n");
    }
    free(pField);
  }

  if(hRes != MQ_ERROR_EXTENSION_FIELD_NOT_FOUND)
    Error("EPGet", hRes);

  EPClose(&hExt);
}

void main(unsigned int argc, char *argv[])
{
WCHAR wszFormatName[256], wszPathName[256];
HRESULT hRes;
DWORD dwSize;
QUEUEHANDLE hQueue;
MSGPROPID PropIds[6] = { PROPID_M_LABEL, PROPID_M_LABEL_LEN,
                         PROPID_M_BODY, PROPID_M_BODY_SIZE,
                         PROPID_M_EXTENSION, PROPID_M_EXTENSION_LEN };
MQPROPVARIANT PropVars[6];
HRESULT       PropStatus[6];
MQMSGPROPS    MsgProps = { 6, PropIds, PropVars, PropStatus };
WCHAR wszLabel[256];
char szBody[256], szExt[1024];

  /* check for queue name */
  if (argc != 2) {
    printf("Usage: EPRecv <queue name>\n");
    return;
  }

  wsprintf(wszPathName, TEXT("%S"), argv[1]);
  dwSize = 256;
  if((hRes = MQPathNameToFormatName(wszPathName, wszFormatName,
    &dwSize)) != MQ_OK)
    Error("MQPathNameToFormatName", hRes);

  if((hRes = MQOpenQueue(wszFormatName, MQ_RECEIVE_ACCESS,
    MQ_DENY_RECEIVE_SHARE, &hQueue)) != MQ_OK)
    Error("MQOpenQueue", hRes);

  printf("Queue opened.\n"
         "Waiting for messages to arrive.\n"
         "Use CTLR-C to stop.\n");
  while(1) {
    PropVars[0].vt = VT_LPWSTR;
    PropVars[0].pwszVal = wszLabel;
    PropVars[1].vt = VT_UI4;
    PropVars[1].ulVal = 256;
    PropVars[2].vt = VT_UI1|VT_VECTOR;
    PropVars[2].caub.cElems = 256;
    PropVars[2].caub.pElems = szBody;
    PropVars[3].vt = VT_UI4;
    PropVars[3].ulVal = 256;
    PropVars[4].vt = VT_UI1|VT_VECTOR;
    PropVars[4].caub.cElems = 1024;
    PropVars[4].caub.pElems = szExt;
    PropVars[5].vt = VT_UI4;
    PropVars[5].ulVal = 1024;

    if((hRes = MQReceiveMessage(hQueue, INFINITE, MQ_ACTION_RECEIVE,
      &MsgProps, NULL, NULL, NULL, NULL)) != MQ_OK)
      Error("MQReceiveMessage", hRes);

    printf("-------> Message arrived:\n");
    printf("Label = '%S'\n", PropVars[0].pwszVal);
    printf("Body  = '%s'\n", PropVars[2].caub.pElems);

    if(PropVars[5].ulVal) {
      printf("Extension property found.  Dumping values:\n");
      DumpExtensionProperty(PropVars[5].ulVal,
        PropVars[4].caub.pElems);
    }

    printf("\n");
  }

  MQCloseQueue(hQueue);
  exit(0);
}
```

EPSEND.C

```
/*///////////////////////////////////////////////////////////////////////
//  PROGRAM: EPSend.c                                                  //
//                                                                     //
//  PURPOSE:                                                           //
//    Sends messages to MSMQ queue, filling in the MQMD extension      //
//    field values. MQMD.MsgType is set to MQMT_REPLY.                 //
//                                                                     //
//  PARAMETERS:                                                        //
//    2. MSMQ queue name to send to.                                   //
///////////////////////////////////////////////////////////////////////*/
```

```
#include <windows.h>
#include <stdio.h>
#include <stdlib.h>
#include <cmqc.h>
#include <mq.h>
#include "MSMQExt.h"
#include "MQSRExt.h"

void Error(char *szFuncName, HRESULT hRes)
{
  printf("Error in %s: %X.\n", szFuncName, hRes);
  exit(1);
}

void main(unsigned int argc, char *argv[])
{
WCHAR wszFormatName[256], wszPathName[256];
HRESULT hRes;
DWORD dwSize;
QUEUEHANDLE hQueue;
MSGPROPID     PropIds[3] = { PROPID_M_LABEL, PROPID_M_BODY,
                             PROPID_M_EXTENSION };
MQPROPVARIANT PropVars[3];
MQMSGPROPS    MsgProps = { 3, PropIds, PropVars, NULL };
char szBody[80], szLabel[80], szBuffer[1024];
WCHAR wszLabel[80];
HANDLE hExt;
MQMD TheMQMD, MQMDDef = { MQMD_DEFAULT };

  /* check for queue name */
  if (argc != 2) {
    printf("Usage: EPSend <queue name>\n");
    return;
  }

  wsprintf(wszPathName, TEXT("%S"), argv[1]);
  dwSize = 256;
  if((hRes = MQPathNameToFormatName(wszPathName, wszFormatName,
    &dwSize)) != MQ_OK)
    Error("MQPathNameToFormatName", hRes);

  if((hRes = MQOpenQueue(wszFormatName, MQ_SEND_ACCESS, 0,
    &hQueue)) != MQ_OK)
    Error("MQOpenQueue", hRes);

  printf("Queue opened.\n");

  while(1) {
    printf("------> Sending message (Use CTRL-C to stop).");
    printf("Label: ");  gets(szLabel);
    printf("Body: ");   gets(szBody);
    wsprintf(wszLabel, TEXT("%S"), szLabel);
    PropVars[0].vt = VT_LPWSTR;  PropVars[0].pwszVal = wszLabel;
    PropVars[1].vt = VT_VECTOR|VT_UI1;
    PropVars[1].caub.cElems = strlen(szBody) + 1;
    PropVars[1].caub.pElems = szBody;

    memcpy(&TheMQMD, &MQMDDef, sizeof(MQMD));
    TheMQMD.MsgType = MQMT_REPLY;

    if((hRes = EPOpen(&hExt, NULL, 0)) != MQ_OK)
      Error("EPOpen", hRes);
    if((hRes = EPAdd(hExt, &sg_MSMQExtMQMD, &TheMQMD, sizeof(MQMD),
      NULL)) != MQ_OK)
      Error("EPAdd", hRes);
    dwSize = 1024;
    if((hRes = EPGetBuffer(hExt, szBuffer, &dwSize)) != MQ_OK)
      Error("EPGetBuffer", hRes);
    EPClose(&hExt);

    PropVars[2].vt = VT_VECTOR|VT_UI1;
    PropVars[2].caub.cElems = dwSize;
    PropVars[2].caub.pElems = szBuffer;

    if((hRes = MQSendMessage(hQueue, &MsgProps, NULL)) != MQ_OK)
      Error("MQSendMessage", hRes);
  }

  MQCloseQueue(hQueue);
  exit(0);
}
```

MQSRRECV.C

```
/*//////////////////////////////////////////////////////////////////////
//  PROGRAM: MQSRRecv.c                                               //
//                                                                    //
//  PURPOSE:                                                          //
//    Receive messages from IBM MQSeries queue using MQI channel.    //
//                                                                    //
//  PARAMETERS:                                                       //
//    1. MQSeries queue manager to connect.                           //
//    2. MQSeries queue name (on the same QM) to receive from.        //
//////////////////////////////////////////////////////////////////////*/

#include <windows.h>
#include <stdio.h>
#include <stdlib.h>
#include <malloc.h>
#include <cmqc.h>
```

```
void main(int argc, char *argv[]){
  PSTR      pszBuf = NULL;
  MQLONG    lCompCode,lReason,lBufferLength = 0,lDataLength;
  MQHCONN   hConn;
  MQHOBJ    hQueue;
  MQOD      MqOd = {MQOD_DEFAULT};
  MQMD      MqMdDef = {MQMD_DEFAULT},MqMd;
  MQGMO     MqGmo = {MQGMO_DEFAULT};
  MQCHAR48  szQMName;

  /* check for queue name */
  if (argc < 3) {
    printf("Usage: MQSRRecv <queue manager name> <queue name>\n");
    return;
  }

  /* connect to queue manager */
  strncpy(szQMName, argv[1], 48);
  MQCONN(szQMName,&hConn,&lCompCode,&lReason);
  if (lCompCode != MQCC_OK){
    printf("ERROR: MQCONN %s CC = %lu RS = %lu\n",argv[1],lCompCode,
      lReason);
    return;
  }

  /* Open the queue for recieve */
  strncpy(MqOd.ObjectName,argv[2],sizeof(MqOd.ObjectName));
  strncpy(MqOd.ObjectQMgrName,argv[1],sizeof(MqOd.ObjectQMgrName));
  MQOPEN(hConn,&MqOd,MQOO_INPUT_SHARED,&hQueue,&lCompCode,&lReason);
  if (lCompCode != MQCC_OK){
    printf("ERROR: MQOPEN QMNGR:%s QUEUE:%s CC = %lu RS = %lu\n",
            argv[1],argv[2],lCompCode,lReason);
    return;
  }

  printf("Use <CTL-C> to stop !\n");

  /* initialize put options */
  MqGmo.Options |= MQGMO_NO_SYNCPOINT | MQGMO_WAIT;
  MqGmo.WaitInterval = MQWI_UNLIMITED;

  /* receive all messages and print out */
  while (TRUE) {
    /* initialize message descriptor */
    memmove(&MqMd,&MqMdDef,sizeof(MqMd));
    /* receive message */
    MQGET(hConn,hQueue,&MqMd,&MqGmo,lBufferLength,pszBuf,&lDataLength,
      &lCompCode,&lReason);
    /* the buffer is too small for the message then realloc */
    if (lCompCode == MQCC_WARNING && lReason ==
      MQRC_TRUNCATED_MSG_FAILED){
```

```
      if (pszBuf) free(pszBuf);
      pszBuf = malloc(lDataLength);
      lBufferLength = lDataLength;
    } else if (lCompCode != MQCC_OK){
      printf("ERROR: MQGET QMNGR:%s QUEUE:%s CC = %lu RS = %lu\n",
              argv[1],argv[2],lCompCode,lReason);
      return;
    } else {
      printf("%s\n", pszBuf);
    }
  }
}
```

MQSRSEND.C

```
/*///////////////////////////////////////////////////////////////////////
//  PROGRAM: MQSRSend.c                                                //
//                                                                     //
//  PURPOSE:                                                           //
//    Send 10 messages from IBM MQSeries using MQI channel.            //
//                                                                     //
//  PARAMETERS:                                                        //
//    1. MQSeries QM to connect (server side of the MQI channel).     //
//    2. Destination QM for the message(s).                           //
//    3. Destination queue name for the message(s).                   //
///////////////////////////////////////////////////////////////////////*/

#include <windows.h>
#include <stdio.h>
#include <cmqc.h>

void main( int argc, char *argv[]){
  DWORD     i;
  char      szMessage[256];
  MQLONG    lCompCode,lReason;
  MQHCONN   hConn;
  MQHOBJ    hQueue;
  MQOD      MqOd = {MQOD_DEFAULT};
  MQMD      MqMdDef = {MQMD_DEFAULT},MqMd;
  MQPMO     MqPmo = {MQPMO_DEFAULT};

  /* check for queue name */
  if (argc < 4) {
    printf("Usage: MQSRSend <server Qmngr> <queue manager name>
      <queue name>\n");
    return;
  }

  /* connect to queue manager */
  MQCONN(argv[1],&hConn,&lCompCode,&lReason);
  if (lCompCode != MQCC_OK){
    printf("ERROR: MQCONN %s CC = %lu RS = %lu\n",argv[1],lCompCode,
      lReason);
    return;
  }

  /* Open the queue for send */
  strncpy(MqOd.ObjectName,argv[3],sizeof(MqOd.ObjectName));
  strncpy(MqOd.ObjectQMgrName,argv[2],sizeof(MqOd.ObjectQMgrName));
  MQOPEN(hConn,&MqOd,MQOO_OUTPUT,&hQueue,&lCompCode,&lReason);
  if (lCompCode != MQCC_OK){
    printf("ERROR: MQOPEN QMNGR:%s QUEUE:%s CC = %lu RS = %lu\n",
            argv[1],argv[2],lCompCode,lReason);
    MQDISC(&hConn, &lCompCode, &lReason);
    return;
  }

  /* initialize put options */
  MqPmo.Options |= MQPMO_NO_SYNCPOINT;

  /* Send 10 messages */
  for (i = 0 ; i < 10 ; i++) {
    /* build message body */
    sprintf(szMessage,"Test Message %lu - ",i);
    GetTimeFormat((LCID)NULL,0,NULL,"HH':'mm':'ss",
      szMessage+strlen(szMessage),256-strlen(szMessage));
    /* initialize message descriptor */
    memmove(&MqMd,&MqMdDef,sizeof(MqMd));

    /* send the message */
    MQPUT(hConn,hQueue,&MqMd,&MqPmo,sizeof(szMessage),szMessage,
      &lCompCode,&lReason);
    if (lCompCode != MQCC_OK){
      printf("ERROR: MQPUT QMNGR:%s QUEUE:%s CC = %lu RS = %lu\n",
              argv[1],argv[2],lCompCode,lReason);
      MQDISC(&hConn, &lCompCode, &lReason);
      return;
    }

    /* print the sent message */
    printf("%s\n",szMessage);
  }
  MQDISC(&hConn, &lCompCode, &lReason);
}
```

MSMQRECV.C

```
/*///////////////////////////////////////////////////////////////////////
//  PROGRAM: MSMQRecv.c                                                //
//                                                                     //
//  PURPOSE:                                                           //
//    Receive messages from Microsoft MSMQ queue.                    //
//                                                                   //
//  PARAMETERS:                                                      //
//    1. Source machine name.                                        //
//    2. Source queue name.                                          //
/////////////////////////////////////////////////////////////////////*/

#include <windows.h>
#include <stdio.h>
#include <wchar.h>
#include <mq.h>

void main( int argc, char *argv[]){
  HRESULT       hRes;
  QUEUEHANDLE   hQueue;
  WCHAR         wszArgv[256];
  WCHAR         wszFormatName[256];
  DWORD         dwFormatNameLen = 256;
  MSGPROPID     aPropID[2] = {PROPID_M_BODY,PROPID_M_BODY_SIZE};
  MQPROPVARIANT aPropVar[2] = {{VT_VECTOR|VT_UI1,0,0,0,0},
                               {VT_UI4,0,0,0,0}} ;
  MQMSGPROPS    MessageProps = {2,aPropID,aPropVar,NULL};

  /* check for queue name */
  if (argc < 2) {
    printf("Usage: MSMQRecv <machine name>\\<queue name>\n");
    return;
  }
  mbstowcs(wszArgv,argv[1], strlen(argv[1])+1);

  /* get the queue format name */
  hRes = MQPathNameToFormatName(wszArgv,wszFormatName,
    &dwFormatNameLen);
  if (hRes != MQ_OK) {
    wprintf(L"ERROR: MQPathNameToFormatName %s RC = %p\n",
      wszArgv,hRes);
    return;
  }

  /* Open the queue for recieve */
  hRes = MQOpenQueue(wszFormatName,MQ_RECEIVE_ACCESS,0,&hQueue);
  if (hRes != MQ_OK) {
    wprintf(L"ERROR: MQOpenQueue %s RC = %p\n",wszArgv,hRes);
    return;
  }

  printf("Use <CTL-C> to stop !\n");

  /* receive all messages and print out */
  while (TRUE) {
    hRes = MQReceiveMessage(hQueue,INFINITE,MQ_ACTION_RECEIVE,
      &MessageProps,NULL,NULL,NULL,NULL);
    /* the buffer is too small for the message then realloc */
    if (hRes == MQ_ERROR_BUFFER_OVERFLOW) {
      if (aPropVar[0].caub.pElems) free(aPropVar[0].caub.pElems);
      aPropVar[0].caub.pElems = malloc(aPropVar[1].ulVal);
      aPropVar[0].caub.cElems = aPropVar[1].ulVal;
    } else if (hRes != MQ_OK) {
        wprintf(L"MQReceiveMessage failed for queue %s RC = %p\n",
          wszArgv,hRes);
        return;
    } else {
      printf("%s\n",aPropVar[0].caub.pElems);
    }
  }
}
```

MSMQSEND.C

```
/*/////////////////////////////////////////////////////////////////////
//  PROGRAM: MSMQSend.c                                              //
//                                                                   //
//  PURPOSE:                                                         //
//    Send 10 messages from Microsoft MSMQ.                          //
//                                                                   //
//  PARAMETERS:                                                      //
//    1. Destination machine name.                                   //
//    2. Destination queue name.                                     //
/////////////////////////////////////////////////////////////////////*/

#include <windows.h>
#include <stdio.h>
#include <wchar.h>
#include <mq.h>

void main(int argc, char *argv[]){
  HRESULT       hRes;
  QUEUEHANDLE   hQueue;
  WCHAR         wszFormatName[256];
  DWORD         dwFormatNameLen = 256,i;
  char          szMessage[256];
  WCHAR         wszArgv[256];
  MSGPROPID     aPropID[1] = {PROPID_M_BODY};
  MQPROPVARIANT aPropVar[1] = {{VT_VECTOR|VT_UI1,0,0,0,0}} ;
  MQMSGPROPS    MessageProps = {1,aPropID,aPropVar,NULL};

  /* check for queue name */
  if (argc < 2) {
    wprintf(L"Usage: MSMQSend <machine name>\\<queue name>\n");
    return;
  }
  mbstowcs(wszArgv,argv[1], strlen(argv[1])+1);

  /* get the queue format name */
  hRes = MQPathNameToFormatName(wszArgv,wszFormatName,
    &dwFormatNameLen);
  if (hRes != MQ_OK) {
    wprintf(L"ERROR: MQPathNameToFormatName %s RC = %p\n",
      wszArgv,hRes);
    return;
  }

  /* Open the queue for send */
  hRes = MQOpenQueue(wszFormatName,MQ_SEND_ACCESS,0,&hQueue);
  if (hRes != MQ_OK) {
    wprintf(L"ERROR: MQOpenQueue %s RC = %p\n",wszArgv,hRes);
    return;
  }

  /* initialize message body property */
  aPropVar[0].caub.pElems = (PBYTE)szMessage;
  aPropVar[0].caub.cElems = sizeof(szMessage);

  /* Send 10 messages */
  for (i = 0 ; i < 10 ; i++) {
    /* build message body */
    sprintf(szMessage,"Test Message %lu - ",i);
    GetTimeFormat((LCID)NULL,0,NULL,"HH':'mm':'ss",
      szMessage+strlen(szMessage),256-strlen(szMessage));

    /* send the message */
    hRes = MQSendMessage(hQueue,&MessageProps,NULL);

    if (hRes != MQ_OK) {
        wprintf(L"MQSendMessage failed for queue %s RC = %p\n",
          wszArgv,hRes);
        return;
    }

    /* print the sent message */
    printf("%s\n",szMessage);
  }
}
```

The MSMQ-MQSeries Bridge has two main components:

- The bridge itself, which converts and transmits messages between the two environments.
- The bridge Explorer, which lets you configure, monitor, and control message traffic through the bridge.

The bridge converts fields or properties between the two environments, mapping the fields of an incoming message to ones that are appropriate to the destination message's queuing system. The actual operation proceeds as follows: in sending a message from MQSeries to MSMQ, the bridge maps the fields of the MQSeries message to its MSMQ counterpart and, if non-existent additional fields are required at the destination, then the bridge provides the fields. For example, if an MSMQ message includes a specific value of the *PROPID_M_TIME_TO_BE_RECEIVED* property, then the MSMQ-MQSeries Bridge maps this property to the MQSeries *MQMD.Expiry* property, multiplying the value by 10 to change the units from seconds to tenths of a second.

The power of using the Bridge comes from the fact that there is no restriction on message content. The sending and receiving applications can impose their own internal structure or encryption schemes, which are left unaltered by the bridge.

The MSMQ-MQSeries Bridge is an MSMQ Connector server application. This requires further explanation: when MSMQ communicates with other PECs or Exchange Server, it does so through the MSMQ Connector. The Connector allows MSMQ applications to communicate with computers that use other messaging systems. The MSMQ-MQSeries Bridge uses the MSMQ Connector – the bridge is installed on a Windows NT system that serves as a connection point between the networks. An MSMQ Server must be installed on the same computer that hosts the MSMQ-MQSeries Bridge, and the computer must be connected by either a TCP/IP or SNA link to an MQSeries Queue Manager.

You can connect any number of MSMQ or MQSeries systems or networks using the MSMQ-MQSeries Bridge, and you can connect more than one MSMQ-MQSeries Bridge to one or more MQSeries Queue Managers.

Running the MSMQ-MQSeries Bridge in conjunction with Level 8 Systems' FalconMQ Client and FalconMQ Server extends MSMQ to non-Windows systems such as CICS or Unix. You can then transmit messages between, for example, MSMQ applications running under Unix and MQSeries applications running under MVS (it's unlikely, though, that MQSeries users would connect Unix and MVS applications this way – it’s more likely that they would run MQSeries on both Unix and MVS).

As indicated earlier, the MSMQ-MQSeries Bridge uses the MSMQ Connector, which allows MSMQ applications to communicate with ‘foreign computers’ that use other messaging systems. The MSMQ Connector communicates with foreign computers using foreign connected networks and foreign queues. The steps to create a foreign connected network (which is just another CN) are:

- At the top of the MSQM Explorer hierarchy, right-click on the *Enterprise* icon to display a context menu for the PEC.
- Select *New* followed by *CN* to create a new connected network.
- You are prompted for a name and a protocol. The protocol options include establishing a connection between MSMQ nodes running different protocols (IP, IPX) and between nodes running a foreign queuing system (including MQSeries).

Once the foreign CN (connected network) is created, you then create a ‘foreign computer’. An MQSeries Queue Manager would be a foreign computer or, essentially, a node hosting queues or functioning like a computer from MSMQ’s point of view. The steps to create a foreign computer are:

1 Within MSMQ Explorer, right-click the yellow ‘building site’ icon.

2 Select *New*, *Foreign computer*.

3 You are prompted for the foreign computer’s name, which is the name of your MQSeries Queue Manager.

4 Select the *Foreign connected network* to which the foreign computer belongs from the list in the left window pane.

5 Click *Add* to transfer the selection to the right window pane.

Next you need to activate the MSMQ Open Connector security permissions. The steps to do this are:

1 Within MSMQ Explorer, right-click the newly created foreign connected network.

2 Select *Properties.*

3 Click the *Security* tab, and then click the *Permissions* button.

4 Double-click the group that will communicate with the foreign connected network.

5 A panel appears with the following fields:

```
Connected Network: Sample_CN
Name: Sample_Group
O Full Control (All)
O Other
[ ] Open Connector
[ ] Set Properties (Sp)
[ ] Delete CN (D)
[ ] Get Permissions (Pg)
[ ] Set Permissions (Ps)
[ ] Take Ownership (O)
```

6 Check the *Open Connector* option.

The MSMQ configuration outlined is used to communicate with another MSMQ Enterprise or any foreign system.

This article concludes in next month's issue of *MQ Update*.

Stephen Ibaraki
Technical Manager and Head of Research
Capilano College (Canada)

Customizing CSQ4MQxx for an application

This article provides an example *CSQ4MQxx* file, which shows how the file *CSQ4DISX* that's supplied in the *MQM.SCSQPROC* dataset can be customized to meet the requirements of a specific application. This sample contains a set of object definitions that are for tailoring a unique *MQxx* queue manager. The attributes of the queues may be changed here, and these objects must be reviewed if this member is being copied to create another queue manager.

AN EXAMPLE OF CUSTOMIZED CSQ4DISX (CSQ4MQXX)

```
******************************************************************
*
* @START_COPYRIGHT@
*   Statement:     Licensed Materials - Property of IBM
*
*                  5695-137
*                  (C) Copyright IBM Corporation. 1993, 1996
*
*   Status:        Version 1 Release 1
* @END_COPYRIGHT@
*
******************************************************************
*
*               IBM MQSeries for MVS/ESA
* CSQ4MQxx
*
******************************************************************
*
* This sample dataset contains a set of object definitions
* that are specific to tailoring an MQxx queue manager.
*
* These objects must be reviewed if this member is being copied to
* create another queue manager.
*
******************************************************************
* NON-SYSTEM DEFINITIONS
******************************************************************
*
* The queue manager should have a default transmission queue and
* a dead-letter local queue.
* The names of the queues are specified by the ALTER QMGR command.
*
* This is a SAMPLE definition of what is needed.
*
* The attributes of the queues may be changed. However, if the
* attributes are changed such that when the queue manager tries
* to PUT a message on the dead-letter queue the PUT fails,
* the message will be discarded.
*
******
DEFINE QLOCAL( MQxx.DEFXMIT.QUEUE' ) +
         REPLACE +
* Common queue attributes
         DESCR( MQxx default transmission queue' ) +
         PUT( ENABLED ) +
         DEFPRTY( 0 ) +
         DEFPSIST( YES ) +
* Local queue attributes
         GET( ENABLED ) +
         SHARE +
         DEFSOPT( EXCL ) +
         MSGDLVSQ( FIFO ) +
         RETINTVL( 999999999 ) +
         MAXDEPTH( 999999999 ) +
         MAXMSGL( 4194304 ) +
         NOHARDENBO +
         BOTHRESH( 0 ) +
         BOQNAME( ' ' ) +
         STGCLASS( 'REMOTE' ) +
         USAGE( XMITQ ) +

* Event control attributes
         QDPMAXEV( ENABLED ) +
         QDPHIEV( DISABLED ) +
         QDEPTHHI( 80 ) +
         QDPLOEV( DISABLED ) +
         QDEPTHLO( 40 ) +
         QSVCIEV( NONE ) +
         QSVCINT( 999999999 ) +

* Trigger attributes
         NOTRIGGER +
         TRIGTYPE( NONE ) +
         TRIGMPRI( 0 ) +
         TRIGDPTH( 1 ) +
         TRIGDATA( ' ' ) +
         PROCESS( ' ' ) +
         INITQ( ' ' )
*
******
DEFINE QLOCAL( MQxx.DEAD.QUEUE' ) +
         REPLACE +
* Common queue attributes
         DESCR( MQxx dead-letter queue' ) +
         PUT( ENABLED ) +
         DEFPRTY( 0 ) +
         DEFPSIST( YES ) +

* Local queue attributes
         GET( ENABLED ) +
         SHARE +
         DEFSOPT( SHARED ) +
         MSGDLVSQ( FIFO ) +
         RETINTVL( 999999999 ) +
         MAXDEPTH( 999999999 ) +
         MAXMSGL( 4194304 ) +
         NOHARDENBO +
         BOTHRESH( 0 ) +
         BOQNAME( ' ' ) +
         STGCLASS( 'SYSTEM' ) +
         USAGE( NORMAL ) +

* Event control attributes
         QDPMAXEV( ENABLED ) +
         QDPHIEV( DISABLED ) +
         QDEPTHHI( 80 ) +
         QDPLOEV( DISABLED ) +
         QDEPTHLO( 40 ) +
         QSVCIEV( NONE ) +
         QSVCINT( 999999999 ) +

* Trigger attributes
         NOTRIGGER +
         TRIGTYPE( NONE ) +
         TRIGMPRI( 0 ) +
         TRIGDPTH( 1 ) +
         TRIGDATA( ' ' ) +
         PROCESS( ' ' ) +
         INITQ( ' ' )
*
******
* Alter the queue manager attributes for this instance.
*
ALTER QMGR +
         DESCR( MQxx, IBM MQSeries for MVS/ESA - V1.1.4' ) +
         TRIGINT( 999999999 ) +
         MAXHANDS( 256 ) +
         INHIBTEV( DISABLED ) +
         LOCALEV( DISABLED ) +
         REMOTEEV( DISABLED ) +
         STRSTPEV( ENABLED ) +
         PERFMEV( DISABLED ) +
         DEFXMITQ( MQxx.DEFXMIT.QUEUE' ) +
         DEADQ( MQxx.DEAD.QUEUE' )
*
*******************************************************************
* END OF CSQ4MQP1
*******************************************************************
```

Saida Davies
IBM (UK)

Migrating MQSI 1 rules and formats to Version 2

If your rules and formats are in MQSeries Integrator 1.1 format, then you don't need to convert them to use them with MQSeries Integrator Version 2, as they're already compatible with the latter. The only requirement is that you must ensure that MQSI V2 components are able to access the rules and formats in your database (this task is described in the IBM documentation – see the *MQSeries Integrator Installation Guide* for your OS). If you're migrating from MQSI 1.0, however, your formats and rules need to be converted. To do this, use the following procedure, which makes use of the support for MQSI V1.1 formats in MQSI V2:

1 Export your rules and formats to flat files using the MQSI V1 NNRie and NNFie Import/Export utilities. Make sure you complete this step *before* you uninstall MQSI V1. Also note that you cannot use the NEONFormatter and NEONRules user interface tools to complete this step.

2 After installing MQSeries Integrator, create the database containing table spaces that are required by MQSI 1.1. The exact details of this step depend on your choice of database (you must choose the one that you use for MQSI V1 data).

3 Import your MQSeries Integrator Version 1.0 rules and formats from the flat file into your MQSeries Integrator Version 1.1 database, using NNFie and NNRie. These tasks are described in IBM's *MQSeries Integrator Version 1.1 Installation and Configuration Guide*.

4 Use the Consistency Checker utility to check the consistency of your MQSI 1.1 rules and formats in the database.

You can now use your V1.1 rules and formats with MQSI V2.

Motul Patel
MQSeries Administrator (UK)

MQ news

Progress Software has extended the connectivity of its SonicMQ Internet messaging server with a family of bridges to other Java Message Service (JMS) and messaging systems, including MQSeries, and Internet B2B services. The new components comprise SonicMQ Bridge for IBM MQSeries, Bridge for JMS, Mail Connector, which extends and integrates JMS messaging with ubiquitous e-mail protocols, and the Bridge Programming Framework.

Beta versions of the bridges and connector, documentation, sample code, and the SonicMQ Programming Framework can be downloaded currently for free. The price and shipping date have yet to be announced.

For further information contact:
Progress Software, 14 Oak Park, Bedford MA 01730, USA
Tel: +1 781 280 4000
Fax: +1 781 280 4095
Web: http://www.progress.com

Progress Software, The Square, Basingview, Basingstoke, Hants RG21 2EQ, UK
Tel: +44 1256 816668
Fax: +44 1256 463226

* * *

J D Edwards has announced OneWorld Adapter for IBM MQSeries, a storefront package that integrates OneWorld B2B software and WebSphere Commerce Suite using Commerce Integrator Server and MQSeries. The integration allows sales orders to be placed and tracked by customers via the Internet.

It's out now, and details on pricing are available on request from the vendor.

For further information contact:
J.D. Edwards, One Technology Way, Denver, CO 80237, USA
Tel: +1 303 334 4000
Fax: +1 303 334 4141
Web: http://www.jde.com

J D Edwards, Colorado House, 300 Thames Valley Park Drive, Reading, Berkshire RG6 1RD, UK
Tel: +44 118 909 1700
Fax: +44 118 909 1699

* * *

IBM has announced VisualAge Smalltalk 5.5, the latest version of its application development tool for building object-oriented programs. The new version supports the latest levels of MQSeries, DB2 Universal Database, Oracle, Domino, and Java. The product includes VisualAge Smalltalk library support for Red Hat Linux Version 6, Windows 2000, and OS/2 Warp Server Version 4.5.

Out now, Enterprise Version 5.5 costs US$5,000.

For further information, contact your local IBM representative.