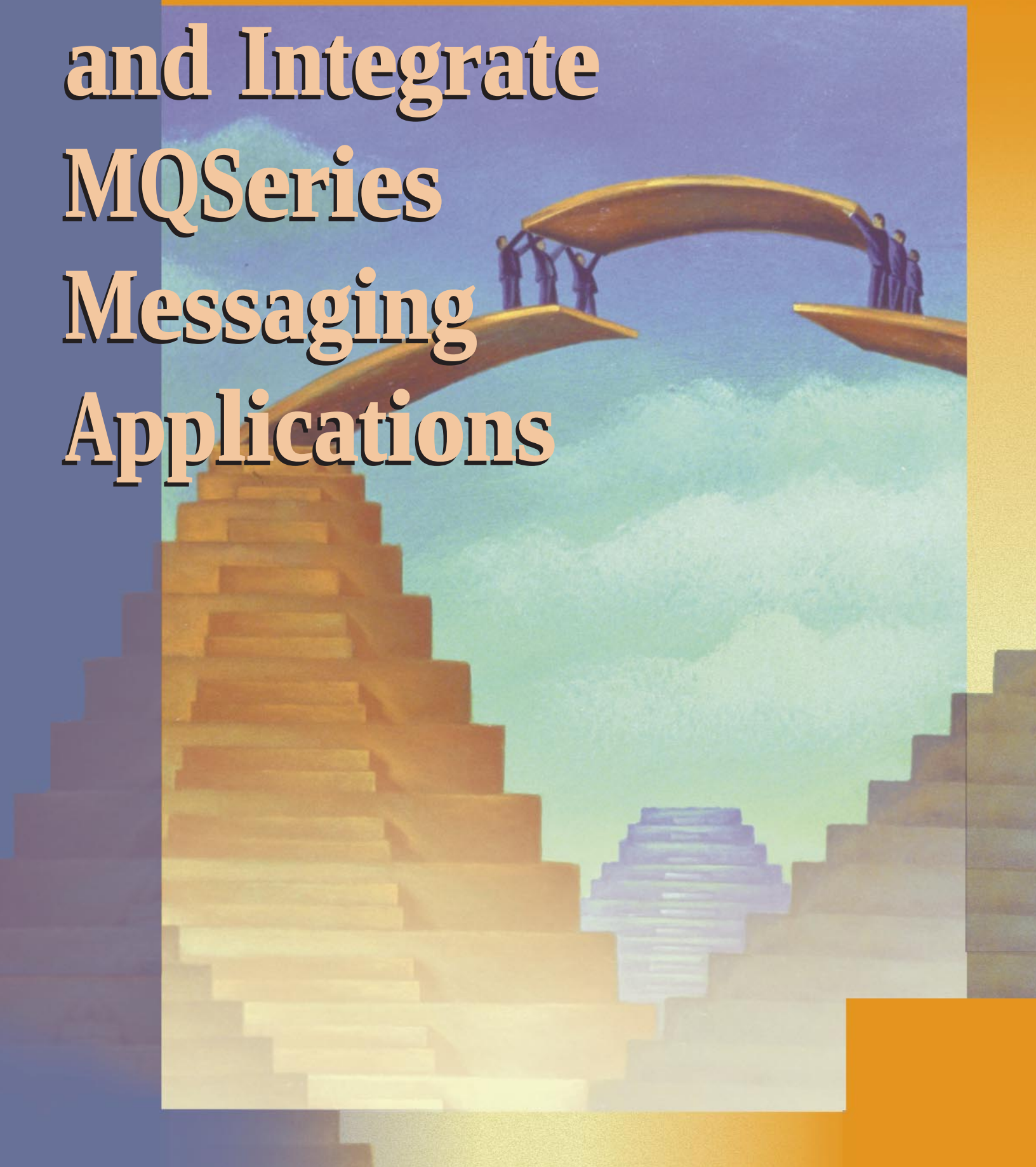
How to Develop and Integrate MQSeries Messaging Applications



The Development

This publication, consisting of two key chapters excerpted from the book entitled, ***Messaging & Queuing Using the MQI****, describes how to develop messaging applications. Messaging and queuing constitutes an efficient, effective, and stable software "machine" that can be used to power new application design styles, integrate existing applications, and integrate new and existing (legacy) applications. The challenge is to recognize the uses to which this software machine can be put.*

As with lots of machinery, possible uses are sometimes not so obvious and a few hints and suggestions are helpful. It is usually helpful to know a little bit about how any particular machine works and then a lot about how to operate it. The same is true with the messaging and queuing machine, and this part of the book tells you a little about the software machine (the innards of the Message Queue Manager) and a lot about how to operate it (access it with the Message Queue Interface).

The MQI provides the programmer with access to messaging and queuing services. The programmer uses the MQI while the messaging and queuing service supports (provides) the MQI (interprets and executes the CALL requests), as shown earlier in Fig. 3.9.

This part of the book describes:

- *The Message Queue Manager (MQM)*
- *The Message Queue Interface (MQI)*

This publication is an excerpt (Chapters 11 and 12) of ***Messaging & Queuing Using the MQI*** by Burnie Blakeley, Harry Harris and Rhys Lewis. Published by McGraw-Hill, Inc.

Chapter 11

MQSeries Messaging and Queing

In this chapter, we will look at the characteristics of the MQSeries Message Queue Managers. We will see how this family of products provides messaging and queuing services. We will also review most of the resources used within MQSeries and cover the terminology used. We begin by looking at the queue manager itself.

11.1 The Local Queue Manager

In MQSeries, the queue manager is the basic software system. Every queue in an MQSeries system belongs to one of the queue managers. The queue manager provides access to the queues on behalf of application programs and other MQSeries components, such as the administration tools.

Applications request services from the queue manager. They do this using an application programming interface known as the *Message Queue Interface*, or MQI. These services include the basic operations of putting a message onto a queue and getting a message from a queue. The queue manager software provides for reliable storage of queued messages. It also manages security authorization and concurrent access to the data, as well as providing specific queuing function, such as triggering, which we will look at later in this chapter.

Any application wishing to use MQSeries facilities must first connect to a queue manager. The queue manager to which an application is directly connected is known as the *local queue manager*. The local queue manager is the one which provides direct support for MQI calls made by the application. Applications can manipulate the resources owned by the local queue manager, as well as being able to put messages destined for other queue managers.

A queue manager is the basic, indivisible unit of execution in an MQSeries product. Frequently, each machine in an MQSeries network runs a single queue manager. In this case, the queue manager effectively represents the machine.

11.1.1 The default queue manager

To reduce the dependency of application code on the environment in which it is executing, MQSeries applications can connect to a queue manager without knowing its name. To do this requires a *default queue manager*. The default queue manager is the one to which a connection is made if the queue manager name is left blank on the MQI call by which applications connect to MQSeries.

Some implementations restrict a given computer system to running only a single queue manager. In this case, that queue manager is the default. Other implementations, however, allow multiple queue managers to execute concurrently on a single system. Such configurations might be used to allow production and test systems to run alongside one another. It might also be used to exploit additional performance available from multiprocessor machines. Whatever the motivation, if multiple queue managers are executing on a single system, one of them is chosen to be the default.

11.2 Queues

The basic resources managed by each queue manager are its queues. There are several different types of queue, as we shall see shortly. Every queue, regardless of its type, has attributes associated with it. The name of a queue, for example, is considered to be one of its attributes. Other attributes include the maximum number of messages allowed to be waiting on the queue, the maximum size of a message, and whether or not triggering is enabled for the queue.

The sequence in which messages are retrieved from a queue can also be controlled by an attribute on many MQSeries queue managers. Messages are usually delivered in the sequence in which they arrived on the queue. However, messages can also carry priority information, and this can be used instead to determine the sequence of delivery. Messages of the same priority are still delivered in arrival sequence. Priority delivery could be used, for example, to ensure that critical messages arrive at a server immediately, rather than having to wait behind ordinary messages.

There are a number of different kinds of queue. We will look at each of them later in this section. Before doing that, we need to examine some features which influence the behavior of queues.

11.2.1 Persistence

Messages placed on queues may be persistent or nonpersistent. Once a persistent message has been written to a queue, it survives, even if the queue manager is stopped or the system fails. Persistent messages are hardened. This means that there will be a copy of the data on disk storage. The precise mechanism used for hardening messages depends on the particular implementation of MQSeries. However, the important point is that, once the queue manager has restarted, processing of persistent messages can continue.

By contrast, nonpersistent messages do not survive across queue manager restarts. Such messages are not hardened. MQSeries guarantees that nonpersistent messages will not be retained after a restart. In some circumstances, MQSeries actually has to discard such messages. At first sight, this may seem somewhat bizarre. However, the intermediate steps in an application might exploit nonpersistent messages because of this behavior. Application designers know that the messages are discarded in the event of a failure.

Nonpersistent messages tend to be processed more quickly by queue managers, because they do not need to be hardened.

11.2.2 Syncpoints

Messages can be placed on queues or retrieved from them using syncpoint processing. Syncpoints identify the scope of logical units of work. GET and PUT operations specifying that syncpoints are to be used cause all changes to the queues to be remembered. Only when the application indicates successful termination of the unit of work, by issuing a commit call, are the changes made permanent. If the application aborts the unit of work, or if it fails, affected queues are returned to the state they were in before the logical unit of work began. Messages written to output queues are discarded and messages read from input queues reappear on them.

11.2.3 Local queues

A *local queue* is a queue which actually resides on the queue manager to which the application is directly connected. A local queue is held in the main memory and disk storage of the local system. The local queue manager controls access to its local queues.

Applications can put messages onto local queues and retrieve messages from them. In addition, they can inquire the values of the attributes of such queues and update some of them. Manipulation of queue attributes usually requires the appropriate security authorizations to have been granted.

11.2.4 Remote queues

A remote queue is one which belongs to a queue manager other than the one to which the application is directly connected. Access to this queue involves the local queue manager and the remote queue manager in some form of communication. This communication involves the Message Mover programs, which we will look at shortly.

Applications can perform operations against remote queues, but there are some restrictions. For example, messages can be placed on remote queues, but cannot be read from them. The general MQSeries model is such that queues are local to the applications which read from them.

Applications can reference remote queues in two distinct ways. The explicit method involves the application supplying both the queue manager and queue names on the MQI call used to open the queue. This requires the application to know or to be able to discover the name of the queue manager holding the queue. Because this involves the application knowing more than absolutely necessary about the configuration of the system, there is a second method. In this, a definition of the remote queue is held in the local queue manager. By opening this queue, the application is connected to the remote queue. The local definition of the remote queue holds sufficient information to allow the local queue manager to identify the appropriate remote queue manager.

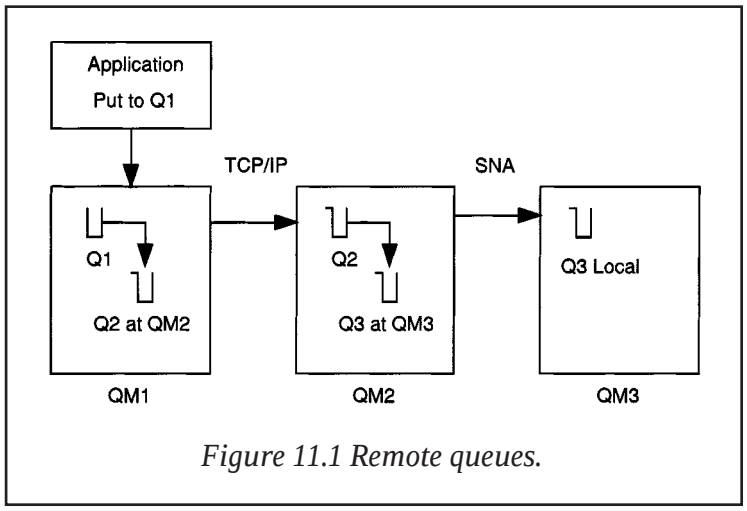


Figure 11.1 Remote queues.

The target queue on the remote queue manager does not have to be a local queue. It may itself be a local definition of yet another remote queue. Figure 11.1 shows an example. The application puts a message onto Q1. The local queue manager QM1 has a definition of Q1 as a queue on remote queue manager QM2. Together with a mover program, it transmits the message to Q2 at QM2. Queue manager QM2 has a definition of Q2 as a queue on remote queue manager QM3. Again, using a mover program, it arranges to send the message to Q3 at queue manager QM3. This is a local queue, and the message will remain on the queue until an application reads it.

There are a number of reasons why this multihop forwarding might be useful. On a network such as TCP/IP, all connected machines are directly addressable. The network itself handles the routing necessary to reach any node. However, to cross from one network type to another (for example, SNA) requires routing function in the middleware layer, as we saw in Sec. 9.1.3.5. The ability to chain remote queue definitions is the basis by which MQSeries provides such routing. In the example in Fig. 11.1, QM2 is running in a machine which bridges a TCP/IP network and an SNA network. The definition of Q2 at QM2 is all that is required to allow messages to transfer from the TCP/IP network to the SNA network.

In addition to providing cross-network routing, chaining remote queue definitions can be used to reduce the amount of administration required when a major reorganization of queues and queue managers is needed.

Because the use of remote queues always involves communications between queue managers, some additional resources do need to be defined. These additional resources include *channels*, which we will look at in Sec. 11.7, and *transmission queues*, which we will cover next.

11.2.5 Transmission queues

Transmission queues are the mechanism by which queue managers arrange to forward messages to remote queue managers. We saw in the previous section that the queue manager works with the mover programs when sending data to remote queues. When a queue manager receives a request to put a message on a remote queue, it actually puts it on the transmission queue associated with the queue manager which is the target for the request. The name of this queue manager may have been supplied by the application or it may have been deduced from the local definition of the remote queue.

A transmission queue forms one end of the link between a pair of queue managers. All messages whose immediate destination is a particular queue manager can be placed in the same transmission queue, regardless of the queue for which they are destined. Only one transmission queue is needed for moving messages from one queue manager to another. It is, however, possible to have multiple links between two queue managers, perhaps offering different classes of service. Each link requires a separate transmission queue.

Transmission queues are processed by the mover programs, as we will see shortly. These programs have the responsibility for transmitting messages reliably between queue managers. Movers can be thought of as special-purpose application programs which process messages on transmission queues.

11.2.6 Dynamic queues and model queues

In addition to the fixed queue definitions, which we have already discussed, MQSeries provides the ability for applications to create queues dynamically for use while they are executing. For example, an application which is a client of some service may choose to create a dynamic queue for the server to use to reply to requests it sends. The alternative would be a permanent queue definition. To simplify the set of parameters needed when creating a dynamic queue, these queues are based on a template called a *model queue.* The model queue must exist before dynamic queues based on it can be used. The model queue defines the attributes which the dynamic queue will have when it is created. A dynamic queue is created when an application issues a request to open it and references the appropriate model queue as the object being opened.

There are two kinds of dynamic queue. They differ in their lifetime and recovery characteristics. Temporary dynamic queues are deleted and any messages on them are lost when they are closed by the application that created them. Also, such queues do not support persistent messages and are not recovered if the queue manager fails. Permanent dynamic queues, on the other hand, are not deleted until an application closes them, specifying that they should be removed. This operation can be carried out by an application other than the one which created the queue. Permanent dynamic queues can be recovered after a queue manager failure and can support having persistent messages placed in them.

Distribution lists exist to allow applications to send messages to multiple destinations with only a single MQI call. A distribution list is a list containing the names of the queues to which the message is to be sent. Such lists can contain the names of local queues or of local definitions of remote queues. Using such lists, an application can send the same message to multiple destinations using a single call.

11.2.7 Dead letter queues

Occasionally, situations may arise in which the queue manager is unable to deliver a message to its intended destination. In this case it must resort to storing the message on a *dead letter queue.* The appearance of messages on dead letter queues usually indicates some problem with the system or its configuration. For example, a message could arrive at the queue manager destined for a queue which does not exist. Similarly, a message might arrive for a queue that is known to the queue manager but is in a state in which it cannot accept messages. It might be full, for example, or it might have been set not to allow messages to be added to it. Not all failing PUT operations necessarily result in messages being placed in a dead letter queue. For example, it is unnecessary to put the message in a dead letter queue when the error is related to a queue on the local queue manager of the application making the request. In this case, the error can be reported directly to the application in the return code of the call to MQSeries.

There are situations in which there is no other way to report the error. For example, the message may already have traversed the network to reach the machine on which a remote queue resides, only to find that the queue is full. The application which originally put the message executed on a machine different from the one which discovered the error. In addition, it may well not be executing any longer. When there is no other way to report the error, the queue manager makes use of dead letter queues to hold the messages, rather than simply discarding them. This way, not only is the problem visible, but there is a chance to correct the fault and redirect the messages in appropriate fashion.

As we shall see shortly, dead letter queues are the last resort for dealing with errors encountered on remote systems. Applications can make explicit use of other MQSeries facilities for monitoring the successful transmission and receipt of important messages. We will look at these facilities when we discuss the different types of report messages available, in Sec. 11.3.1.

11.2.8 Alias queues

Alias queues provide alternative naming for local queues, local definitions of remote queues, or distribution lists. They are simply a name-to-name mapping to allow queues to be referred to in alternative ways. They provide an extra level of indirection between an application and the queues it uses. MQSeries already provides significant insulation of an application from the details of the underlying system and, in particular, the configuration of network resources. Alias queues allow even the name of the underlying queue to be changed without the applications being affected.

11.3 Messages

From the point of view of the queue manager, the life of a message begins when it is first placed on a queue and ends when it is finally read from the queue. Of course, messages may pass through many queue managers as they traverse the network. Also, as we have seen, notions of when a message is considered to be actually on a queue are modified by considerations associated with syncpoint processing.

Messages carry attribute information with them as they pass through the system. This information includes the name of the queue on which replies should be sent, as well as identifiers which can allow an application to relate this message to others. Also included is the message context, and information about the origin of the message and about the user associated with the request. Almost all of the information carried with the message can be specified when the message is put onto a queue. Changing some of it, such as message context, may require that the user or application is operating with the appropriate level of security authorization.

11.3.1 Types of messages

MQSeries defines four basic types of messages. Applications are free to define additional types. The four basic types are:

Request messages
Reply messages
Datagram messages
Report messages

Requests are messages which require replies. Messages sent to a server to retrieve or update information would probably be request messages. Request messages need to carry information which will allow the reply to be routed appropriately.

Reply messages are responses to requests. Information from the associated request message determines the destination of the reply message. The applications involved in processing requests and replies control the relationship between the messages. The queue manager itself does not enforce any rules, but the information transmitted with the messages does enable applications to relate and process requests and replies.

Datagrams are messages to which no reply is anticipated or required.

Reports are messages generated in response to some error occurring in the system. They can be generated by applications, if desired. Some report messages are generated by the queue manager. This can occur, for example, if a message cannot be delivered. This situation might occur if the destination queue is on a remote machine and is full or absent. The error cannot be detected by the local queue manager of the application which put the message originally. Only later is the problem discovered, resulting in the generation of the report message. The queue manager always sends the report message to the queue specified in the original message as being for replies.

The queue manager uses report messages for other purposes, as well as for indicating errors. Messages can have expiration times associated with them. This allows the queue manager to purge them if they have been in the system for some period of time and yet have not been processed. Applications choose whether messages are to be allowed to expire and, if so, how long the expiration period should be. If a message expires, the queue manager deletes it and a report message is generated and sent to the queue for replies, specified in the expired message.

Another use of report messages is for confirmation of the arrival of a message. It is possible for applications to request that they be notified when the message actually arrives at its destination queue. This is known as *confirmation of arrival*. Similarly, an application can request to be notified when another application actually reads the message from the queue. This is called *confirmation of delivery*. In either case, the queue manager generates a report message to the queue for replies specified in the original message.

Application-defined message types can also be used. The queue manager itself takes no action based on the message type. Consequently, arbitrary values can be specified. A range of values has been set aside for application use. These values can be used, for example, to distinguish between different types of application message on the same input queue.

11.3.2 Trigger messages

Trigger messages are a particular kind of message generated by the queue manager itself. They form the core of the mechanism, provided by the queue manager, by which applications can be started automatically when work arrives for them to do. It is often preferable to start an application only when there is work for it to process, particularly if the work load is sporadic. Naturally, initial responses take longer, because the application has to be started before it can process the first message. However, unless the work load for an application is fairly constant, there will be periods of time during which it is idle but nonetheless using some system resources.

The mechanism by which applications can be started on demand is known as *triggering*. The idea is that the queue manager recognizes the arrival of messages on queues for which triggering has been enabled and informs a special-purpose application, known as *a trigger monitor*. This application is responsible for starting the application which will process the newly arrived message. The overall scheme is shown in Fig. 11.2.

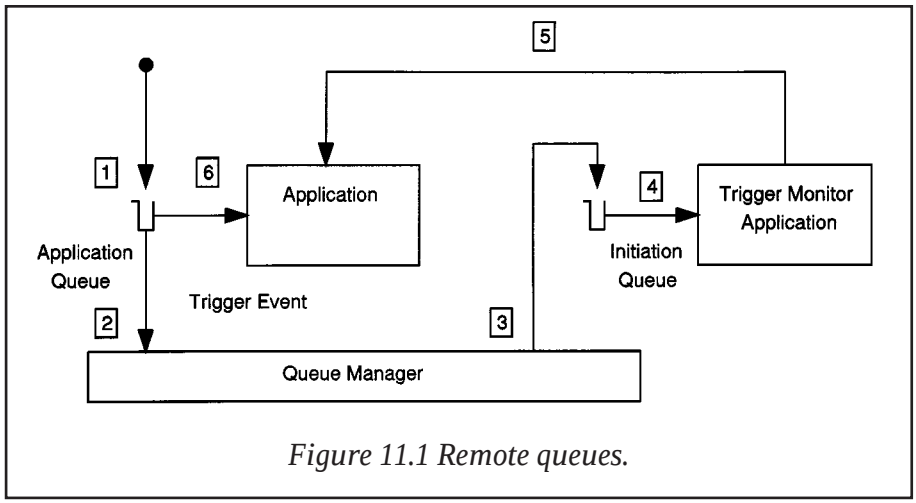


Figure 11.1 Remote queues.

In step 1, a message arrives on an application queue for which triggering is enabled. In step 2, the queue manager examines the circumstances surrounding the message arrival and determines that it constitutes a trigger event. We will look at the sorts of rules which can be applied shortly. In step 3, the queue manager generates a trigger message and places it on another queue. This is the queue processed by the trigger monitor application and is known as an *initiation queue*. Its name is part of the information associated with the application queue and held because triggering is enabled. The trigger monitor application reads the message from the initiation queue at step 4. In step 5, it issues an operating-system-dependent command to start the application associated with the queue which was triggered. The association between the queue and the application to be started is held in a *process definition*. The name of this definition is one of the attributes of the queue itself. The process definition contains details which the trigger monitor needs to start the application. The information includes the name of the application, data to be passed to the application when it is started, and environment information. This information is made available to the trigger monitor in the trigger message it receives from the queue manager. Finally, in step 6, the application reads the message which had arrived on its queue.

Because a single trigger monitor can service many different queues rather than having many idle applications waiting for work, only the trigger monitor needs to be running continually. The use of process definitions means that the trigger monitor does not need to know anything about the applications it is starting. It gets all the information it needs from the trigger message. Trigger monitors can be completely generic. Conversely, if there are specialized requirements, different trigger monitors can replace the standard one supplied with MQSeries. A trigger monitor is just another MQSeries application. For triggering to be used, there must be at least one trigger monitor running. There is, however, no limit to the number of monitors which can be executing simultaneously.

11.3.2.1 Triggering rules.

Because triggering might be used in a variety of ways, MQSeries tries to provide a comprehensive set of options to cover all requirements. Three basic types of trigger are defined:

- Trigger on first message
- Trigger on every message
- Trigger when specified number of messages waiting

The first two kinds of trigger leave triggering enabled. The third type disables triggering once a trigger has been generated. The first kind of triggering generates a trigger whenever a message arrives on an empty queue. This trigger is intended to be used when the application being triggered continues processing until the queue is empty and then terminates. It is triggered again when work arrives on the queue. This kind of triggering is appropriate where work arrives fairly sporadically but where the messages tend to come in batches. This might be true, for example, where remote sites have collected data over some period of time and are transmitting it to a central site to bring central records into line. Some retail operations work this way, transmitting records of the day's sales to central operations during the night.

The second kind of triggering generates a trigger whenever a message arrives on the queue. This kind of triggering is intended for applications which process a single message and terminate. The application is triggered every time a message arrives. Consequently, there is more overhead associated with processing each message, in this case, than when compared with triggering on first message. Clearly, this kind of trigger is appropriate for rare messages which must be processed immediately. If two such messages arrive in quick succession, a copy of the application will be started for each one, allowing them to be processed together. This kind of triggering is appropriate for rare messages which are, nonetheless, very important to the operation and which require immediate processing.

The final kind of triggering causes a trigger whenever there are more than a specified number of messages on the queue. In addition to generating the trigger message, this kind of triggering also disables further triggering once a trigger has been generated. This

kind of triggering is intended for starting applications which will tend to run for a reasonable period of time. The arrival of messages on the queue indicates that there will probably be work to do for a reasonable period of time. The triggered application processes the messages queue to it. Once the queue is empty, it waits for more work to arrive. If no work arrives within a predetermined time, the application rearms the trigger for the queue and exits. It will be triggered again once more work arrives. We will see an example of an application which uses this kind of triggering in Sec. 12.3.

Although we have given examples here of how these triggering rules can be used, the trigger mechanism itself makes no demands on the behavior of the applications. Application designers can choose to use the trigger mechanisms in whatever way they wish.

11.3.3 Message context

In client-server systems, server applications usually perform work on behalf of client applications or end users. Servers normally execute in a context quite different from that of the applications and users for whom they are performing work. In particular, a server's security profile will often be quite different from that of the user or client on whose behalf it is performing work. Servers frequently have higher levels of authorization. To prevent unauthorized access to resources, it is obviously necessary for the server to be able to test access rights based on the capabilities of the entity submitting the work rather than its own authorizations. To enable the server to do this, *context* information can be passed in MQSeries messages. Context information describes the submitter of the request.

Context information in messages is in two parts. *Identity context* contains the information we just discussed. There is also space for accounting information, which applications can use to apportion work load to users or groups. Identity context is normally passed on unchanged if a message moves from queue to queue under application control. This allows all work to be performed in the context of the original user. Similarly, if one application processes a message and sends others in order to get work done, it can set the context in the new messages to be the same as the original. Again, all work performed will be in the context of the user who submitted the original message.

In contrast, *origin context* holds information about the application which actually put the message on the queue. Origin context is normally supplied automatically by the queue manager. Applications can override it if required.

Applications which modify or set context normally need to be given additional authorization to do so for the appropriate queues.

11.4 Processing Application Requests

MQSeries provides an application programing interface (API) as a set of functions which can be called from the C and COBOL programming languages. The API consists of a relatively small number of individual calls, known as *verbs*. Each verb has a number of basic parameters. Flexibility of operation stems from the nature of some of the parameters. These are data structures containing fields which define the options which can be set to modify the behavior of some of the calls. In every case, these parameters can be given sensible default values very simply. Consequently, basic applications can be written very easily and the subtleties of the options need be addressed only when specific behavior is required.

The kinds of function available through the MQSeries API include:

- Establishing a connection to the local queue manager
- Opening a queue to work with
- Getting a message from a queue
- Putting a message onto a queue
- Querying or changing the attributes associated with a queue

The interface uses *handles* to represent the objects being manipulated. Handles are values which are meaningless to the caller but which are used internally by MQSeries to distinguish instances of the objects being referenced. The interface is described in more detail in Chap. 12, which also includes examples of its use.

When processing API calls, MQSeries takes the same basic approach, regardless of the particular call involved. First, parameters are verified for correctness. In some cases, the values associated with one parameter may influence the values which must be supplied for another. If the parameters are correct, processing can occur. In the newer implementations of MQSeries, processing takes place in such a way that important MQSeries data, such as the contents of the queues, is isolated from the application code. This prevents application code from inadvertently or maliciously accessing and modifying the underlying data structures used internally by MQSeries.

The mechanism chosen for isolation is implementation dependent. Some systems—for example, IBM's MVS and OS/400—provide the ability to change access control authority when executing certain pieces of code. The MQSeries code executes with higher authority than the application code and, consequently, has access to data structures denied to the application code. In other systems, notably the newer UNIX-based implementations, processing of the MQSeries data structures takes place in a process separate from that in which the application is executing. Again, this isolates and protects the internal MQSeries data.

11.5 Managing the Queues

The task of maintaining the message queues involves two main kinds of operation. First, the semantics of the MQSeries API have to be performed. For example, when a message is put onto a queue for which triggering is enabled, the appropriate trigger message may need to be generated. Second, the data associated with the structure and content of the messages and the queues must be managed. This set of operations on the queue data itself is what gives each MQSeries product many of its performance and reliability characteristics. These include the ability to recover from system failures without loss of data. The techniques employed when dealing with queue-related data are often similar to those used by database or transaction managers. The aims are very similar in that data loss must be avoided without the performance overhead associated with synchronous I/O operations to disk devices. The underlying storage medium chosen to hold hardened data depends on the particular implementation. However, it is usually the file system of the underlying platform. Since file systems are rarely high-reliability components, MQSeries products take special actions when hardening data to ensure that its data is highly reliable and recoverable.

11.6 Systems Management

The management of an MQSeries system is mainly concerned with the creation, modification, and removal of queue managers and queues. In addition, the connections between queue managers also need to be maintained. These connections are known as channels. We'll cover them in more detail shortly.

The creation of a Message Queue Manager is a fundamental operation and one which typically is not carried out very often. Usually, initial queue manager creation is carried out immediately after the product itself is installed. Once a queue manager has been created and is running, its own queuing mechanisms become a component of its management. A standard systems management queue is created. Messages of appropriate format placed on this queue can be used to perform systems management. Since these messages can originate from remote machines, this mechanism provides for distributed management of MQSeries queue managers.

There are two main ways to manage the newer MQSeries queue managers. The first is the systems administration application, an interactive program providing a user interface through which management tasks can be carried out. On systems where a graphical user interface is available, the systems administration application makes use of it. Otherwise, a full-screen, panel-style interface is used. The systems administration application can be used to perform all MQSeries systems management tasks for the local queue manager and the majority of such tasks for suitable remote queue managers. This is true even where the remote queue manager is running on a machine of a different architecture.

The second mechanism for administration is based on files containing commands, rather than an interactive application. These MQSeries Script Commands are portable. The same syntax is used by any MQSeries product which supports them, regardless of the underlying machine and operating system. Consequently, a script written on one system can be executed on a different one, allowing another mechanism by which remote administration can be performed. Scripts are moved between machines by normal file transfer techniques. Scripts are particularly useful when bulk changes need to be made, especially if several queue managers are involved. In addition, they can be used to store particular sets of operations which might need to be repeated.

There is a third way to perform systems administration on some MQSeries products. Some of the operations are made available as commands, using the command interface of the underlying operating system. As with MQSeries script commands, normal commands made available this way can be combined together in files. For example, on UNIX-based implementations, these are shell scripts and on OS/2, they are command files. However, these kinds of files are less portable than MQSeries script commands. A UNIX shell script cannot be executed on OS/2, for example, whereas an MQSeries script command file would run equally well on both. Script portability was a major factor in the decision to provide MQSeries script commands as the prime mechanism for noninteractive systems administration.

11.7 Moving Messages Between Queue Managers

Messages move between queue managers via programs known as *message channel agents*, or more simply, *movers*. Movers are essentially normal, message-queuing applications. Movers read the messages to be transmitted from a queue. This queue is a transmission queue. We discussed these in Sec. 11.2.5. Messages read from the transmission queue are sent over a particular communications connection, known as a channel. At the other end of the channel is another mover program which receives the data. This program assembles the data it receives over the channel into messages and puts them onto the appropriate queues on the target queue manager. Information about the target queues flows over the channel with the messages.

11.7.1 Channels

Channels are the communications links over which data flows between queue managers. Channels are defined as point-to-point links, even though the intervening network may be very complex. For example, the route between two machines on a TCP/IP network might involve several machines and include bridges and routers. The channel definition, however, references only the machines running the queue managers being linked. A channel implies a particular network type. A single channel cannot, for example, link one machine on an SNA network with another on TCP/IP. To achieve this kind of connection, channels need to be defined from each machine to a machine which links both networks. These sets of channel definitions provide the cross network routing function within MQSeries.

Channels provide a logical view of the underlying network topology. They map MQSeries communications concepts onto the underlying network implementation. Channel definitions and the channels themselves are managed via normal MQSeries systems administration. It is possible to stop and start channels without affecting operation of the queue manager. Naturally, if a channel is stopped, work cannot be transferred over it. Messages queue up until the channel is made available once more.

11.7.2 Recovery from channel failures

The communications component of any distributed system is usually its least reliable element. Network failures occur for a variety of reasons. Components of the network, especially the medium used for wide area transmission, have to operate in the least well controlled electrical environment. Consequently, channels and their associated software are the parts of an MQSeries system most likely to be affected by an external failure. Indeed, one of the benefits of messaging and queuing is that these kinds of failures can be isolated to a component which is not intimately connected with the applications. Failures on the network affect only the channels and the message channel agents.

Detection of and recovery from channel failures is the task of the message channel agents. Typically, a network failure is detected when an operation to send or receive data fails. In addition to transmitting data between themselves, message channel agents record and transmit status information. When the channel once again becomes available, the message channel agents converse with each other to agree on the overall state of the transmission. Essentially, they need to determine the last packet of data successfully transmitted so that they can restart operations at the correct point. MQSeries ability, to guarantee that messages are not lost or transmitted twice, is heavily dependent on the process by which channel agents agree where to restart transmissions.

The only effect of a channel failure, or of the deliberate stopping of a channel, is that messages to be transmitted are delayed until the channel is available once more.

11.8 Summary

In this chapter, we have taken a very brief tour of the facilities of typical MQSeries queue managers. We looked at the queue manager itself and covered the characteristics of the various kinds of queues which are supported. We discussed various types of messages, including the reports which can be generated in response to specific events or remote errors. Finally, we looked briefly at the tasks the queue manager performs and covered the work performed by message channel agents.

In the next chapter, we will look in more detail at the application programming interface provided by MQSeries and we will see some examples of its use in practice.

Chapter 12

The Message Queue Interface

The interface by which application programs access message-queuing function in MQSeries products is known as the *Message Queue Interface* or simply the MQI. In this chapter, we will look at the facilities offered by the MQI and see how to use them in some simple examples. The information in this chapter is, of necessity, incomplete. The aim is to introduce the MQI and to show how it can be used. It is not feasible to reproduce all the technical information associated with MQSeries here. Full details of the interfaces and the data structures associated with them can be found in the reference documentation associated with the MQSeries products themselves.

12.1 A Quick Tour of the Message Queue Interface

There are relatively few operations which applications need to perform when interacting with queues. Clearly, putting messages onto queues and retrieving messages from queues are the most common. However, in any comprehensive programming interface, there needs to be some mechanism by which applications can exercise control over the details of the processing. As a simple example, consider the interface to a simple, two-dimensional drawing package. The interface to the operation to draw a straight line can be extremely simple. The only information required is some definition of the coordinates of the points joined by the line. However, the line itself may have many attributes associated with it. For example, it may have a specific color, a particular style, and a width. The challenge, when designing a programming interface, is to allow the application the full range of control over the processing, without obscuring the basic simplicity of the operations.

The MQI tries to solve the interface problem by offering a relatively small number of operations, but allowing the detailed behavior of each to be modified by options and descriptors. Default values for these are provided, allowing normal operations to be carried out without the necessity for programmers to understand the subtleties of the interface until they need to be used.

We will look at the basic operations provided by the MQI first, and then we will consider some of the more important options available.

12.1.1 The basic verbs

The major functions provided by the MQI are termed *verbs*. There are 11 of these in the newer MQSeries implementations. The following sections describe each of these functions.

12.1.1.1 MQCONN—connect to the queue manager.

All MQSeries operations require that the application be connected to a queue manager. This connection is established via the MQCONN call. A particular queue manager can be specified by name on this call. However, applications frequently do not know and do not care which queue manager is providing their services. In this case, they can connect to the default queue manager by not specifying a name.

The result of a successful MQCONN operation is a connection handle. This is used on subsequent MQI calls to specify the particular queue manager to which the application is connected.

12.1.1.2 MQDISC—disconnect from the queue manager.

When an application has finished work and no longer needs a connection to the queue manager, it issues the MQDISC call. This allows the queue manager to release any resources which were being used to support the application. Any MQSeries objects which the application had been using, but which it had not explicitly released, are implicitly released by this call.

12.1.1.3 MQOPEN—open an MQSeries object.

In order to work with MQSeries objects such as queues, an application must first open them. This is analogous to the notion of opening a file before reading or writing to it. Queues are the most important class of MQSeries object for most applications. There are others, though. For example, there is a queue manager object, which can be opened. The class of an MQSeries object determines the kinds of operation which can be performed. For example, messages can be retrieved from MQSeries queue objects but not from queue manager objects. Both kinds of object do support inquiry of attributes, however. Applications specify the particular object to be opened by filling in an object descriptor. We will look at object descriptors in more detail in Sec. 12.1.3.

When opening an object, an application must specify what it intends to do with it. Once again, this is analogous to the situation when opening files. The intent, specified by the application, is used to determine the kind of locking and serialization which the queue manager must perform in order to protect the integrity of the data in the object. The intent is passed to the MQI in the options parameter. We will look at these options in Sec. 12.1.2.

The result of a successful MQOPEN call is an *object handle*. This is used in subsequent MQI calls to specify the object on which the requested operation is to be performed.

12.1.1.4 MQCLOSE—close an MQSeries object.

When an application no longer needs to use an MQSeries object, it can issue an MQCLOSE call against it. This allows the queue manager to release resources associated with the object. Objects still open when an application disconnects from the queue manager are implicitly closed. Once again, this behavior is analogous to that of typical file systems.

12.1.1.5 MQPUT—put a message on an MQSeries queue.

Applications put messages onto MQSeries queues with the MQPUT call. In addition to the data itself, applications specify a descriptor for the message. This controls a number of aspects of the processing of the message within MQSeries. The descriptor travels with the message, and the information it contains may be used to control processing anywhere on its journey through the network. For example, in the event that the message eventually proves undeliverable, the system's behavior can be specified. In addition, the message descriptor holds values associated with the message, such as its priority, type, and persistence properties. A persistent message is one which survives queue manager restarts.

Applications can also control the details of the PUT operation itself, by using the options parameter. This is used to specify, for example, whether or not the operation is part of a logical unit of work. Also, applications can use the options to specify that the context associated with the message should be taken from a source other than the application itself. This is very useful for server applications, which may need to send messages to other servers and have them processed in the context of the original request rather than that of the server itself. As we saw in Sec. 11.3.3, the context of a message includes information such as the identifier of the user associated with it.

12.1.1.6 MQGET—get a message from an MQSeries queue.

Applications retrieve messages from MQSeries queues with the MQGET call. The parameters to this call are very similar to those for MQPUT. Along with the data, the message descriptor associated with the message is returned. Once again, the application can control the details of processing using an options parameter. For example, the options control whether MQSeries should return immediately if there is no message available or wait for a message to arrive. They also control a number of other aspects, including whether or not the operation is part of a logical unit of work. In addition, many MQSeries products support conversion of the data in the message between the format used on the machine from which it originated and that used on the machine receiving it. If so, that conversion can be requested via the options parameter.

12.1.1.7 MQPUT1.

This call can be thought of as a combination of the MQOPEN, MQPUT, and MQCLOSE calls. It puts a single message on a named queue, which does not have to have been opened. It is a convenience, reducing the amount of code necessary for applications which write only a single message to a particular queue. Since the queue is opened and closed automatically every time this call is made, applications which put many messages to the same queue should use the separate MQOPEN, MQPUT, and MQCLOSE calls instead.

The MQPUT1 call takes a set of parameters which is a combination of those for MQOPEN and MQPUT, allowing the application to have full control over the operation just as it would for the separate calls. Since each MQPUT1 call involves an implicit MQOPEN call and an implicit MQCLOSE call, applications which output multiple messages to a queue should open it explicitly and use the MQPUT call.

12.1.1.8 MQINQ—inquire about the attributes of an MQSeries object.

The MQINQ call allows the current values of attributes of MQSeries objects to be queried by an application. MQINQ can be used to inquire on the attributes of queues, name lists, process definitions, and the queue manager itself. For example, this call can be used to determine how many messages are currently waiting on a particular queue. The call allows multiple attributes for a given object to be retrieved together. Also, some attributes have numeric values while others are represented as character strings. Consequently, the interface is a little unusual, involving two arrays and a character buffer, together with information on their size. The first array contains *selectors*. Each of these is an integer representing the particular attribute whose value is to be returned. The second array is space in which integer attributes will be returned. The buffer provides space for the values of character attributes to be returned. Character attributes are returned as blank padded, fixed-length character arrays concatenated together in the buffer. A single call to MQINQ might, for example, request five integer attributes and three character attributes. The integer attributes will be returned in the array provided by the application, and the character attributes will be placed in the character buffer, concatenated together.

The interface to MQINQ and to the related routine MQSET is designed to maximize the flexibility of manipulation of attribute values. For example, the syntax of these calls will not need to change simply because more attributes or more kinds of objects may be defined in future versions of MQSeries.

12.1.1.9 MQSET—set the attributes of an MQSeries object.

The MQSET call allows the current values of attributes of MQSeries objects to be altered by an application. For example, it can be used to specify the number of messages which must be waiting on a queue before triggering occurs. The call allows several updates to be processed together. As with MQINQ since some attributes have numeric values and others are represented as character strings, the interface to MQSET also involves two arrays and a character buffer, together with information on their size. Once again, the first array contains selectors. Each of these is an integer representing the particular attribute whose value is to be updated. The second array contains the values which will be used to update the integer attributes. The buffer contains the values which will be used to update the character attributes. These are concatenated together. A single call to MQSET might, for example, update two integer attributes and a character attribute.

12.1.1.10 MQCMIT—commit operations performed under syncpoint.

MQSeries allows operations to be grouped into local logical units of work. The results of such groups of operations are not made permanent until the logical unit of work is committed. MQCMIT is the call used to commit such a unit of work. Effectively, operations carried out under syncpoint are pended until the commit occurs. Messages retrieved from queues are not deleted, but are made unavailable to other applications until the commit decision has been made. Likewise, messages put onto queues are stored, but are cannot be retrieved until the commit has been processed.

MQSeries products can also support operations being carried out as part of a distributed unit of work, under the control of a transaction manager. We will look at how how this is achieved in Sec. 12.5.

12.1.1.11 MQBACK—backout operations performed under syncpoint.

The MQBACK call partners the MQCMIT call. It is used to back out a series of operations performed within a local logical unit of work. Applications call this function when a logical unit of work is being backed out. In response, the queue manager restores the state of any affected objects to the state that existed before the unit of work started. The result is as if the changes made as part of the logical unit of work had never occurred.

12.1.2 Extending the function—MQI options parameters

In this section, we look in a little more detail at the way in which the MQI operations can be tailored by the options that can be specified.

12.1.2.1 Options on MQOPEN.

When opening an MQSeries object, particularly if it is a queue, the *options* parameter allows applications to specify the kind of operations they wish to perform. The following list describes some of the possibilities:

Open for input and allow the queue to be shared
Open for input exclusively for this application
Open for browsing

Open for output
Open to allow inquiry of attributes
Open to allow attributes to be updated
Open using the default options associated with the object

Some of these options are naturally exclusive. Clearly, it is not valid to specify that a queue be opened for shared read access and exclusive read access at the same time. However, it is perfectly permissible for a queue to be opened for both input and output simultaneously as well as for inquiry.

The options parameter also allows applications to request the ability to manipulate the context associated with messages which they will place on queue objects. This allows applications to send messages which will process using the context of messages they themselves received. This is useful to allow, for example, the user identifier associated with the original request to be propagated throughout all the programs involved in processing it. Applications need authority to be able to pass context, since otherwise it would be trivial to bypass security checks by claiming to be a user with the appropriate authority. An application's ability to set context is verified when it opens the queue in question. Its request to be granted the ability to manipulate context is specified by options such as:

Allow context to be specified by the application
Allow context to be set from the most recently retrieved message on another queue

An application can also request that the MQOPEN call be checked for valid authority using a user identifier other than the one under which the application is running. This allows a server to open an object with the access rights of the user submitting the request, rather than with those of the server itself.

The ability to change context and to use alternate user identifiers is dependent on the application itself having been granted authority to do so. If the application does not have sufficient authority, the MQOPEN call fails.

Finally, an application can request to be notified if the queue manager is in the process of normal shutdown when it opens an object. During a normal shutdown, the queue manager prevents new connections but allows all connected applications to complete normally before closing itself down. Most applications will probably not want to continue if the queue manager is about to stop. However, some critical operations might need to be completed before the queue manager terminates. Consequently, applications can choose whether or not the MQOPEN call should fail if the queue manager is in the process of quiescing.

12.1.2.2 Options on MQCLOSE.

The options on MQCLOSE are associated with dynamic queues. Dynamic queues are queues created during the execution of an application. Such queues may be temporary, in which case they are deleted when the application terminates, or they may be permanent, in which case some method of subsequently removing them is required. The method provided by the MQI relies on options used when the queue is closed. One option will delete the queue as long as there are no messages on it and it is not in use. A second option purges any remaining messages and deletes the queue.

12.1.2.3 Options on MQGET.

So far, the options we have looked at have been simple flag values specifying the detailed behavior of the MQI. The options on MQGET are a little more complex. They consist of a data structure containing both flags and value fields, some of which may be modified by the queue manager. The basic options include the following:

Wait for a message to arrive if none is available
Return immediately if there is no message available
The operation is part of a logical unit of work
Browse the first message on the queue
Browse the next message on the queue
Retrieve the message at the current browse position
Lock message
Unlock message
Accept truncated data
Convert the message data
Fail if queue manager is quiescing

The browse options allow an application to scan the messages on the queue without removing them. This way, an application can search the queue for a message meeting a particular set of criteria. The lock option, used with browse, allows an application to reserve each message that it browses. This way, it can be sure that, even if several applications are reading the same queue, the mes-

sage will be available for it to read from the queue should it decide to do so. The option to accept truncated data allows an application to read the message from the queue even if the buffer it supplied to contain the message data is too short. The final option in the list allows the MQGET operation to fail if the queue manager is in the process of shutting down. This option has the same behavior as when used on MQOPEN.

In addition to these options, the get message options structure includes a field in which the length of time to wait for a message to arrive can be specified. This wait interval is specified as an integer number of milliseconds, allowing quite short delays to be requested.

Finally, the get message options structure contains a field in which the queue manager returns the resolved name of the queue from which the message was actually read. This will be different from the name specified in the MQOPEN call if the queue which was opened was an alias queue, for example.

12.1.2.4 Options on MQPUT.

As with the options used when getting messages, the options on MQPUT consist of a structure containing flags and values. Once again, some of these values are returned by the queue manager. The basic options include the following:

- The operation is part of a logical unit of work
- There is no context associated with the message
- Context should be taken from the specified queue
- Context should be taken from the message descriptor
- Fail if queue manager is quiescing

When context is to be taken from a queue, a valid object handle for the queue must also be specified in the appropriate field of the put message options. If this is done, the context associated with the message most recently read from that queue will be used on the message being put. If the application is setting the context itself, it fills in the appropriate fields of the message descriptor, which we will examine shortly.

Once again, the application can request that the operation fail if the queue manager is in the process of shutting down.

Finally, the put message options structure contains fields in which the queue manager returns the resolved name of the queue on which the message was put and the resolved name of the queue manager holding that queue.

12.1.3 Descriptors

In this section, we look at the two descriptors used in the MQSeries application programming interface.

12.1.3.1 Message descriptor.

The *message descriptor* contains information which travels with the message across the network to its destination. The first major element of the descriptor is a set of options flags describing the kinds of report messages which are to be associated with the message. Report messages can be generated in response to various events which may occur during the life of the message within the network. The following list shows some of the events:

- Deletion of the message because it was undelivered when its expiry time passed
- Arrival of the message on its destination queue
- Delivery of the message to an application
- Occurrence of an exception

An exception might be caused, for example, if after traversing the network, the destination queue for the message is already full when it arrives.

In addition to the generation of report messages, the message descriptor also allows applications to control some of the data which those reports will contain. For example, it is possible to request that the message identifier of the message generating the report be used as the correlation identifier in the report message. This allows an application receiving the report message to identify the particular message which gave rise to the report. Similarly, the message identifier associated with the report can be taken from the message causing the report, if desired.

The message descriptor holds the message type associated with the message. This may contain a system value, indicating that the message is a request message, a datagram, and so on, or it may contain an application-defined value. Application-defined values can be used to distinguish messages of different types on a single queue. By browsing the queue, an application could, for example, retrieve only messages of a particular type.

The priority associated with the message is also held in the message descriptor. It can be given an explicit value, or it can be specified as taking the default value for the queue.

The expiry time for the message is held in the message descriptor. It can be set to unlimited, meaning that the message will never expire and will remain in the system indefinitely, even if undelivered. Alternatively, it can be used to limit the lifetime of the message. If the message has not been read from its destination queue when its lifetime expires, it becomes eligible to be discarded. If it is discarded, and the appropriate report option is set, a report message is generated. This message is sent to the destination specified in the original message as being that to which replies should be sent. Often, this will be the original source of the message.

The message descriptor also includes information used when converting the message data. Conversion is necessary if the machine architectures of the sending and receiving systems differ. It is also necessary if the national language of the systems differ. The information held in the message descriptor includes details of the encoding applied by the machine on which the message originated. It also includes details of the character set used to encode character data. Since knowledge of the structure of the data in the message is necessary to allow conversion to be carried out, a format name is also sent in the message descriptor. This allows the conversion service in the queue manager to apply the appropriate conversions. Some format names are reserved. They are used to define messages which queue managers themselves use to communicate with one another.

The message descriptor contains a field indicating the persistence properties of the message. Persistent messages survive queue manager restarts, whereas nonpersistent messages do not. As with priority, each queue has a default value for persistence and messages can be specified to use this value.

The message identifier field of the message descriptor allows a specific value to be associated with the message. This is normally used to identify reply messages and to allow them to be associated with the original message which gave rise to them. Applications can specify this field or allow the queue manager to generate a unique value for it. In addition to the message identifier, the correlation identifier is also in the message descriptor. This identifier is usually used to associate messages with one another. Both the message identifier and the correlation identifier can be used independently or together to select which messages are retrieved during MQGET operations.

To enable correct routing of the messages which result from processing of a particular message, the message descriptor also contains fields in which the queue name and queue manager name for replies to the message can be specified. If the queue manager name is not given explicitly, the queue manager from which the reply will be sent must be able to determine the target queue manager name from its own tables in the usual way.

Context information travels with the message in the message descriptor. The message context includes information about the user and the application creating the message, and the date and time when the MQPUT operation occurred.

12.1.3.2 Object descriptor.

The object descriptor is used when opening an object with the MQOPEN call or when putting a single message on a queue with MQPUT1. This descriptor defines the name and type of the object to be used. The object type could, for example, be queue, queue manager, or process definition. If the object does not belong to the local queue manager, the name of the queue manager owning the object is also specified in the object descriptor.

To allow applications to open objects using authority other than that associated with the application itself, the object descriptor contains a field which can be used for an alternate user identifier. This will be used in authority checking if the appropriate flags were set in the open options and if the application has appropriate authority to request use of an alternate identifier.

To support the use of dynamic queues, the object descriptor contains a field to hold a dynamic queue name. To open a dynamic queue, the appropriate model queue is specified in the object name field and the name for the new dynamic queue is given in the dynamic queue name field.

12.2 Using Basic MQI Functions

In this section, we look at the use of the basic MQI functions, including connecting to the queue manager, opening the appropriate queue, putting messages onto queues, and getting messages from queues. In a later section, we will look at more complex operations, including triggering and the use of logical units of work.

To provide a framework for the discussion, we will use a simple application program. This program transfers small files. Although not particularly representative of typical MQSeries applications, it does illustrate the basic operations in a very simple way. The example is written in C. However, the code is simple and the explanatory text is written with the needs of those who do not program in C in mind.

12.2.1 The file transfer example application

The file transfer application has a single function. It transfers a file from one place to another. To do this, it makes use of two MQSeries application programs. One, called mqftp, reads the file and sends its data as an MQSeries message. The other, mqftpr, reads these MQSeries messages and writes them to the destination files.

To perform the transfer, several pieces of information are required. First, the name of the file to be sent is needed. Second, the name to be used to store the file in its new location is required. Third, the destination queue manager needs to be known. Finally, the name of the queue to be used must be known.

To get this information, the application takes the following approach. The name of the file to be transferred, its name after the transfer, and the target queue manager are all taken as input parameters to the sending program, mqftp. The queue to be used is fixed. The receiver program, mqftpr, always reads a particular queue. File data is always sent on that queue.

We said that this application is not representative of typical MQSeries applications. It is worth examining one of the reasons here, before we look at the source code in detail. One of the main reasons that the example application is untypical is that it uses a fixed queue name. Also, the user must specify the queue manager to which the file will be transferred. While this kind of information is required to transfer files to remote computer systems and, indeed, is very similar to that needed by the standard TCP/IP-based ftp application, it is unusual for MQSeries. Many MQSeries applications do not care which queues they service. In addition, when replying to requests, all the information necessary to determine where to send replies is contained in the input messages they receive. MQSeries provides the means by which application code can be completely independent of the particular queues being used. Triggering, for example, removes the need even to know which input queue is to be serviced. Exploiting these features of MQSeries makes for more flexible systems and obviates the need to change program code in order to alter the flow of messages through it. However, for simplicity in the example application code, we have ignored these advantages of MQSeries to allow us to give concrete examples of the use of the basic MQI functions.

It is time to look at the first component of the application in detail. We will start with the sender program.

12.2.2 The sender program **mqftp**

The sender program is shown in Fig. 12.1. At the top of the listing is a block of comments identifying the program. The real code starts immediately after the comment labeled (A). The lines which begin #include cause the compiler to read the files mentioned to access standard definitions frequently used in C programs. These definitions allow the use of standard C facilities, for example, reading and writing files. The #include statement after point (B) in the code causes the compiler to read the cmqc.h file, which contains all the definitions required for an application to use the MQI.

At point (C) in the code, the #define statements associate literals with specific values. For example, the compiler will associate MAX_FILE_SIZE with the actual value 4096. This is the value we use to limit the size of the files which are transferred by the program. Consequently, this application will transfer whole files up to 4096 bytes in length, or the first 4096 bytes of any longer files.

The next section of code, at point (D), defines the correct usage for the function named TerminateProg. We will see the implementation of the routine later. This function gathers together the actions needed to end the program cleanly, whether normally or in error. The C language allows the return value and the parameters to a function to be defined separately from the function itself. That is being done here. The definition is known as a *function prototype*. It allows the compiler to verify that, elsewhere in the code, the function is being invoked correctly. It is especially useful when the implementation and the call are in separate compilation units. In this case, the definition shows that the function takes four parameters. The first is a character string called endmsg. The specification char* indicates that endmsg is a pointer to a character. This is the normal way to reference strings in C. A string is a sequence of characters. The end of the string is marked by a character with a binary value of zero. The second parameter to TerminateProg is of type PMQHOBJ. This is a type defined by the MQI header file, cmqc.h, and is used to hold a pointer to an object handle. This is the kind of handle used to identify an object, such as a queue, to a queue manager. The third parameter to TerminateProg is of type PMQHCONN. This is also a type defined by the MQI header file, and is used to hold a pointer to a connection handle. This is the kind of handle used to identify a connection to a queue manager. The final parameter is of type MQLONG. Once again, this is a type defined by the MQI header file. It represents the completion code of the program.

The main entry point to the program is at point (E) in the code. This is where the code starts to execute. The definition of the main entry point is fixed for C programs. The parameters are argc, a count of the number of parameters passed to the program, and argv, an array of character strings, one for each parameter. As we shall see shortly, the program will use these strings to identify the file to be copied and its new name, as well as the queue manager to which it will be copied.

```
/*****************************************************************/
/* Module Name: mqftp.c                                          */
/*                                                               */
/* Function:    This program is the sender half of the file transfer    */
/*              example. It reads the specified file and sends its      */
/*              contents as a series of messages to the                 */
/*              File.Transfer.Queue on the specified queue manager      */
/*                                                               */
/* Usage:       Invoked from command line as                    */
/*                                                               */
/*   mqftp <queue_manager> <from_file> <to_file>                 */
/*                                                               */
/* where                                                         */
/*   <queue_manager>    is the queue manager to send the file to        */
/*   <from_file>        is the file to send                       */
/*   <to_file>          is the name to use for the new copy of the file */
/*                                                               */
/* NOTES:                                                        */
/* 1) This code is only an example. For simplicity it performs only     */
/*    very basic error checking and has a crude user interface.          */
/*****************************************************************/

/*****************************************************************/
/* (A) Include definitions for standard C language functions     */
/*****************************************************************/
#include        <stdio.h>
#include        <stdlib.h>
#include        <string.h>

/*****************************************************************/
/* (B) Include definitions for MQSeries                          */
/*****************************************************************/
#include        <cmqc.h>

/*****************************************************************/
/* (C) Define constants used in this program                     */
/*****************************************************************/
#define MAX_FILE_SIZE  4096

/*****************************************************************/
/* (D) Prototypes for local functions                            */
/*****************************************************************/
void TerminateProg(char     *endmsg,
                   PMQHOBJ  pQueue_Handle,
                   PMQHCONN pConnection_Handle,
                   MQLONG   End_Reason) ;

/*************************/
/* (E) Main Entry Point   */
/*************************/
int main(int argc, char**argv)
{
  /*****************************************************************/
  /* (F) Declare Structures used in the MQ API calls. Also, initialize  */
  /* the descriptors and options.                                  */
  /*****************************************************************/
  MQOD          Queue_Descriptor    = {MQOD_DEFAULT} ;
  MQMD          Message_Descriptor  = {MQMD_DEFAULT} ;
  MQPMO         Put_Message_Options = {MQPMO_DEFAULT} ;
  MQHCONN       Connection_Handle   = MQHC_UNUSABLE_HCONN ;
```

Figure 12.1 The sender program for the file transfer example.

```
MQHOBJ        Queue_Handle        = MQHO_UNUSABLE_HOBJ ;
MQLONG        Open_Options ;
MQLONG        CompCode ;
MQLONG        Reason ;

/*************************************************************************/
/* (G) Declare other variables used in the program                       */
/*************************************************************************/
FILE          *fp ;
int           nbytes ;
char          *qmgr_name ;
char          *from_file ;
char          *to_file ;

struct
{
  char          to_file[128] ;
  unsigned int  Data_Length ;
  MQBYTE        Buffer[MAX_FILE_SIZE] ;
} F_Transfer_Msg ;

/*************************************************************************/
/* (H) Retrieve command line arguments.                                  */
/*     argv[1] is the name of the target queue manager                   */
/*     argv[2] is the name of the file being sent                        */
/*     argv[3] is the name of the to_file                                */
/*************************************************************************/
if (argc != 4)
{
  printf("Usage:\n") ;
  printf(" mqftp queue_manager_name from_file to_file\n") ;
  exit(16) ;
}
qmgr_name = argv[1] ;
from_file = argv[2] ;
to_file   = argv[3] ;

/*************************************************************************/
/* (I) Open the file to be transferred                                   */
/*************************************************************************/
fp = fopen(from_file, "r") ;
if (fp == NULL)
{
  printf("Could not open from_file %s\n", from_file) ;
  exit(16) ;
}

/*************************************************************************/
/* (J) Connect to the local queue manager (i.e. default)                 */
/*************************************************************************/
MQCONN ("",
        &Connection_Handle,
        &CompCode,
        &Reason) ;
if (CompCode == MQCC_FAILED)
{
  TerminateProg("Could not connect to default queue manager",
                &Queue_Handle,
                &Connection_Handle,
                Reason) ;
```

Figure 12.1 (Continued)

```
  exit(16) ;
}

/*****************************************************************/
/* (K) Construct the descriptor for the file transfer queue      */
/*****************************************************************/
strncpy(Queue_Descriptor.ObjectName,
        "File.Transfer.Queue",
        MQ_Q_NAME_LENGTH) ;

strncpy(Queue_Descriptor.ObjectQMgrName,
        qmgr_name,
        MQ_Q_MGR_NAME_LENGTH) ;

/*****************************************************************/
/* (L) Open the queue on which we will send the data             */
/*****************************************************************/
Open_Options = MQOO_OUTPUT ;
MQOPEN (Connection_Handle,
        &Queue_Descriptor,
        Open_Options,
        &Queue_Handle,
        &CompCode,
        &Reason) ;

if (CompCode == MQCC_FAILED)
{
  TerminateProg("Could not open File.Transfer.Queue",
                &Queue_Handle,
                &Connection_Handle,
                Reason) ;
}

/*****************************************************************/
/* (M) Send the content of the file                              */
/*****************************************************************/
strncpy(F_Transfer_Msg.to_file,
        to_file,
        sizeof(F_Transfer_Msg.to_file)) ;


/*****************************************************************/
/* (N) Read the data from the file                               */
/*****************************************************************/
nbytes = fread(F_Transfer_Msg.Buffer, 1, MAX_FILE_SIZE, fp) ;
if (nbytes == MAX_FILE_SIZE)
{
  printf("WARNING: Copy of file may have been truncated\n") ;
}

if (nbytes != 0)
{
  /***************************************************************/
  /* (O) Send the file data via the queue                        */
  /***************************************************************/
  F_Transfer_Msg.Data_Length = nbytes ;

  MQPUT (Connection_Handle,
         Queue_Handle,
         &Message_Descriptor,
```

Figure 12.1 (Continued)

```
            &Put_Message_Options,
            (sizeof(F_Transfer_Msg) - MAX_FILE_SIZE + nbytes),
            &F_Transfer_Msg,
            &CompCode,
            &Reason) ;

      if (CompCode == MQCC_FAILED)
      {
        TerminateProg("Could not send file data",
                      &Queue_Handle,
                      &Connection_Handle,
                      Reason) ;
      }
    }

    /*****************************************************************/
    /* (P) Close the file                                            */
    /*****************************************************************/
    fclose(fp) ;

    /*****************************************************************/
    /* (Q) Terminate with all work finished                          */
    /*****************************************************************/
    TerminateProg("Processing Completed Normally",
                  &Queue_Handle,
                  &Connection_Handle,
                  MQRC_NONE) ;

    return(MQRC_NONE) ;
}

/*****************************************************************/
/* (R) Function to terminate the program                         */
/*****************************************************************/
void TerminateProg(char     *endmsg,
                   PMQHOBJ  pQueue_Handle,
                   PMQHCONN pConnection_Handle,
                   MQLONG   End_Reason)
{
    MQLONG CompCode ;
    MQLONG Reason ;
    MQLONG Close_Options ;

    /*****************************************************************/
    /* (S) Issue the message associated with termination             */
    /*****************************************************************/
    printf("Sender: %s. Reason was %ld\n",
           endmsg,
           End_Reason) ;

    /*****************************************************************/
    /* (T) Close the queue                                           */
    /*****************************************************************/
    if (*pQueue_Handle != MQHO_UNUSABLE_HOBJ)
    {
      Close_Options = MQCO_NONE ;
      MQCLOSE (*pConnection_Handle,
               pQueue_Handle,
               Close_Options,
               &CompCode,
```

Figure 12.1 (Continued)

```
                &Reason) ;

      printf("Sender: Queue Close Reason Code was %ld\n",
             Reason) ;
    }
    /*****************************************************************/
    /* (U) Disconnect from the local queue manager                   */
    /*****************************************************************/
    if (*pConnection_Handle != MQHC_UNUSABLE_HCONN)
    {
      MQDISC (pConnection_Handle,
              &CompCode,
              &Reason) ;

      printf("Sender: Disconnect Reason Code was %ld\n",
              Reason) ;
    }
    exit(End_Reason) ;
  }.
```

Figure 12.1 (Continued)

At point (F) are declarations of data structures used within the main program itself. For example, the first line declares an object descriptor. Its type, MQOD, is defined in the MQI header cmqc.h. The structure contains fields which we discussed in Sec. 12.1.3.2. Because we will use it when manipulating the queue on which we will send the messages, we call the variable itself Queue_Descriptor. In this declaration and the two which follow it, the structures are initialized with default values. These values are defined in the MQI header file, cmqc.h. In the case of the object descriptor, the default values are represented by the literal MQOD_DEFAULT, defined in cmqc.h. The braces which surround the literal are a required part of the C language syntax for initialization of a variable which is a structure. This method of assigning a default value to an MQI structure can be used only where the variable is being declared. Later in the program, if we should need to reset individual fields within the structures, we will need to assign values explicitly.

We initialize the Connection_Handle variable to the value MQHC_UNUSABLE_HCONN, defined by the MQI header file, showing that we do not yet have a usable connection. Likewise, we initialize the Queue_Handle. During program termination, we can test for these values to see if we managed to open the queue or if we managed to connect to the queue manager.

The next section of the code, at point (G), declares other variables used in the program. The variable F_Transfer_Msg is a type we have not met before. It is known as a ***structure***, which is why it begins with the keyword struct. A structure is a collection of individual variables which are treated as a single entity. F_Transfer_Msg is the structure which holds the information which we will send as an MQSeries message when transferring a file. Within F_Transfer_Msg there is a field called to_file in which the name of the target file will be placed. This is an area 128 bytes in length. The field Data_Length will hold the amount of file data being transferred. Finally, the Buffer field, an area MAX_FILE_SIZE in length, will hold the data from the file itself.

Program execution proper starts at (H), where the command line arguments are retrieved. The test on the count of arguments, argc, establishes whether the correct number has been supplied. In C, the term != is used to mean “not equal to.” The count is always one more than the actual number of parameters. The first argument is always the name of the program itself. If the count is wrong, the program prints a simple message about how it should be used. The printf function is used to do this. In this form, it prints two simple character strings. The notation \n, used within the strings, causes a new line to be started after the values have been printed.

The lines which follow the if statement cause the variables qmgr_name, from_file, and to_file to be set from the corresponding input parameters. The notation, involving square brackets, used in these assignment statements selects the appropriate element of the argv array to assign to each variable. At (I), the program attempts to open the file to be copied. The name of the file is in variable from_file. The fopen function will open the file and return a file pointer if successful. The value of this pointer will be stored in variable fp. The second parameter to fopen indicates the mode in which the file is being opened. The value “r” indicates that the file is being opened for reading. If fopen fails, the value of fp is set to the special value NULL. The program detects this with the if statement at point (I). The term == is used in C to express equality in tests, whereas = is used to specify assignment. If the file could not be opened, the

program prints an error message, using printf, and terminates, passing a return code of 16 to the operating system. The printf for the error message illustrates the way in which values as well as literals can be output. The term %s within the string indicates a substitution point. In this case, it is for the string variable from_file. Consequently, the error message will contain the name of the file which could not be opened.

At (J), we finally reach code which calls the MQI. The first call has to be to connect to the local queue manager. The first parameter to MQCONN is the name of the queue manager. As this is an empty string, we are requesting connection to the default local queue manager. The second parameter is the connection handle we declared earlier. This is filled in by the mqconn call and will allow us to identify the connection on subsequent MQI calls. The leading ampersand on the reference to this parameter, and indeed to CompCode and Reason as well, indicates that the address of the variable is being passed into MQCONN. This is necessary because of the mechanism which C uses to update parameters passed in by callers. The final two parameters appear on all MQI calls. CompCode is the completion code from the result and Reason is an associated code giving more detail about any error that may have occurred. For example, if CompCode contains the value MQCC_WARNING, Reason might contain MQRC_ALREADY_CONNECTED, indicating that a request to connect has been ignored because the application is already connected to the queue manager. This is only a warning. Execution can continue, and the connection handle returned will be valid. On the other hand, if CompCode contains the value MQCC_FAILED, it indicates that a real error has occurred. Reason, in this case, might contain MQRC_Q_MGR_STOPPING, indicating that the queue manager is refusing connections because it is about to stop. If the connection call fails, MQSeries sets the connection handle to the value MQHC_UNUSABLE_HCONN to mark it as incapable of being used.

If the connection operation does fail, for any reason, the program ends by calling the TerminateProg function. We will see how this function works later. In essence, it issues the message passed as its first parameter, ends the use of any MQSeries resources in use, and finally exits from the program.

The next block of code, at point (K), sets up two of the important values for the object descriptor to be used to open the queue. The name of the queue and the name of the queue manager are filled in. Although the program is connected to the default queue manager, the queue to be opened can be on any queue manager which is known about. Remember that the queue manager to which we are connected has tables which allow it to route messages to the correct destination. As long as there are appropriate definitions which specify where the messages are to be sent, the open call will succeed. The mechanism to set the names involves the standard function strncpy. This copies character strings, but limits the maximum length which can be transferred. In this case, the queue name is a literal. The queue is called File.Transfer.Queue. The maximum length of queue names is given by MQ_Q_NAME_LENGTH. Once again, this is defined in cmqc.h. The queue manager name, on the other hand, is one of the input parameters. We saved it earlier in the variable qmgr_name. The maximum length of a queue manager name is MQ_Q_MGR_NAME_LENGTH.

Now that we have established the object descriptor values, we can open the queue. The MQOPEN call is at point (L). The open options are set to MQOO_OUTPUT, requesting that we can put messages onto the queue. The first parameter to the MQOPEN call itself is the connection handle we received when connecting to the queue manager. The second parameter is the object descriptor which specifies the queue we want to open. Next come the open options and then the address of the variable which the MQOPEN call will update with a handle for the object, if the call is successful. Once again, the final pair of parameters are the completion code and reason. If the open call fails, we invoke the TerminateProg routine, as before, to end the program.

At point (M), we start to set up variables for the transfer of the file. The strncpy function is used to copy the variable to_file, containing the name of the file to be copied to, into the to_file field of the F_Transfer_Msg data structure, which represents the message. The notation F_Transfer_Msg.to_file specifies this field of the structure.

At point (N), we read the data from the file into the field Buffer of the message. The fread function does this. Variable fp is the file pointer we received when we opened the input file back at point (I). The term MAX_FILE_SIZE specifies the maximum number of bytes we want fread to transfer from the file. The literal 1, which precedes it in the parameter list to fread, indicates that specification of how much data can be transferred is in bytes. Variable nbytes is set by the call to the number of bytes actually transferred from the file. If this count is less than MAX_FILE_SIZE, it means we have read the entire file. However, if the count equals MAX_FILE_SIZE, there may be more data in the file than we can transfer. In this case, we issue a warning about possible file truncation by using printf. The text of the warning is slightly vague, since, if the file is exactly MAX_FILE_SIZE in length, we will issue the warning without truncating the file. There are ways to distinguish these cases, but, as this is an example of MQSeries programming rather than of file handling, we will not concern ourselves with them here.

If data has been read from the file, we issue the MQPUT call at point (O). We use the connection handle Connection_Handle and the object handle Queue_Handle given to us by MQSeries in response to previous successful calls. We use the default message descriptor and put message options which we initialized back at point (F). The expression (sizeof(F_Transfer_Msg) - MAX_FILE_SIZE + nbytes)

deserves some study. We need to tell MQSeries the total size of the message being sent by the MQPUT call. We could simply specify the total size of the F_Transfer_Msg structure. However, for files smaller than MAX_FILE_SIZE, this results in the transfer of unused data across the network, which is wasteful. The expression evaluates the amount of data which must be transferred in order to send the contents of the file. It works like this. The term sizeof(F_Transfer_Msg) causes the compiler to calculate the number of bytes needed to represent the entire F_Transfer_Msg structure. This includes the Buffer field. If we subtract the size of Buffer, we are left with the size of the rest of the structure. Finally, adding nbytes, the size of the data we read from the file, yields the size of the part of F_Transfer_Msg which we do need to send. If the MQPUT operation fails, we end the program via TerminateProg. Once the file has been sent, at (P) we close the input file.

We have now finished the work the program has to do. We terminate by calling TerminateProg at point (Q). The value MQRC_NONE passed as return code is the one used by MQSeries when there has been no error.

The TerminateProg routine itself begins at point (R). The first statement defines the function entry point, specifying the parameters it expects. This is very closely related to the function prototype we saw back at point (D).

The first executable statement is at point (S), where printf is used to display the termination message passed to the routine as the endmsg parameter. This call to printf also displays the value of the End_Reason parameter. The specification %ld in the first parameter to printf tells it that End_Reason is a long integer, allowing printf to format and print it appropriately.

At point (T), we close the queue, if it was successfully opened. We do this by testing the value of the queue handle pointed to by the pQueue_Handle parameter. The value MQHO_UNUSABLE_HOBJ is the one we used to initialize the object handle originally. It is also the value which MQSeries uses to mark an object handle when it is not open, for example, after a failing MQOPEN call or after a successful MQCLOSE. We display the final return code from the MQCLOSE call using printf.

Although we have shown it in the example, it is not strictly necessary to close the open MQSeries resources explicitly this way, though it is recommended. Disconnection from the queue manager will implicitly close any open resources. Such implicit close operations use default values for the close options. As a consequence, there may be situations in which an application does need to issue MQCLOSE calls explicitly, because it needs to control the close options. Think of MQCLOSE in much the same way as you would think of using fclose or any other file closing function. It is not strictly necessary to use these functions, but it is good programming practice to do so wherever possible.

At point (U) we disconnect from the queue manager. Once again, we test the relevant handle to see if we did manage to connect. In this case, we check to see if the pConnection_Handle parameter has the value MQHC_UNUSABLE_HCONN. If not, we did manage to connect. The return code from the MQDISC call is displayed using printf.

Finally, the exit call stops the program and returns End_Reason to the operating system as the program's termination code.

12.2.3 The receiver program MQFTPR2

The listing of the receiver program is given in Fig. 12.2. The code from point (A) to point (F) is essentially identical to that in the sender program mqftp. At (G), where the additional variables used in the program are declared, we see the first real differences. We do not need variables to hold the name of the file being copied from or the queue manager name. There is a new variable called ended, however. This will be used to control a loop later in the program.

At point (H), the program verifies the command line with which it was started. In contrast to mqftp, this program takes no command line parameters. If it detects that any were supplied because argc is other than 1, it issues an error message via printf and terminates.

Connection to the default queue manager takes place at point (I), with any failure causing the program to terminate via TerminateProg.

At point (J), the object descriptor for the file transfer queue is constructed. In contrast to the way this is done in mqftp, only the queue name itself is specified here. The queue must be local to the mqftpr program. It is a rule within MQSeries that queues from which messages are being read are local to the application reading them. Consequently, we do not need to specify a queue manager when opening the queue.

The program opens the queue at point (K). The options for the MQOPEN call are set to MQOO_INPUT_EXCLUSIVE, requesting that only this program be allowed to read the queue. As before, any failure is handled by calling TerminateProg.

At point (L), the program starts to execute a loop. The while statement causes all the statements between the { immediately following it and the matching } just before point (Q) to be executed continuously while the expression in parentheses is true. We will see shortly the conditions under which ended becomes nonzero.

```
/*****************************************************************/
/* Module Name: mqftpr.c                                         */
/*                                                               */
/* Function:    This program is the receiver half of the file transfer  */
/*              example. It reads messages from a queue, interpreting   */
/*              them as data to be placed into files. The program reads */
/*              File.Transfer.Queue on the local queue manager.         */
/*                                                               */
/* Usage:       Invoked from command line as                     */
/*                                                               */
/*   mqftpr                                                      */
/*                                                               */
/* NOTES:                                                        */
/* 1) This code is only an example. For simplicity it performs only     */
/*    very basic error checking.                                 */
/*****************************************************************/

/*****************************************************************/
/* (A) Include definitions for standard C language functions     */
/*****************************************************************/
#include        <stdio.h>
#include        <stdlib.h>
#include        <string.h>

/*****************************************************************/
/* (B) Include definitions for MQSeries                          */
/*****************************************************************/
#include        <cmqc.h>

/*****************************************************************/
/* (C) Define constants used in this program                     */
/*****************************************************************/
#define MAX_FILE_SIZE  4096

/*****************************************************************/
/* (D) Prototypes for local functions                            */
/*****************************************************************/
void TerminateProg(char     *endmsg,
                   PMQHOBJ  pQueue_Handle,
                   PMQHCONN pConnection_Handle,
                   MQLONG   End_Reason) ;

/*************************/
/* (E) Main Entry Point   */
/*************************/
int main(int argc, char**argv)
{
  /*****************************************************************/
  /* (F) Declare Structures used in the MQ API calls. Also, initialize  */
  /* the descriptors and options.                                  */
  /*****************************************************************/
  MQOD          Queue_Descriptor    = {MQOD_DEFAULT} ;
  MQMD          Message_Descriptor  = {MQMD_DEFAULT} ;
  MQGMO         Get_Message_Options = {MQGMO_DEFAULT} ;
  MQHCONN       Connection_Handle   = MQHC_UNUSABLE_HCONN ;
  MQHOBJ        Queue_Handle        = MQHO_UNUSABLE_HOBJ ;
  MQLONG        Data_Length ;
  MQLONG        Open_Options ;
  MQLONG        CompCode ;
  MQLONG        Reason ;
```

Figure 12.2 The sender program for the file transfer example.

```
/*****************************************************************/
/* (G) Declare other variables used in the program               */
/*****************************************************************/
int           ended = 0 ;
FILE          *fp ;
int           nbytes ;

struct
{
  char          to_file[128] ;
  unsigned int  Data_Length ;
  MQBYTE        Buffer[MAX_FILE_SIZE] ;
} F_Transfer_Msg ;

/*****************************************************************/
/* (H) Verify command line                                       */
/*     There are no command line arguments                       */
/*****************************************************************/
printf ("Receiver Starting\n") ;

if (argc != 1)
{
  printf("Usage:\n") ;
  printf(" mqftpr\n") ;
  exit(16) ;
}

/*****************************************************************/
/* (I) Connect to the local queue manager (i.e. default)         */
/*****************************************************************/
MQCONN ("",
        &Connection_Handle,
        &CompCode,
        &Reason) ;
if (CompCode == MQCC_FAILED)
{
  TerminateProg("Receiver: Could not connect to default queue manager",
                &Queue_Handle,
                &Connection_Handle,
                Reason) ;
  exit(16) ;
}

/*****************************************************************/
/* (J) Construct the descriptor for the file transfer queue      */
/*****************************************************************/
strncpy(Queue_Descriptor.ObjectName,
        "File.Transfer.Queue",
        MQ_Q_NAME_LENGTH) ;

/*****************************************************************/
/* (K) Open the queue on which we receive data                   */
/*****************************************************************/
Open_Options = MQOO_INPUT_EXCLUSIVE ;
MQOPEN (Connection_Handle,
        &Queue_Descriptor,
        Open_Options,
        &Queue_Handle,
        &CompCode,
```

Figure 12.2 (Continued)

```
        &Reason) ;

if (CompCode == MQCC_FAILED)
{
  TerminateProg("Could not open File.Transfer.Queue",
                &Queue_Handle,
                &Connection_Handle,
                Reason) ;
}

/*****************************************************************/
/* (L) Read messages from the queue                              */
/*****************************************************************/
while (ended == 0)
{
  /*************************************************************/
  /* (M) Retrieve a message from the queue                     */
  /*************************************************************/
  Get_Message_Options.Options = MQGMO_WAIT ;
  Get_Message_Options.WaitInterval = 30000 ;
  memcpy(Message_Descriptor.MsgId, MQMI_NONE, sizeof(MQMI_NONE)) ;
  memcpy(Message_Descriptor.CorrelId, MQCI_NONE, sizeof(MQCI_NONE)) ;

  MQGET (Connection_Handle,
         Queue_Handle,
         &Message_Descriptor,
         &Get_Message_Options,
         sizeof(F_Transfer_Msg),
         &F_Transfer_Msg,
         &Data_Length,
         &CompCode,
         &Reason) ;

  if (CompCode == MQCC_FAILED)
  {
    ended = 1 ;
  }
  else
  {
    /***********************************************************/
    /* (N) Report what is happening                            */
    /***********************************************************/
    printf("Receiver: Transferring file to %s\n",
           F_Transfer_Msg.to_file) ;

    /***********************************************************/
    /* (O) Open the file to be transferred.                    */
    /***********************************************************/
    fp = fopen(F_Transfer_Msg.to_file, "w") ;
    if (fp == NULL)
    {
      printf("Receiver: Could not open output file %s\n",
             F_Transfer_Msg.to_file) ;
      printf("   Data will be discarded\n") ;
    }
    else
    {
      /*********************************************************/
      /* (P) Write out the data.                               */
      /*********************************************************/
```

Figure 12.2 (Continued)

```
            nbytes = fwrite (F_Transfer_Msg.Buffer,
                             1,
                             F_Transfer_Msg.Data_Length,
                             fp) ;
            if (nbytes != F_Transfer_Msg.Data_Length)
            {
              printf("Receiver: Problem writing to file %s\n",
                     F_Transfer_Msg.to_file) ;
            }
            fclose(fp) ;

          }
        }
      }     /* End while(ended = 0) */


      /*****************************************************************/
      /* (Q) Terminate the program                                     */
      /*****************************************************************/
      TerminateProg("Processing Completed",
                    &Queue_Handle,
                    &Connection_Handle,
                    Reason) ;

      return(Reason) ;
    }

    /*****************************************************************/
    /* (R) Function to terminate the program                         */
    /*****************************************************************/
    void TerminateProg(char     *endmsg,
                       PMQHOBJ  pQueue_Handle,
                       PMQHCONN pConnection_Handle,
                       MQLONG   End_Reason)
    {
      MQLONG CompCode ;
      MQLONG Reason ;
      MQLONG Close_Options ;

      /*****************************************************************/
      /* (S) Issue the message associated with termination             */
      /*****************************************************************/
      printf("Receiver: %s. Reason was %ld\n",
             endmsg,
             End_Reason) ;

      /*****************************************************************/
      /* (T) Close the queue                                           */
      /*****************************************************************/
      if (*pQueue_Handle != MQHO_UNUSABLE_HOBJ)
      {
        Close_Options = MQCO_NONE ;
        MQCLOSE (*pConnection_Handle,
                 pQueue_Handle,
                 Close_Options,
                 &CompCode,
                 &Reason) ;

        printf("Receiver: Queue Close Reason Code was %ld\n",
               Reason) ;
```

Figure 12.2 (Continued)

```
  }
  /*****************************************************************/
  /* (U) Disconnect from the local queue manager                   */
  /*****************************************************************/
  if (*pConnection_Handle != MQHC_UNUSABLE_HCONN)
  {
    MQDISC (pConnection_Handle,
            &CompCode,
            &Reason) ;

    printf("Receiver: Disconnect Reason Code was %ld\n",
            Reason) ;
  }
  exit(End_Reason) ;
}
```

Figure 12.2 (Continued)

At point (M), a message is read from the input queue. The options for the MQGET call are set to cause the program to wait until a message is available. The length of time to wait is set to be 30 seconds. The value assigned to the WaitInterval field of the Get_Message_Options structure is in milliseconds. If no message becomes available within this period, the MQGET call will fail. The code immediately following the MQGET call detects the failure and will cause the loop to terminate by setting ended to 1. Of course, this does not actually take effect until the while statement is again executed. However, all of the rest of the processing in the loop is in the else clause of the if statement which tests for a failure in the MQGET call. Consequently, none of this code is executed if the MQGET fails.

The message identifier and correlation identifier in the message descriptor are also initialized before the MQGET call. This is necessary because they are input and output fields which are updated after a successful operation. Their values can affect subsequent calls if the message descriptor is reused, as it is in this program. If you are not explicitly using these identifiers, it is a good idea to get into the habit of resetting them just before making the call that uses the message descriptor. Programmers new to the MQI frequently fall foul of this aspect of the interface.

After a successful MQGET operation, at point (N) the program reports that it has received a request to transfer a file and displays the target file name, taken from the message which was received.

At point (O), the file is opened using fopen, as we saw in mqftp. The difference here is that the second parameter is specified as "w". This requests the ability to write to the file. If the file already exists, any data already in it is deleted. The name of the file to be opened is taken from the message structure. Any problems encountered in opening the file cause an error message to be issued but do not stop the program. The receiver program behaves like a server, staying active and processing many requests. Failure of a single request does not stop it executing.

At point (P), the file data is written out to the file from the Buffer field of the F_Transfer_Msg structure which now contains the message we obtained from the earlier MQGET call. The length of the file data is taken from the Data_Length field of the structure. This was filled in when the file was read, in the sender program mqftp. Any problems in writing the file data are reported via a printf call.

Just before point (Q) is the end of the while loop which began back at point (L). The program will remain in this loop until the variable ended becomes nonzero. This happens on any failure of the MQGET call at point (M), as we saw earlier.

At point (Q), TerminateProg is called to end the program. Since the reason that the loop ended is a failure in the MQGET call, there will always be a nonzero reason code, and this is passed to TerminateProg. Normal termination of this program is when no message has been received for 30 seconds. Under these conditions, the final reason code will be MQRC_NO_MSG_AVAILABLE.

At point (R) is the implementation of the TerminateProg function, which is essentially identical to that in mqftp. It prints the termination message, closes the queue if open, and disconnects from the queue manager if connected.

12.2.4 Running the example

To execute the file transfer example requires some resources to be defined on one or more queue managers. For the case where files are being transferred between machines, the receiver requires a local queue definition and the sender a remote queue definition.

In addition, a channel needs to be defined between the machines. While less realistic, it is perfectly possible to run the example entirely within one queue manager by executing the sender and receiver on the same machine. This requires only that we define a single local queue. Figure 12.3 shows an MQSC script which defines a suitable queue. In this script, lines beginning with an asterisk are comments. The + characters at the end of lines indicate continuation.

The script contains a single DEFINE command creating a local queue called File.Transfer.Queue. This is the queue referenced by both sender and receiver programs. The REPLACE clause specifies that any existing definition of the queue be overwritten by this one. This is useful when experimenting with different queue attributes, though obviously it must be used carefully in a production environment. The LIKE clause requests that the queue inherit all the attributes from the existing queue called SYSTEM.DEFAULT.LOCAL.QUEUE. This default queue definition is normally created at the same time as the queue manager itself. A number of other default objects are created, including a remote queue, an alias queue, and a channel. These default objects have useful attributes set, making it easy to create other, similar objects. Indeed, the only attribute we have changed is the textual description of the queue. The resulting queue will be enabled for PUT and GET operations, but only a single application will be able to open the queue for GET. This is the behavior we desire for the receiver program.

```
************************************************************************
*                                                                      *
* Script Name: mqftpsc.mqs                                             *
*                                                                      *
* Description: Define the resources needed by the file transfer        *
*              example program                                         *
*                                                                      *
************************************************************************
************************************************************************
* (A) FILE.TRANSFER.QUEUE                                              *
************************************************************************
  DEFINE QLOCAL('File.Transfer.Queue') REPLACE +
* The queue on which to base the definition
         LIKE('SYSTEM.DEFAULT.LOCAL.QUEUE') +
* Text description of the queue
         DESCR('Queue for file transfer example')
```

Figure 12.3 An MQSC script defining the resources for the file transfer example.

We create the queue, by running the MQSC program and specifying the commands in Fig. 12.3. Since the MQSC program is a command, the precise syntax used to run it is operating-system-specific. On UNIX or OS/2, for example, the syntax is:

```
runmqsc < mqftpsc.mqs
```

The command is runmqsc and the name of the file containing the MQSC script is mqftpsc.mqs. Users of PC systems and UNIX workstations will be familiar with the < symbol indicating that the command is to take its input from the specified file. As it executes, MQSC generates a report. By default, this appears on the terminal, though it can be redirected to a file or printer.

Once the queue has been created, we can execute the application programs. For example, to start the receiver on a UNIX system, the following command is all that is necessary:

```
mqftpr &
```

On OS/2, the equivalent is

```
start mqftpr
```

The difference here is simply in the syntax which these operating systems use to specify that the program be run in the background. This just means that while running, the receiver does not tie up the keyboard and display. This is important, because we need them in order to run the sender program.

Suppose that on a UNIX system, a user called jim wants to transfer a file to temporary disk storage. Also, suppose that the queue manager on which we defined File.Transfer.Queue is called qm1. The command issued would be:

```
mqftp qm1 /home/jim/file2go /tmp/newfile
```

On UNIX systems, the /tmp directory is usually provided for temporary work space for any user. This command copies file file2go from the user's own disk storage into the temporary space and renames it to newfile.

While the receiver program is running, any number of transfers can be initiated using the mqftp command. If the receiver is idle for 30 seconds, it stops.

If the receiver is not running when the sender is executed, the messages will accumulate on the queue. The sender program runs to completion normally. When the receiver is subsequently started, the transfers are completed normally. This is a very simple example of *time-independent processing*. It also illustrates that a server does not have to be available for a client to be able to execute.

Copying local files this way is, of course, not terribly useful. Operating systems have facilities for doing this sort of operation. However, the application works just as well if the receiver is on a remote machine. As we pointed out earlier, remote operation requires that a channel be available between the two queue managers involved. Although a discussion of the finer points of channel definitions is beyond the scope of this book, we will extend the file transfer example shortly when we look at triggered applications.

12.2.5 Programming style—The example was written to be easy to read. Because of this, there is considerable duplication between mqftp and mqftpr. By keeping the code as a single entity, each program can be read without the need for continual cross references to other programs. However, in real applications, common definitions, such as that for the F_Transfer_Msg structure and the value of MAX_FILE_SIZE, would be held in a single file. This file would be included by both the application programs, using a #include statement similar to those at point (A) in mqftp. Likewise, functions common to both programs would be placed in a separate file, compiled independently from the main programs and linked with them to form the final executable files. The TerminateProg function could easily be modified to be common for mqftp and mqftpr, and indeed also to the trigger monitor program we will look at shortly.

12.3 Triggered Applications

Having covered the basic use of the MQI, it is time to look at some more advanced topics. In this section, we will extend the file transfer application to make the receiver program be triggered into life by the arrival of work for it to do.

Triggering is the mechanism which MQSeries products provide to allow applications, and particularly servers, to be started when there is work for them to do. We looked at this mechanism in Sec. 11.3.2. Basically, the idea is that when some condition is met on a particular queue, the queue manager writes a trigger message to an initiation queue. A special application program, called a *trigger monitor*, reads this message and uses the information it contains to start the appropriate program to process the queue. Many application queues can be triggered by a single trigger monitor application.

To convert the file transfer example to triggered operation, we need to do several things. We must change the definition of the File.Transfer.Queue to enable triggering. We must also define the initiation queue which will be used by the trigger monitor. We must create a process definition for the receiver program. This definition provides information the trigger monitor needs to start the receiver. Remember that, in the general case, the trigger monitor must be capable of starting many different applications to run against many different queues. Finally, we must write the trigger monitor application.

12.3.1 The new definitions

Figure 12.4 shows the new MQSC commands we need to run to define the resources for the triggered version of the application. At (A), the first definition in the file is for the file transfer queue. The definition now includes extra fields which establish how the triggering operates. The TRIGGER keyword enables triggering for the queue. The clauses TRIGTYPE(DEPTH) and TRIGDPTH(1). define the conditions under which triggering occurs. In this case, the appearance of the first message on the queue will cause a trigger message to be generated. However, once this message has been generated, triggering will be disabled. This prevents a flood of trigger messages causing copies of the receiver application to be started. Remember that the receiver program does not terminate as soon as it has finished transferring a file. Instead, it waits for 30 seconds to see of there is more work to do. As we shall see later, the application reenables triggering on the queue just before terminating, ensuring that the subsequent arrival of work will cause it to be started once more. There are many options related to triggering. This particular combination is very convenient for simple server-type applications with a sporadic work load.

The queue definition contains three further clauses related to triggering. The TRIGDATA clause defines data which is passed from the queue definition to the trigger monitor when the queue is triggered. It has no meaning as far as the queue manager is concerned. It could be used to pass specific information to an application which might be triggered to process one of a number of different queues. The PROCESS clause specifies that the details of the application to be started when this queue is triggered will be found in a process definition called MQFTPR2. We will look at the process definition in detail shortly. Finally, the INITQ clause specifies that trigger messages should be sent to the queue called INITIATION.QUEUE. This queue is defined at (B) in Fig. 12.4. There is very little

to say about this definition. Initiation queues do not have any special attributes. They are normal queues used for a special purpose. In this case, we take all default attributes from the SYSTEM.DEFAULT.LOCAL.QUEUE apart from the description.

At (C) is the process of definition which carries the information used by the trigger monitor program when starting the receiver application. The definition of File.Transfer.Queue references this process definition. The fields in the process definition have no particular meaning to the queue manager, and could be used for almost anything. However, by convention, each field is used for a specific purpose.

```
*****************************************************************
*                                                               *
* Script Name: mqftpsc2.mqs                                     *
*                                                               *
* Description: Define the resources needed by the file transfer *
*              example program with triggering enabled          *
*                                                               *
*****************************************************************
*****************************************************************
* (A) FILE.TRANSFER.QUEUE                                       *
*****************************************************************
  DEFINE QLOCAL('File.Transfer.Queue')  REPLACE                +
* The queue on which to base the definition
        LIKE('SYSTEM.DEFAULT.LOCAL.QUEUE')                     +
* Text description of the queue
        DESCR('Queue for file transfer example with triggering') +
* Enable triggering for this queue
        TRIGGER                                                +
* Trigger when first message arrives, then disable triggering
        TRIGTYPE(DEPTH)                                        +
        TRIGDPTH(1)                                            +
* Trigger Data to be passed to trigger monitor
        TRIGDATA('File Transfer Example Application')          +
* Process definition associated with trigger
        PROCESS(MQFTPR2)                                       +
* Trigger regardless of message priority
        INITQ('INITIATION.QUEUE')

*****************************************************************
* (B) INITIATION.QUEUE                                          *
*****************************************************************
  DEFINE QLOCAL('INITIATION.QUEUE')  REPLACE                   +
* The queue on which to base the definition
        LIKE('SYSTEM.DEFAULT.LOCAL.QUEUE')                     +
* Text description of the queue
        DESCR('Initiation Queue')

*****************************************************************
* (C) Receiver Process Definition                               *
*****************************************************************
  DEFINE PROCESS(MQFTPR2) REPLACE                              +
* Descriptive text
        DESCR('Triggered File Transfer Receiver')              +
* Application Type
        APPLTYPE(0)                                            +
* Applicaton Identifier
        APPLICID('mqftpr2 &')                                  +
* User Data
        USERDATA(' ')                                          +
* Environment Data
        ENVRDATA(' ')
```

Figure 12.4 An MQSC script defining the resources for the triggered example.

The DESCR clause holds descriptive text which helps in system administration. The APPLTYPE clause records the application type. This is needed for environments where different kinds of application need to be started in different ways. For example, a normal program, such as the file transfer receiver, can be started by issuing a command. However, if the application to be started runs under the control of a transaction manager, such as CICS, the mechanism of invoking it might be entirely different. User-defined values can be used here to allow a specific customer installation to have its own mechanisms for initiating applications via the trigger monitor. For the file transfer example, the application type is unused. The trigger monitor supports only one kind of application, namely those which can be started as commands.

The APPLICID clause holds the details of the application to be started by the trigger monitor. In this case, the clause actually specifies the command to be issued to start the receiver program. In Fig. 12.4 we show the UNIX form of the command.

The function of the user data field, defined in the USERDATA clause, is to hold additional information which is passed to the application when it starts. Typically, this is passed to the application via its command line. The file transfer receiver does not require any parameters, and, consequently, this field is blank.

The final clause in the process definition is the ENVRDATA. Its function is to hold any data used in establishing the environment in which the application is to run. For example, on UNIX or OS/2, this might include environment variable values to be set before execution. Once again, the file transfer example does not require any such environment. In fact, as we shall see, the trigger monitor application ignores this data.

12.3.2 The trigger monitor program **trigmon**

The listing of the trigger monitor program is given in Fig. 12.5. The initial code is once again very similar to that which we have seen before. One difference, at (F), is that the variable Trigger_Message is declared as being of type MQTM. This is the special data type associated with trigger messages. Another difference, at (H), is that we pass the name of the initiation queue to the trigger monitor program as a parameter. This is in the spirit of trigger monitors, the idea being that the code should be as general as possible.

At (J), the object descriptor for the initiation queue is initialized. The name of the queue is taken from the parameter passed in when the application is started. At (K), when the queue is opened, the program specifies some options we have not seen before. The value MQOO_INPUT_AS_QDEF replaces explicit specification of shared or exclusive read. It means that this program is prepared to read the queue using whatever rule was specified when the queue was defined. Since we took default values in the script for defining the queue (Fig. 12.4), we will, in fact, have exclusive access to the initiation queue. The value MQOO_FAILIF_QUIESCING means that the MQOPEN call should fail if the queue manager is attempting to shut down. This is a good option to use in general. Although a queue manager will reject new connections while it is trying to quiesce, applications already connected are allowed to continue to completion. Obviously, some applications may have critical tasks to perform before the queue manager terminates. In most cases, however, applications can simply stop, safe in the knowledge that any queued messages will be held until the queue manager restarts. The vertical bar sign | between the options is the symbol which C uses for a logical OR operation. This combines the options together in a way which means that the MQOPEN call will honor them both. This pair of options is used here simply to illustrate some different possible values for the open options. The values used by the receiver program would work just as well here.

At (L), the loop for reading messages begins. At (M), the options for the MQGET call are established. As before, we specify that the call is to wait if no messages are available. The timeout period is set to 60 seconds. In practice, trigger monitor applications would probably use an indefinite wait period, specified by the MQWI_UNLIMITED value. As on the MQOPEN call, we request that the MQGET call fail if the queue manager is trying to stop. As in the receiver program, any failure of the MQGET call causes the program to end, via TerminateProg. This includes the case when the MQGET times out with no trigger message available. This is convenient for experimentation, because it means that the trigger monitor will terminate cleanly if we simply leave it alone.

If a trigger message is successfully retrieved, the code at (N) extracts the information it contains. One small difficulty with using the MQI from C language programs is that the MQI represents character data as fixed-size arrays, blank padded on the right, rather than C language strings. As we have already mentioned, C language strings have variable length, the end being marked by a character with value of decimal zero. The function CvtArray2Strg performs the required conversion, stripping the unnecessary trailing blanks and properly terminating the strings. Its action will not be described here. Readers familiar with C should have no trouble understanding its operation. Readers unfamiliar with C need not worry about the details. The result is that data from the trigger message is converted into a form in which it can be manipulated naturally by a C program.

The trigger monitor program displays the data it received from the trigger message at point (O), using the printf function we have already seen many times. At point (P), it actually starts the application which has been triggered. To do this, it creates a string which will be passed to the system function. This function causes the string passed to it to be executed as a command. The effect is as if the

```
/*****************************************************************/
/* Module Name: trigmon.c                                        */
/*                                                               */
/* Function:    This program is a trigger monitor. It is capable of */
/*              starting the mqftpr file transfer receiver program */
/*              when messages arrive on its queue.               */
/*                                                               */
/* Usage:       Invoked from command line as                     */
/*                                                               */
/*   trigmon <initiation_queue>                                  */
/*                                                               */
/* NOTES:                                                        */
/* 1) This code is only an example. For simplicity it performs only */
/*    very basic error checking.                                 */
/*****************************************************************/

/*****************************************************************/
/* (A) Include definitions for standard C language functions     */
/*****************************************************************/
#include        <stdio.h>
#include        <stdlib.h>
#include        <string.h>

/*****************************************************************/
/* (B) Include definitions for MQSeries                          */
/*****************************************************************/
#include        <cmqc.h>

/*****************************************************************/
/* (C) Define constants used in this program                     */
/*****************************************************************/


/*****************************************************************/
/* (D) Prototypes for local functions                            */
/*****************************************************************/
void TerminateProg(char     *endmsg,
                   PMQHOBJ  pQueue_Handle,
                   PMQHCONN pConnection_Handle,
                   MQLONG   End_Reason) ;

void CvtCArray2Strg(char   *string_version,
                    char   array_version[],
                    MQLONG array_length) ;

/*************************/
/* (E) Main Entry Point  */
/*************************/
int main(int argc, char**argv)
{
  /*****************************************************************/
  /* (F) Declare Structures used in the MQ API calls. Also, initialize */
  /* the descriptors and options.                                  */
  /*****************************************************************/
  MQOD          Queue_Descriptor    = {MQOD_DEFAULT} ;
  MQMD          Message_Descriptor  = {MQMD_DEFAULT} ;
  MQGMO         Get_Message_Options = {MQGMO_DEFAULT} ;
  MQHCONN       Connection_Handle   = MQHC_UNUSABLE_HCONN ;
  MQHOBJ        Queue_Handle        = MQHO_UNUSABLE_HOBJ ;
  MQTM          Trigger_Message ;
```

Figure 12.5 The trigger monitor program for the file transfer example.

```
MQLONG          Data_Length ;
MQLONG          Open_Options ;
MQLONG          Close_Options ;
MQLONG          CompCode ;
MQLONG          Reason ;

/*****************************************************************/
/* (G) Declare other variables used in the program               */
/*****************************************************************/
char            *initiation_queue ;
char            QNameStrg[MQ_Q_NAME_LENGTH + 1] ;
char            ProcessNameStrg[MQ_PROCESS_NAME_LENGTH + 1] ;
char            TriggerDataStrg[MQ_TRIGGER_DATA_LENGTH + 1] ;
char            ApplIdStrg[MQ_PROCESS_APPL_ID_LENGTH + 1] ;
char            EnvDataStrg[MQ_PROCESS_ENV_DATA_LENGTH + 1] ;
char            UserDataStrg[MQ_PROCESS_USER_DATA_LENGTH + 1] ;
char            CommandStrg[MQ_PROCESS_APPL_ID_LENGTH +
                            MQ_PROCESS_USER_DATA_LENGTH + 2] ;

/*****************************************************************/
/* (H) Verify command line                                       */
/*     argv[1] is the name of the initiation queue on which      */
/*             trigger messages will arrive                      */
/*****************************************************************/
printf ("Trigger Monitor Starting\n") ;

if (argc != 2)
{
  printf("Usage:\n") ;
  printf(" trigmon <initiation_queue>\n") ;
  exit(16) ;
}
initiation_queue = argv[1] ;

/*****************************************************************/
/* (I) Connect to the local queue manager (i.e. default)         */
/*****************************************************************/
MQCONN ("",
        &Connection_Handle,
        &CompCode,
        &Reason) ;
if (CompCode == MQCC_FAILED)
{
  TerminateProg("Trig Mon:Could not connect to default queue manager",
                &Queue_Handle,
                &Connection_Handle,
                Reason) ;
  exit(16) ;
}

/*****************************************************************/
/* (J) Construct the descriptor for the initiation queue         */
/*****************************************************************/
strncpy(Queue_Descriptor.ObjectName,
        initiation_queue,
        MQ_Q_NAME_LENGTH) ;

/*****************************************************************/
/* (K) Open the initiation queue                                 */
```

Figure 12.5 (Continued)

```
/*****************************************************************/
Open_Options = MQOO_INPUT_AS_Q_DEF    |
               MQOO_FAIL_IF_QUIESCING ;
MQOPEN (Connection_Handle,
        &Queue_Descriptor,
        Open_Options,
        &Queue_Handle,
        &CompCode,
        &Reason) ;

if (CompCode == MQCC_FAILED)
{
  TerminateProg("Could not open initiation queue",
                &Connection_Handle,
                &Queue_Handle,
                Reason) ;
}

/*****************************************************************/
/* (L) Read messages from the queue                              */
/*****************************************************************/

while (CompCode == MQCC_OK)
{
  /***************************************************************/
  /* (M) Retrieve a message from the queue                       */
  /***************************************************************/
  Get_Message_Options.Options = MQGMO_WAIT |
                                MQOO_FAIL_IF_QUIESCING ;
  Get_Message_Options.WaitInterval = 60000 ;
  memcpy(Message_Descriptor.MsgId, MQMI_NONE, sizeof(MQMI_NONE)) ;
  memcpy(Message_Descriptor.CorrelId,MQCI_NONE,  sizeof(MQCI_NONE)) ;

  MQGET (Connection_Handle,
         Queue_Handle,
         &Message_Descriptor,
         &Get_Message_Options,
         sizeof(Trigger_Message),
         &Trigger_Message,
         &Data_Length,
         &CompCode,
         &Reason) ;

  if (CompCode == MQCC_OK)
  {
    /*************************************************************/
    /* (N) Extract the contents of the trigger message as NULL   */
    /*     terminated strings                                    */
    /*************************************************************/
    CvtCArray2Strg(QNameStrg,
                   Trigger_Message.QName,
                   sizeof(Trigger_Message.QName)) ;

    CvtCArray2Strg(ProcessNameStrg,
                   Trigger_Message.ProcessName,
                   sizeof(Trigger_Message.ProcessName)) ;

    CvtCArray2Strg(TriggerDataStrg,
                   Trigger_Message.TriggerData,
```

Figure 12.5 (Continued)

```
                    sizeof(Trigger_Message.TriggerData)) ;

        CvtCArray2Strg(ApplIdStrg,
                    Trigger_Message.ApplId,
                    sizeof(Trigger_Message.ApplId)) ;

        CvtCArray2Strg(EnvDataStrg,
                    Trigger_Message.EnvData,
                    sizeof(Trigger_Message.EnvData)) ;

        CvtCArray2Strg(UserDataStrg,
                    Trigger_Message.UserData,
                    sizeof(Trigger_Message.UserData)) ;

        /*****************************************************************/
        /* (O) Display the contents of the trigger message               */
        /*****************************************************************/
        printf("Trigger message received:\n") ;
        printf("  Triggered queue:    %s\n", QNameStrg);
        printf("  Process Definition: %s\n", ProcessNameStrg);
        printf("  Trigger Data:       %s\n", TriggerDataStrg);
        printf("  Application ID:     %s\n", ApplIdStrg);
        printf("  Environment Data:   %s\n", EnvDataStrg);
        printf("  User Data:          %s\n", UserDataStrg);

        /*****************************************************************/
        /* (P) Perform the operation defined by the Application ID and   */
        /* User Data                                                     */
        /*****************************************************************/
        strcpy(CommandStrg, ApplIdStrg) ;
        strcat(CommandStrg, " ") ;
        strcat(CommandStrg, UserDataStrg) ;

        system(CommandStrg) ;
      }
    } /* End while MQCC == OK */

    /*******************************************************************/
    /* (Q) Disconnect with all work finished                           */
    /*******************************************************************/
    TerminateProg("Processing Completed",
                  &Connection_Handle,
                  &Queue_Handle,
                  Reason) ;

    return(MQRC_NONE) ;
}

/*********************************************************************/
/* (R) Function to convert a character array used by the MQI into    */
/*     a NULL terminated string                                      */
/* NOTE: string_version must be 1 byte longer than array_version     */
/*********************************************************************/
void CvtCArray2Strg(char   *string_version,
                    char   array_version[],
                    MQLONG array_length)
{
    int i ;

    /*****************************************************************/
```

Figure 12.5 (Continued)

```
  /* Copy the whole array into the string version                    */
  /*****************************************************************/
  memcpy(string_version, array_version, array_length) ;

  /*****************************************************************/
  /* Find the start of any trailing blanks                          */
  /*****************************************************************/
  for (i = (array_length-1) ; i >= 0 ; i--)
  {
    switch(string_version[i])
    {
      case '\0':
        /*************************************************/
        /* Its already NULL terminated so we need do no more */
        /*************************************************/
        i = -1 ;
      break;

      case ' ':
        /*************************************************/
        /* Its a blank, convert to NULL                  */
        /*************************************************/
        string_version[i] = '\0' ;
      break;

      default:
        /*************************************************/
        /* Its non-blank. Stamp a NULL in the following  */
        /* position and end                              */
        /*************************************************/
        string_version[i+1] = '\0' ;
        i = -1 ;
      break;

    }
  }
  return ;
}

/*********************************************************************/
/* (S) Function to terminate the program                             */
/*********************************************************************/
void TerminateProg(char     *endmsg,
                   PMQHOBJ  pQueue_Handle,
                   PMQHCONN pConnection_Handle,
                   MQLONG   End_Reason)
{
  MQLONG CompCode ;
  MQLONG Reason ;
  MQLONG Close_Options ;

  /*****************************************************************/
  /* (T) Issue the message associated with termination             */
  /*****************************************************************/
  printf("Trig Mon: %s. Reason was %ld\n",
         endmsg,
         End_Reason) ;

  /*****************************************************************/
  /* (U) Close the queue                                           */
```

Figure 12.5 (Continued)

```
  /*****************************************************************/
  if (*pQueue_Handle != MQHO_UNUSABLE_HOBJ)
  {
    Close_Options = MQCO_NONE ;
    MQCLOSE (*pConnection_Handle,
             pQueue_Handle,
             Close_Options,
             &CompCode,
             &Reason) ;

    printf("Trig Mon: Queue Close Reason Code was %ld\n",
           Reason) ;
  }
  /*****************************************************************/
  /* (V) Disconnect from the local queue manager                   */
  /*****************************************************************/
  if (*pConnection_Handle != MQHC_UNUSABLE_HCONN)
  {
    MQDISC (pConnection_Handle,
            &CompCode,
            &Reason) ;

    printf("Trig Mon: Disconnect Reason Code was %ld\n",
            Reason) ;
  }
  exit(End_Reason) ;
}
```

Figure 12.5 (Continued)

command were typed in from the command line. The trigger monitor creates this command in a variable called CommandStrg. First it copies in the value of the application identifier retrieved from the trigger message. It does this using the strcpy function. For the definitions in Figure 12.4, this field will have the value mqftpr &. The trigger monitor appends a blank character after the application identifier, using the strcat function and then appends the contents of the user data retrieved from the trigger message. For our example, this field is blank. As we pointed out earlier, for simplicity, this trigger monitor ignores any environment data in the trigger message.

Once the command has been issued and, assuming that it is run in the background as with our example, the trigger monitor regains control and reads the next trigger message back at point (M).

The rest of the program is very similar to the receiver program and will not be described again here.

12.3.3 The modified receiver program MQFTPR2

To work with the form of triggering we specified in Fig. 12.4, the receiver program needs to be modified slightly. A full listing is given in Fig. 12.6. Only the modifications need to be described. The first of these is at point (K) where we open the queue specifying the additional option of MQOO_SET. This option allows the receiver program to alter the attributes of the queue, as well as being able to read messages from it. We will use this ability to rearm the triggering of the queue just before the program terminates. This rearming takes place within the TerminateProg function at point (T). This is the other change from the previous version of the program.

In TerminateProg, the object handle for the queue is tested. If it does not have the value MQHO_UNUSABLE_HOBJ, it means that the queue is actually open and so the attribute can be set. The MQSET call allows multiple attributes to be set in one call. In this case, we need to set only one, but even so we need to set up the arrays and counts expected by the call. The variable Selectors is an array of numbers, each of which uniquely identifies an attribute to be updated. The values of the attribute numbers, used by MQSeries, are defined in cmqc.h. In this case, we need to update just one attribute. Its attribute number is MQIA_TRIGGER_CONTROL and it controls whether triggering is currently enabled for the queue. The value we wish to set it to, namely MQTC_ON, goes into the variable IntAttrs at the same array index as the corresponding attribute number goes into Selectors. Since there is only one attribute to be updated, these values go into array index 0. In C, array indices always start from 0. The first value of 1 in the parameter list to MQSET indicates that just a single attribute is to be set. The second 1 in the list is the number of integer attributes to be set. MQSET can

```
/*****************************************************************/
/* Module Name: mqftpr2.c                                        */
/*                                                               */
/* Function:    This program is the receiver half of the file transfer  */
/*              example. It reads messages from a queue, interpreting   */
/*              them as data to be placed into files. The program reads */
/*              File.Transfer.Queue on the local queue manager.         */
/*                                                               */
/*              This is the triggered version                    */
/*                                                               */
/* Usage:       Invoked via the trigger monitor                  */
/*                                                               */
/*   mqftpr                                                      */
/*                                                               */
/* NOTES:                                                        */
/* 1) This code is only an example. For simplicity it performs only     */
/*    very basic error checking.                                 */
/*****************************************************************/

/*****************************************************************/
/* (A) Include definitions for standard C language functions     */
/*****************************************************************/
#include        <stdio.h>
#include        <stdlib.h>
#include        <string.h>

/*****************************************************************/
/* (B) Include definitions for MQSeries                          */
/*****************************************************************/
#include        <cmqc.h>

/*****************************************************************/
/* (C) Define constants used in this program                     */
/*****************************************************************/
#define MAX_FILE_SIZE  4096

/*****************************************************************/
/* (D) Prototypes for local functions                            */
/*****************************************************************/
void TerminateProg(char     *endmsg,
                   PMQHOBJ  pQueue_Handle,
                   PMQHCONN pConnection_Handle,
                   MQLONG   End_Reason) ;

/*************************/
/* (E) Main Entry Point  */
/*************************/
int main(int argc, char**argv)
{
  /*****************************************************************/
  /* (F) Declare Structures used in the MQ API calls. Also, initialize  */
  /* the descriptors and options.                                  */
  /*****************************************************************/
  MQOD          Queue_Descriptor    = {MQOD_DEFAULT} ;
  MQMD          Message_Descriptor  = {MQMD_DEFAULT} ;
  MQGMO         Get_Message_Options = {MQGMO_DEFAULT} ;
  MQHCONN       Connection_Handle   = MQHC_UNUSABLE_HCONN ;
  MQHOBJ        Queue_Handle        = MQHO_UNUSABLE_HOBJ ;
  MQLONG        Data_Length ;
  MQLONG        Open_Options ;
```

Figure 12.6 The modified receiver program for the triggered example.

```
MQLONG        CompCode ;
MQLONG        Reason ;
/*****************************************************************/
/* (G) Declare other variables used in the program               */
/*****************************************************************/
int           ended = 0 ;
FILE          *fp ;
int           nbytes ;
char          to_file[128] ;

struct
{
  char          to_file[128] ;
  unsigned int  Data_Length ;
  MQBYTE        Buffer[MAX_FILE_SIZE] ;
} F_Transfer_Msg ;


/*****************************************************************/
/* (H) Verify command line                                       */
/*     There are no command line arguments                       */
/*****************************************************************/
printf ("Receiver Starting\n") ;

if (argc != 1)
{
  printf("Usage:\n") ;
  printf(" mqftpr2\n") ;
  exit(16) ;
}

/*****************************************************************/
/* (I) Connect to the local queue manager (i.e. default)         */
/*****************************************************************/
MQCONN ("",
        &Connection_Handle,
        &CompCode,
        &Reason) ;
if (CompCode == MQCC_FAILED)
{
  TerminateProg("Could not connect to default queue manager",
                &Queue_Handle,
                &Connection_Handle,
                Reason) ;
  exit(16) ;
}

/*****************************************************************/
/* (J) Construct the descriptor for the file transfer queue      */
/*****************************************************************/
strncpy(Queue_Descriptor.ObjectName,
        "File.Transfer.Queue",
        MQ_Q_NAME_LENGTH) ;

/*****************************************************************/
/* (K) Open the queue on which we receive data                   */
/*****************************************************************/
Open_Options = MQOO_INPUT_EXCLUSIVE |
               MQOO_SET ;
MQOPEN (Connection_Handle,
```

Figure 12.6 (Continued)

```
        &Queue_Descriptor,
        Open_Options,
        &Queue_Handle,
        &CompCode,
        &Reason) ;

if (CompCode == MQCC_FAILED)
{
  TerminateProg("Could not open File.Transfer.Queue",
                &Queue_Handle,
                &Connection_Handle,
                Reason) ;
}

/*****************************************************************/
/* (L) Read messages from the queue                              */
/*****************************************************************/
while (ended == 0)
{
  /***************************************************************/
  /* (M) Retrieve a message from the queue                       */
  /***************************************************************/
  Get_Message_Options.Options = MQGMO_WAIT ;
  Get_Message_Options.WaitInterval = 30000 ;
  memcpy(Message_Descriptor.MsgId, MQMI_NONE, sizeof(MQMI_NONE)) ;
  memcpy(Message_Descriptor.CorrelId, MQCI_NONE, sizeof(MQCI_NONE)) ;

  MQGET (Connection_Handle,
         Queue_Handle,
         &Message_Descriptor,
         &Get_Message_Options,
         sizeof(F_Transfer_Msg),
         &F_Transfer_Msg,
         &Data_Length,
         &CompCode,
         &Reason) ;

  if (CompCode == MQCC_FAILED)
  {
    ended = 1 ;
  }
  else
  {
    /*************************************************************/
    /* (N) Report what is happening                              */
    /*************************************************************/
    printf("Receiver: Transferring file to %s\n",
           F_Transfer_Msg.to_file) ;

    /*************************************************************/
    /* (O) Open the file to be transferred.                      */
    /*************************************************************/
    fp = fopen(F_Transfer_Msg.to_file, "w") ;
    if (fp == NULL)
    {
      printf("Receiver: Could not open output file %s\n",
             F_Transfer_Msg.to_file) ;
      printf("   Data will be discarded\n") ;
    }
    else
```

Figure 12.6 (Continued)

```
        {
          /*******************************************************/
          /* (P) Write out the data.                             */
          /*******************************************************/
          nbytes = fwrite (F_Transfer_Msg.Buffer,
                           1,
                           F_Transfer_Msg.Data_Length,
                           fp) ;
          if (nbytes != F_Transfer_Msg.Data_Length)
          {
            printf("Receiver: Problem writing to file %s\n",
                   F_Transfer_Msg.to_file) ;
          }
          fclose(fp) ;

        }
      }
    }      /* End while(ended = 0) */


    /*********************************************************************/
    /* (Q) Terminate the program                                         */
    /*********************************************************************/
    TerminateProg("Processing Completed",
                  &Queue_Handle,
                  &Connection_Handle,
                  Reason) ;

    return(Reason) ;
}

/*************************************************************************/
/* (R) Function to terminate the program                                 */
/*************************************************************************/
void TerminateProg(char     *endmsg,
                   PMQHOBJ  pQueue_Handle,
                   PMQHCONN pConnection_Handle,
                   MQLONG   End_Reason)
{
    MQLONG CompCode ;
    MQLONG Reason ;
    MQLONG Close_Options ;
    MQLONG Selectors[1] ;
    MQLONG IntAttrs[1] ;

    /*********************************************************************/
    /* (S) Issue the message associated with termination                 */
    /*********************************************************************/
    printf("Receiver: %s. Reason was %ld\n",
           endmsg,
           End_Reason) ;

    /*********************************************************************/
    /* (T) Rearm the trigger for next time                               */
    /*********************************************************************/
    if (*pQueue_Handle != MQHO_UNUSABLE_HOBJ)
    {
      Selectors[0] = MQIA_TRIGGER_CONTROL ;
      IntAttrs[0]  = MQTC_ON ;
      MQSET (*pConnection_Handle,
```

Figure 12.6 (Continued)

```
            *pQueue_Handle,
            1,
            Selectors,
            1,
            IntAttrs,
            0,
            NULL,
            &CompCode,
            &Reason) ;
  }

  /*****************************************************************/
  /* (U) Close the queue                                           */
  /*****************************************************************/
  if (*pQueue_Handle != MQHO_UNUSABLE_HOBJ)
  {
    Close_Options = MQCO_NONE ;
    MQCLOSE (*pConnection_Handle,
             pQueue_Handle,
             Close_Options,
             &CompCode,
             &Reason) ;

    printf("Receiver: Queue Close Reason Code was %ld\n",
           Reason) ;
  }

  /*****************************************************************/
  /* (V) Disconnect from the local queue manager                   */
  /*****************************************************************/
  if (*pConnection_Handle != MQHC_UNUSABLE_HCONN)
  {
    MQDISC (pConnection_Handle,
            &CompCode,
            &Reason) ;

    printf("Receiver: Disconnect Reason Code was %ld\n",
            Reason) ;
  }
  exit(End_Reason) ;
}
```

Figure 12.6 (Continued)

simultaneously update integer and character attributes. The 0 and the NULL parameters following IntAttrs indicate to MQSET that we are not updating any character attributes. The final two parameters are the now familiar completion and reason codes.

This code for rearming triggering works together with the particular kind of triggering we specified in Fig. 12.4 to give the following behavior. When a message arrives on File.Transfer.Queue, the queue manager generates a trigger message and puts it onto INITIATION.QUEUE. At the same time, it disables triggering of File.Transfer.Queue. This allows us to prevent the trigger monitor program from starting multiple copies of the receiver. One copy is started and processes any messages on the queue. If the queue becomes empty, the receiver waits for up to 30 seconds and then terminates itself. Before ending, however, it rearms the trigger, via the MQSET call, so that the trigger monitor can restart it should more work become available.

12.3.4 Arming the trigger

Although we have said that the examples are not complete robust applications, they do illustrate some of the issues facing developers of MQI applications. One of these came to light when testing the programs. We cover it here because it is a generic problem facing anyone using the kind of triggering we specified for the file transfer example. Recall that once the queue has been triggered, the queue manager disables triggering, a feature we use to prevent multiple copies of the receiver from executing. We rely on the

receiver program itself rearming the queue when it terminates. If the receiver program ever terminates without rearming the queue—for example, because of some serious programming or system error—the queue will remain with triggering disabled. It will never be reenabled, since the code which does that is in the receiver program which runs in response to a trigger. One solution to the problem is to have an independent method of rearming the trigger. An MQSC script to do this is shown in Fig. 12.7. The ALTER command can be used to change any queue attribute. Specifying the TRIGGER keyword rearms the trigger for the queue.

12.4 Local Logical Units of Work

Most MQSeries queue managers provide the ability to perform sets of operations on queues as local logical units of work. To use this feature, applications specify MQGMO_SYNCPOINT as one of the options on each MQGET which is to be part of the unit of work. Similarly, they specify MQGMO_SYNCPOINT as one of the options on each MQPUT call. The first MQGET or MQPUT call specifying that the operation is part of a local unit of work causes the unit of work to be started. While the unit of work is in progress, the application can see the effects of its changes, but other applications cannot.

The application completes the unit of work by issuing either an MQCMIT or MQBACK call. The MQCMIT call commits all the changes, which occurred during the unit of work, causing them to be made permanent. The MQBACK call discards all the changes, returning the system to the state it was in before the unit of work started. An MQCMIT or MQBACK call ends the unit of work.

12.5 Participating in Transactions

Many MQSeries queue managers are able to participate in distributed logical units of work. As we saw in Sec. 9.4.2, distributed units of work required a coordinator to control the two-phase commit process. MQSeries queue managers do not provide unit-of-work coordination. Instead, they act as resource managers in the same way as most relational databases do. Support for distributed logical units of work requires that resource managers interact directly with the logical-unit-of-work coordinator. Applications commit or roll back the unit of work by calls to the unit-of-work coordinator rather than to the resource managers themselves. The coordinator then works directly with the resource managers to complete the unit of work.

When working under a unit-of-work coordinator, applications do not issue MQCMIT or MQBACK calls. They do, however, need to specify MQGMO_SYNCPOINT for MQGET operations and MQGMO_SYNCPOINT for MQPUT operations. When the application requests a commit or rollback operation via the unit-of-work coordinator, the appropriate action will be taken by the queue manager in response to direct calls from the unit-of-work coordinator.

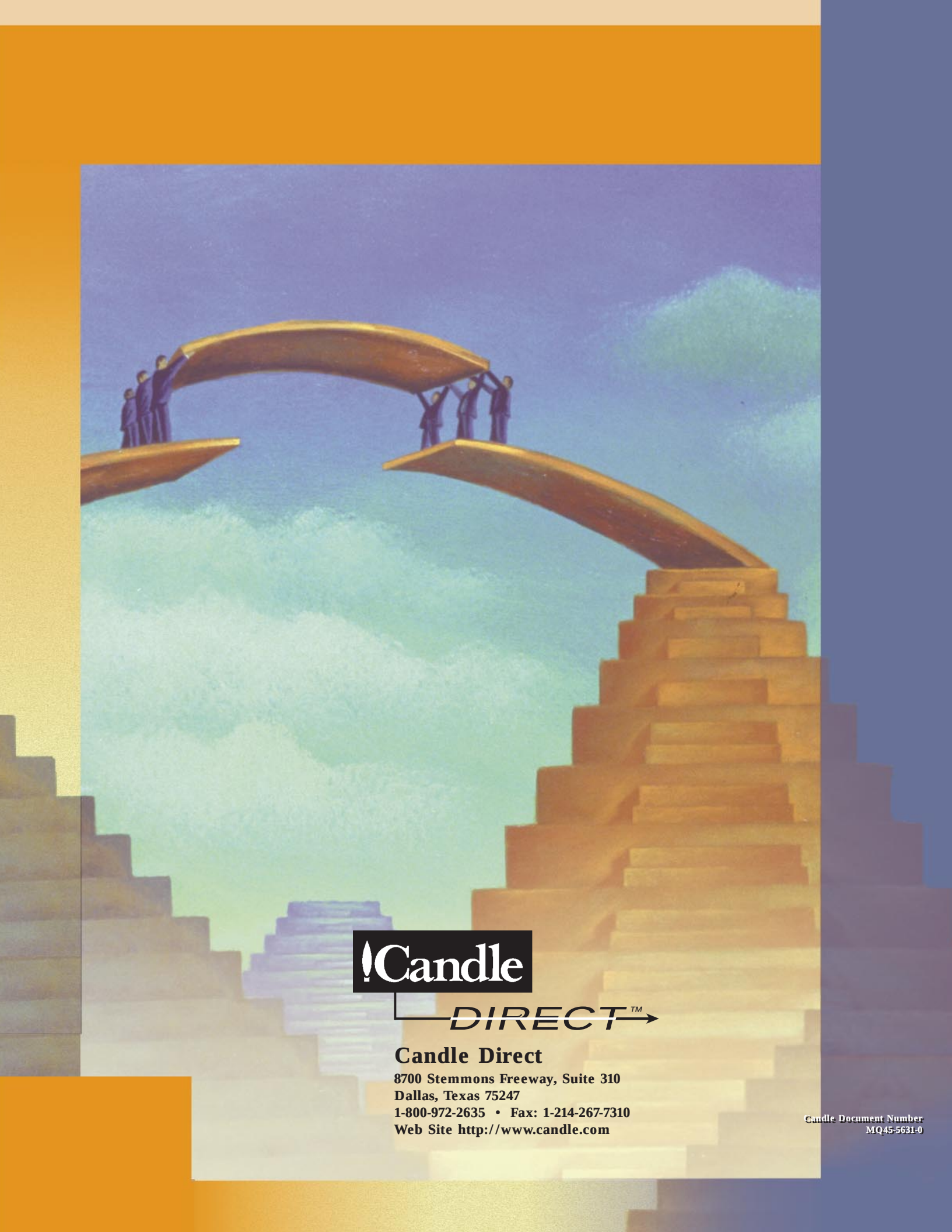
Although the applications have very little to do to support distributed units of work, there is some administration involved in defining the relationship between the queue manager and the unit-of-work coordinator. The exact details of this vary from system to system, but the work involved is, in general, similar to that for establishing the relationship between a database manager and a unit-of-work coordinator.

```
*****************************************************************
*                                                               *
* Script Name: mqftpsc3.mqs                                     *
*                                                               *
* Description: Arm the triggering of the file transfer queue    *
*                                                               *
*****************************************************************
*****************************************************************
* (A) FILE.TRANSFER.QUEUE                                       *
*                                                               *
*****************************************************************
  ALTER QLOCAL('File.Transfer.Queue')                          +
* Enable triggering for this queue
        TRIGGER
```

Figure 12.7 MQSC script to enable triggering for the file transfer example.

12.6 Summary

In this chapter, we looked at the MQI, the application programming interface to MQSeries products. We discussed the basic verbs provided by the interface and described some of the options available when using them. We looked at an example application which illustrated the use of the basic MQI calls, and saw how it could be extended to be triggered. Finally, we examined some of the other, more advanced MQI calls which support transaction-oriented applications.



!Candle DIRECT™

Candle Direct
8700 Stemmons Freeway, Suite 310
Dallas, Texas 75247
1-800-972-2635 • Fax: 1-214-267-7310
Web Site http://www.candle.com

Candle Document Number
MQ45-5631-0