Unit 5. Introduction to the Message Queue Interface (MQI)

What this unit is about

This unit covers a brief introduction to the Message Queue Interface calls used by application programs.

What you should be able to do

After completing this unit, you should be able to:

- Recognize MQI calls in a program
- Describe the purpose of each MQI parameter
- Explain the purpose of each field in the message descriptor

How you will check your progress

Accountability:

- Checkpoint
- Machine exercises

Unit objectives

After completing this unit, you should be able to:
- Recognize MQI calls in a program
- Describe the purpose of each MQI parameter
- Explain the purpose of each field in the message descriptor

Figure 5-1. Unit objectives WM203 / VM2032.0

Notes:

Notation

- WebSphere MQ application programming reference

```
MQPUT1 (Hconn, ObjDesc, MsgDesc, PutMsgOpts, BufferLength,
        Buffer, CompCode,  Reason)
```

- Equivalent in C

```
MQPUT1 (Hconn, &ObjDesc, &MsgDesc, &PutMsgOpts,
        BufferLength, Buffer, &CompCode, &Reason);
```

- Equivalent in COBOL

```
CALL "MQPUT1" USING HCONN, OBJDESC, MSGDESC, PUTMSGOPTS,
        BUFFERLENGTH, BUFFER, COMPCODE, REASON.
```


Notes:

Parameters common to all MQI calls

- *Hconn*, connection handle
  - Returned by the MQCONN and MQCONNX calls
  - Input parameter on all other MQI calls
- *CompCode*, completion code
  Examples: MQCC_OK
  MQCC_WARNING
  MQCC_FAILED
- *Reason*, reason code
  Examples: MQRC_NONE
  MQRC_. . .

Figure 5-3. Parameters common to all MQI calls WM203 / VM2032.0

Notes:

A listing of all the reason codes is contained in SC34-6596 *WebSphere MQ Application Programming Reference*.

Object descriptor

- *ObjectType*

MQOT_Q	Queue
MQOT_PROCESS	Process
MQOT_Q_MGR	Queue manager

- *ObjectName*
  - Must be blank if *ObjectType* is MQOT_Q_MGR
  - Every object defined to a queue manager has a unique name within its type
- *ObjectQMgrName*
  - Blank denotes the local queue manager
- *DynamicQName* specifies the name of the dynamic queue to be created
- *AlternateUserId* is alternative user identifier for checking the authorization to open the object

Figure 5-4. Object descriptor

Notes:

The object descriptor is used by an application to identify a WebSphere MQ object that it wants to open. It is one of the parameters on the MQOPEN call and MQPUT1 call, where it is referred to as *ObjDesc*.

Three fields in the object descriptor are needed to identify an object uniquely:

- The object type, *ObjType*, because the name of an object is only unique within its type
- The name of the object, *ObjectName*
- The name of the queue manager which owns the object, *ObjectQMgrName*

Connecting and disconnecting

- Connect queue manager

```
MQCONN (QMgrName, Hconn, CompCode, Reason)
```

```
MQCONNX (QMgrName, ConnectOpts, Hconn, CompCode, Reason)
```

- Disconnect queue manager

```
MQDISC (Hconn, CompCode, Reason)
```


Figure 5-5. Connecting and disconnecting WM203 / VM2032.0

Notes:

An application must first connect to a queue manager before it can access any of the queue manager's resources. It issues an MQCONN or MQCONNX call in order to do this.

In addition to CompCode and Reason, the parameters of the MQCONN call are as follows:

- QMgrName - The name of the queue manager to which the application wants to connect. If the parameter consists entirely of blanks, the application connects to the default queue manager.
- Hconn - The connection handle that is returned to the application. This handle represents the connection to the queue manager and it must be specified on all subsequent MQI calls to the queue manager.

Opening and closing an object

- Open object

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

  - The object handle, *Hobj*, identifies the object in later calls
  - Open options can be added together

    Example in C:

```
Options = MQOO_INPUT_SHARED + MQOO_FAIL_IF_QUIESCING;
```

- Close object

```
MQCLOSE (Hconn, Hobj, Options, CompCode, Reason)
```


Notes:

The Options parameter on the MQOPEN call is used by the application to specify which operations to perform on the object being opened, for example, MQOO_INPUT_SHARED, or to control the action of MQOPEN, for example, MQOO_FAIL_IF_QUIESCING.

An application normally needs authority to open an object for each of the requested operations. These authorities are checked at the time the object is opened.

For getting messages from a queue, an application may request either MQOO_INPUT_SHARED or MQOO_INPUT_EXCLUSIVE. Alternatively, it might request MQOO_INPUT_AS_Q_DEF. In the latter case, the queue is opened in the manner specified by the DefInputOpenOption attribute of the local or model queue being opened. The value of this attribute may be MQOO_INPUT_SHARED or MQOO_INPUT_EXCLUSIVE. The MQOO_INPUT_ options are for removing messages from the queue. There is a separate option, MQOO_BROWSE, for reading messages on a queue and leaving them there.

Another attribute of a local or model queue is *shareability*. The value of this attribute may be MQQA_SHAREABLE or MQQA_NOT_SHAREABLE. The latter value takes precedence over any input options specified on the MQOPEN call.

Dynamic queue

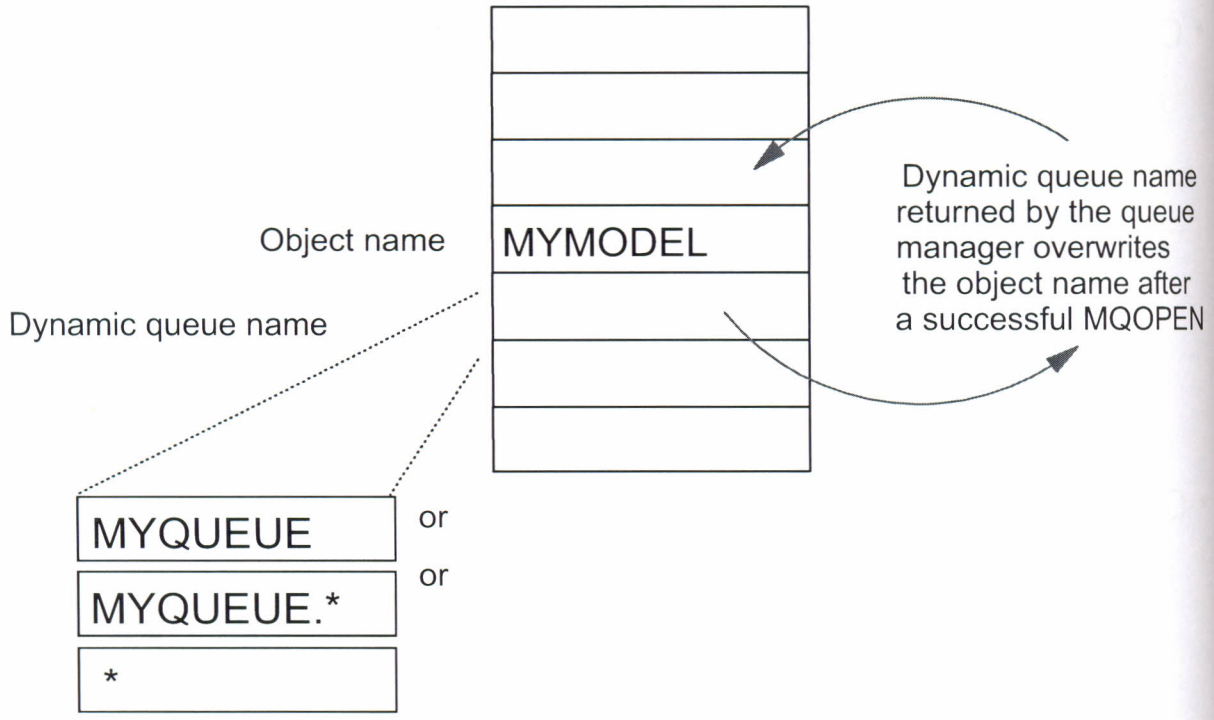


Figure 5-7. Dynamic queue

WM203 / VM2032.0

Notes:

A dynamic queue is created when a model queue is opened. The following fields in the object descriptor are set before the MQOPEN call:

- *ObjectName* identifies the model queue
- *DynamicQName* specifies the name of the dynamic queue to be created

After the MQOPEN call, *ObjectName* contains the name of the dynamic queue.

A temporary dynamic queue:

- Is deleted when the queue is closed
- Does not survive a queue manager restart
- Can store nonpersistent messages only

A permanent dynamic queue:

- Can be deleted by an option on the MQCLOSE call or by using the WebSphere MQ command `DELETE QLOCAL`.
- Survives a queue manager restart
- Can store persistent messages

Put message

```
MQPUT (Hconn, Hobj, MsgDesc, PutMsgOpts, BufferLength,
       Buffer, CompCode, Reason)
```

- *MsgDesc*
  - Message descriptor
  - Input fields set by the application
  - Output fields set by the queue manager
- *PutMsgOpts*
  - Message options

  Examples:

  MQPMO_SYNCPOINT
  MQPMO_FAIL_IF_QUIESCING
- *BufferLength*
  - Length of the message in buffer
- *Buffer*
  - Message data

Figure 5-8. Put message

Notes:

The MQPUT call puts a message on a queue. In addition to *CompCode* and *Reason*, the parameters of the MQPUT call are as follows.

Hconn - The connection handle.

Hobj - The object handle returned from a successful MQOPEN call with the option MQOO_OUTPUT specified.

MsgDesc - The message descriptor structure, MQMD. Some fields are input fields to the MQPUT call, some are output, and some are both input and output. The message descriptor accompanying a message on a queue has some fields supplied by the application and some supplied by the queue manager.

PutMsgOpts - The put message options structure, MQPMO. One of the fields in this structure, Options, is used by the application to specify options that control the action of MQPUT.

Priority

- *Priority*
  Field in message descriptor, one is set by application to one of these:
  - Value in the range 0 (lowest) to 9 (highest)
  - MQPRI_PRIORITY_AS_Q_DEF
- *MsgDeliverySequence*
  Attribute of a local or model queue with values
  - MQMDS_PRIORITY — Messages enqueued with respect to the priority field in the MQ message descriptor
  - MQMDS_FIFO — Messages enqueued at the default priority of the queue
- *DefPriority*
  - Attribute of queue specifying the default message priority
  - Used when an application sets *Priority* to MQPRI_PRIORITY_AS_Q_DEF
- Conventions
  - In general, use the default priority
  - Use the same priority as the original message for a reply or a report

Figure 5-9. Priority

Notes:

When the message is put on a queue, the application might set the *Priority* field in the message descriptor to a value in the range 0 (lowest priority) to 9 (highest priority). This value determines the priority of the message which in turn might determine when the message is retrieved from the queue by an MQGET call.

The way message priority is used depends upon the value of the *MsgDeliverySequence* attribute of the queue.

Get message

```
MQGET (Hconn, Hobj, MsgDesc, GetMsgOpts, BufferLength, Buffer,
                    DataLength, CompCode, Reason)
```

- *GetMsgOpts*
  - Message options

    Examples:
    MQGMO_ACCEPT_TRUNCATED_MSG
    MQGMO_BROWSE
    MQGMO_CONVERT
    MQGMO_FAIL_IF_QUIESCING
    MQGMO_SYNCPOINT
    MQGMO_WAIT

- *WaitInterval*
  - Maximum time the application waits
  - Application can specify an unlimited wait.

- *BufferLength*
  - Length of the buffer area

- *Buffer*
  - Area to contain the message data

- *DataLength*
  - Length of the message returned

Figure 5-10. Get message

Notes:

The MQGET call gets a message from a queue. In addition to *CompCode* and *Reason*, the parameters of the MQGET call are as follows:

- Hconn - The connection handle.
- Hobj - The object handle returned from a successful MQOPEN call with either of the options MQOO_INPUT_ or MQOO_BROWSE specified.
- MsgDesc - The message descriptor structure, MQMD. Some fields are input fields to the MQGET call, some are output, and some are both input and output. Essentially, the message descriptor structure returned to the application by an MQGET call is the same as that which accompanied the message on the queue.
- GetMsgOpts - The get message options structure, MQGMO. The MQGMO Options field, is used by the application to specify options that control the action of MQGET. The option MQGMO_WAIT indicates that the application is to wait until a suitable message arrives if one is not already on the queue.

Reply-to queue

- Queue to which reply and report messages should be sent
- Message descriptor fields set by application before `MQPUT` call:
  - *`ReplyToQ`* can be the name of a dynamic queue or a reply-to queue alias
  - *`ReplyToQMgr`* can be set to blank

Figure 5-11. Reply-to queue WM203 / VM2032.0

Notes:

When an application puts a request message on a queue, it might set two fields in the message descriptor to indicate where the reply message, or any requested report messages, should be sent.

More fields in the message descriptor

- *MsgType*
  - MQMT_DATAGRAM
  - MQMT_REQUEST
  - MQMT_REPLY
  - MQMT_REPORT
- *Report*
  - Report options

  Examples:
  - MQRO_EXCEPTION
  - MQRO_EXPIRATION
  - MQRO_COA
  - MQRO_COD
  - MQRO_DISCARD
  - MQRO_PAN
  - MQRO_NAN
- *Expiry*
  - Expiry time
- *Feedback*
  - Used in a report message
  - Feedback or reason code

  Examples:
  - MQFB_EXPIRATION
  - MQFB_COA
  - MQFB_COD
  - MQRC_Q_FULL
  - MQRC_NOT_AUTHORIZED
  - MQFB_QUIT
  - MQFB_PAN
  - MQFB_NAN

Figure 5-12. More fields in the message descriptor

Notes:

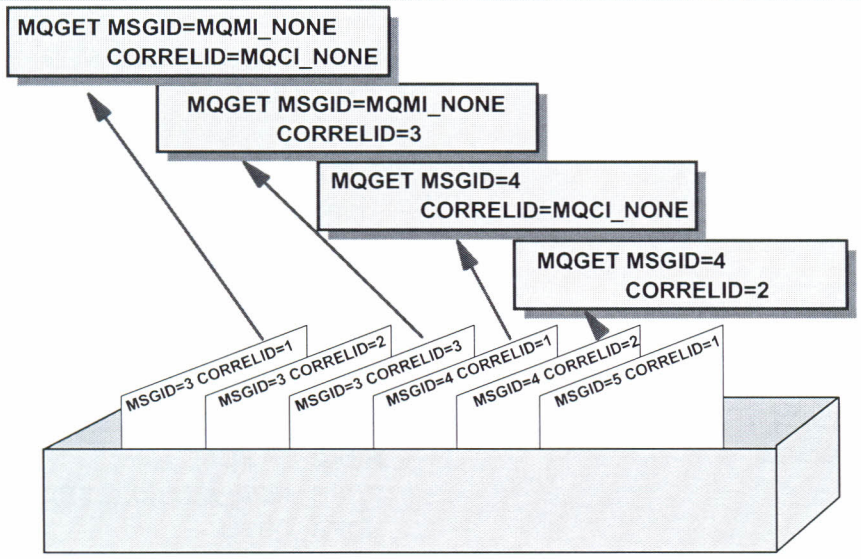
MsgType - Four message types are currently defined by WebSphere MQ but other types may be defined in the future. You can, however, define your own types.

Expiry is a time set by the application. The message becomes eligible to be discarded if it has not been removed from the destination queue before this period of time has elapsed.

Report - This field is used by the application to specify which types of report messages are required.

Feedback - This field is only used in a report message. It indicates the nature of the report or, for an exception report, the reason for the report. Some values of this field are defined by WebSphere MQ but you can define your own values as well. In exception reports, this field may also contain certain reason codes. One special value, MQFB_QUIT, indicates that the application should end.

Message and correlation identifiers



- Two fields in the message descriptor
  - *MsgId*, message identifier
    - If an application specifies MQMI_NONE on an MQPUT call, the queue manager generates a unique message identifier
  - *CorrelId*, correlation identifier
    - In a reply or report message, it is normally copied from the *MsgId* of the original message
- Both fields are treated as bit strings by the queue manager and are not subject to any data conversion

Figure 5-13. Message and correlation identifiers WM203 / VM2032.0

Notes:

On an MQPUT call, an application might either specify a particular value for the message identifier or it may specify the value MQMI_NONE. In the latter case, the queue manager generates a unique message identifier and places it the message descriptor which accompanies the message. The queue manager also returns the generated message identifier to the application as an output field in the message descriptor. It is important therefore, that the application resets the MsgId field to MQMI_NONE before each MQPUT call if it wants the queue manager to generate a unique message identifier.

An application can use the put message option MQPMO_NEW_MSG_ID to request the generation of a unique message identifier. The use of this option relieves the application of the need to reset the MsgId field to MQMI_NONE before each MQPUT call.

The correlation identifier is normally used to provide an application with a means of matching a reply or report message with the original message. In a reply or report message therefore, the value of the CorrelId field is normally copied from the MsgId field of the original message. There is a report option to vary this rule, however.

Retrieving messages

- With selection criteria
  - Set *MsgId* and *CorrelId* before an MQGET call
  - Ensure that *MatchOptions* in get message options is set to MQMO_MATCH_MSG_ID + MQMO_MATCH_CORREL_ID
- Without selection criteria
  - Reset *MsgId* to MQMI_NONE and *CorrelId* to MQCI_NONE before each MQGET call

  or
  - Set *MatchOptions* in get message options to MQMO_NONE (default)
- On return from an MQGET call, *MsgId* and *CorrelId* in MQ message descriptor are set to the values for the message retrieved

Figure 5-14. Retrieving messages

Notes:

Using the MQGET call, an application may retrieve messages from a queue in the order determined by the MsgDeliverySequence attribute of the queue. Alternatively, an application might retrieve only selected messages from a queue according to the values in their MsgId or CorrelId fields.

To retrieve a message with a specific message identifier or correlation identifier, the application sets the MsgId and CorrelId fields in the message descriptor before an MQGET call.

Order of retrieving messages

- Messages on a queue can be retrieved by an application in the same order they were put by another application, provided:
  - The messages all have the same priority
  - The messages are all put within the same unit of work, or all put outside of a unit of work
  - No other application is getting messages from the queue
  - The queue is local to the putting application
  - Messages might be interspersed with messages put by other applications

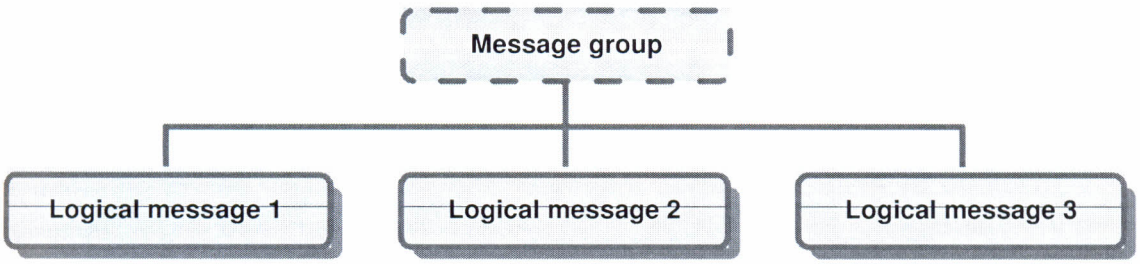
- If the queue is not local to the putting application, the order of retrieval is still preserved provided:
  - The first three conditions above still apply
  - Only a single path is configured for the messages
  - No message is put on a dead letter queue
  - No nonpersistent messages are transmitted over a fast message channel

Figure 5-15. Order of retrieving messages WM203 / VM2032.0

Notes:

Message Groups are intended as a solution to this problem.

Message group



- Message group consists of one or more logical messages
  - Allows an application to group together related messages
  - Ensures ordering on retrieval (where it is not already guaranteed)
- Logical message is:
  - A physical message (unless it is split into segments)
  - Identified by the `GroupId` and `MsgSeqNumber` fields in the message descriptor

Figure 5-16. Message group

Notes:

A *message group* is a group of one or more logical messages.

A *logical message* is a physical message, unless it is split into segments. For this illustration, it assumed that a logical message is a physical message.

A logical message within a group is identified by two fields in the message descriptor, `GroupId` and `MsgSeqNumber`. All logical messages belonging to the same group have the same value for the `GroupId` field. The `MsgSeqNumber` field has the value 1 for the first logical message, 2 for the second, and so on. There is, therefore, an implied ordering of the logical messages within a group.

A *physical message* which does not belong to any group has binary zeros in the `GroupId` field, and the value 1 in the `MsgSeqNumber` field.

There are two main uses of a message group.

- To ensure ordering on retrieval in circumstances where order is not already guaranteed.

  An application is able to put a sequence of messages constituting a message group on a queue by specifying the put message option `MQPMO_LOGICAL_ORDER`. The queue

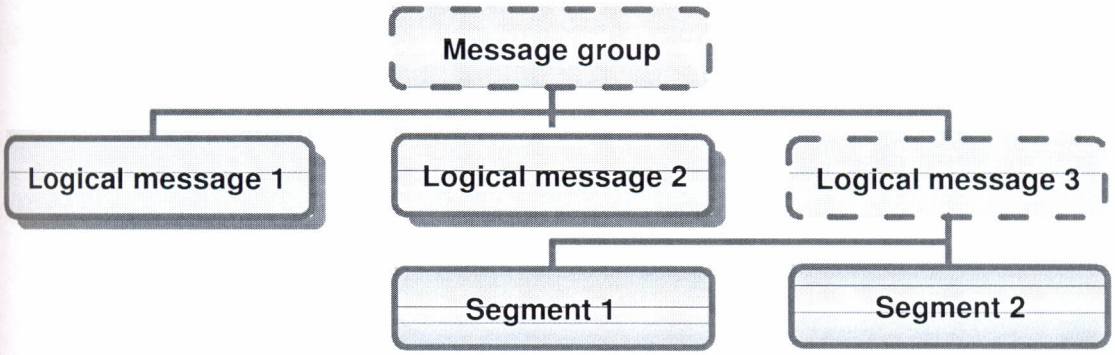
manager generates a unique group identifier and assigns a message sequence number to each message as it is put on the queue.

Another application is then able to get the messages constituting the group from the queue, in the same order they were put, by specifying the get message option `MQGMO_LOGICAL_ORDER`.

- To allow an application to group together related messages.

  This option can be useful, for example, if it is important for a group of related messages to be processed by the same server instance, or by a particular server instance. By setting the value `MQMO_MATCH_GROUP_ID` in the `MatchOptions` field in get message options, an application can retrieve only those messages with a specified group Identifier.

Message segmentation



- A segment is:
  - A physical message
  - Identified by the `GroupId`, `MsgSeqNumber`, and `Offset` fields in the message descriptor
- Segmentation is needed when a message is too large for an application, a queue, or queue manager
- A message can be segmented and reassembled:
  - By the queue manager
  - By an application

Figure 5-17. Message segmentation WM203 / VM2032.0

Notes:

A logical message may consist of two or more physical messages, each of which is called a segment. All segments of the same logical message have the same values for the `GroupId` and `MsgSeqNumber` fields, but the value of the `Offset` field is different for each segment. The segment offset is the offset of the data in the physical message from the start of the logical message. The offset of the first segment is 0.

Message segmentation is needed when a message is too large for an application, a queue, or a queue manager. Thus, a logical message is essentially the unit of information that needs to be handled by an application, but its size may mean that it has to be split into more than one physical message.

Provided a message is not too large for an application to handle in a single buffer, segmentation and reassembly of a message can be performed by the queue manager. This scenario is the simplest.

To ask the queue manager to segment a message if necessary, the putting application sets the value `MQMF_SEGMENTATION_ALLOWED` in the `MsgFlags` field of the message descriptor and issues one `MQPUT` call. Similarly, the getting application specifies the get message

option MQGMO_COMPLETE_MSG in order to request the queue manager only to return a complete logical message on an MQGET call. And if the logical message is segmented, the queue manager reassembles it before returning it to the application.

If a message is too large for an application to handle in a single buffer, the application may perform the segmentation itself by issuing an MQPUT call for each segment. Similarly, an application may issue an MQGET call for each segment of a logical message.

Distribution list

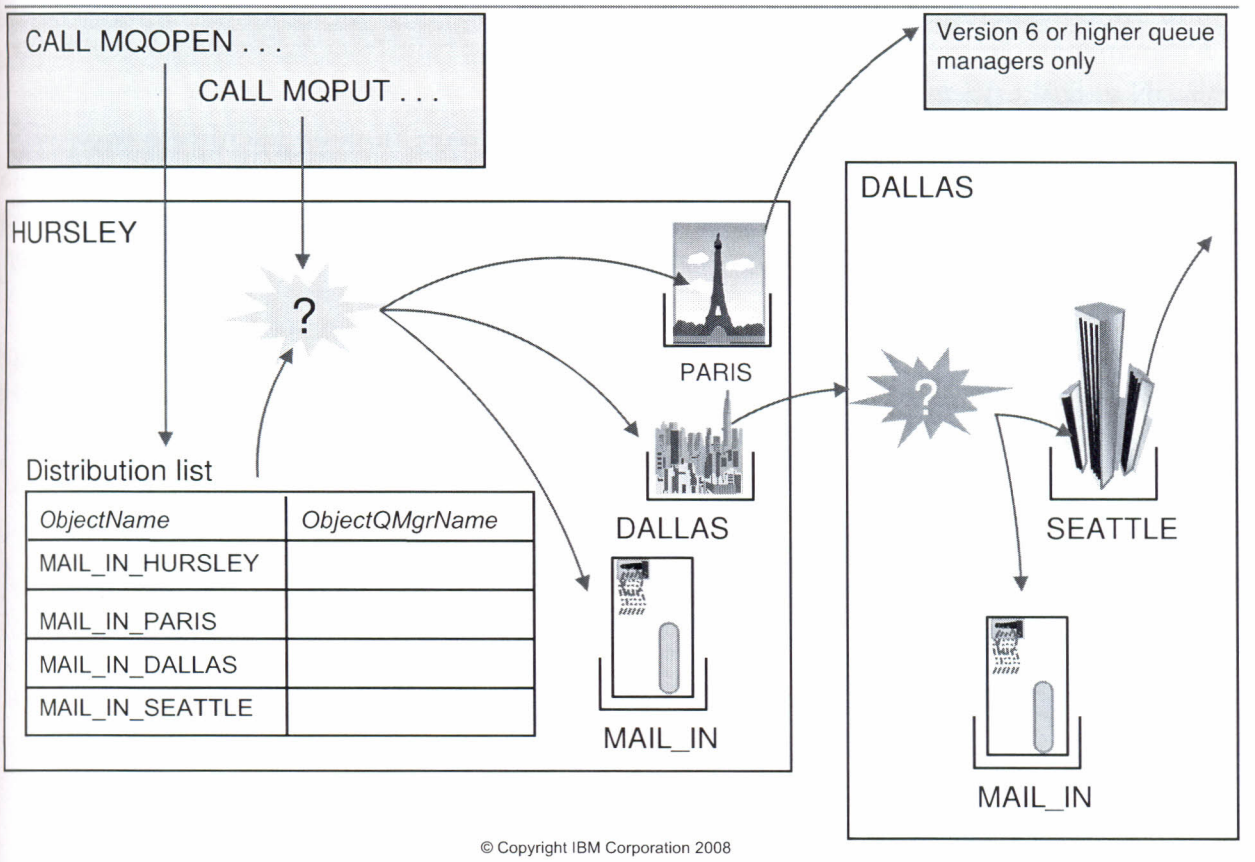


Figure 5-18. Distribution list WM203 / VM2032.0

Notes:

A distribution list might contain the name of an alias queue or the name of a local definition of a remote queue. In the example of a distribution list shown on the visual, the following are assumed.

- MAIL_IN_HURSLEY is an alias queue which resolves to the local queue MAIL_IN on queue manager HURSLEY.
- MAIL_IN_PARIS is a remote queue definition which resolves to the queue MAIL_IN on queue manager PARIS using transmission queue PARIS.
- MAIL_IN_DALLAS is a remote queue definition which resolves to the queue MAIL_IN on queue manager DALLAS using transmission queue DALLAS.
- MAIL_IN_SEATTLE is a remote queue definition which resolves to the queue MAIL_IN on queue manager SEATTLE using transmission queue DALLAS.

As a result, ObjectQMgrName is to blank for each object record.

Once a distribution list has been opened, the application can call MQPUT to put a message on the queues in the list. As a response to the call, the queue manager puts **one** copy of

the message on each local queue, including the transmission queues. Thus, only one copy of the message is put on the transmission queue DALLAS even though the message is ultimately destined for two queues.

On the transmission queue, the application data is prefixed by a distribution header, as well as a transmission queue header. Appended to the distribution header are object records for MAIL_IN at DALLAS and MAIL_IN at SEATTLE. Thus, the message that is transmitted between HURSLEY and DALLAS effectively contains a distribution list for the two destinations. Therefore, when the message is received by the receiving MCA at DALLAS, the MCA has sufficient information to open a distribution list and put the message to it. The queue manager puts one copy of the message on the local queue MAIL_IN and another copy on the transmission queue SEATTLE. And so on.

You can see therefore that an important property of the implementation is that a message destined for multiple queues is only replicated at the last possible point at each stage of its journey. In this way, network traffic is minimized.

Message properties

- Arbitrary values associated with the message but not part of the body
  - Like a user-extendable MQMD
  - Already part of JMS
- New verbs MQSETMP and MQINQMP
  - Properties can be integers, strings, Booleans, and so on
- Easier to use than RFH2 folders
  - Receiving applications do not see them unless they want
  - No need to parse and skip over message headers
- Appear as RFH2 properties on older queue managers
- Permits explicit statement of relationships between messages

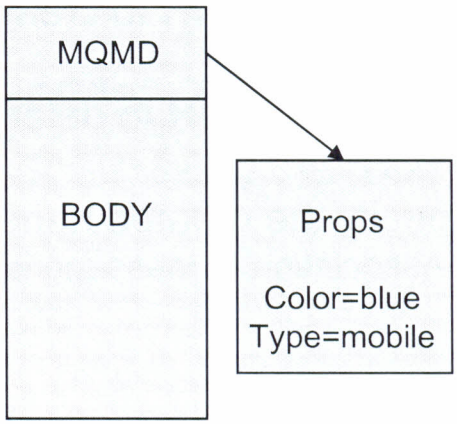


Figure 5-19. Message properties WM203 / VM2032.0

Notes:

Message properties are used by message selectors to filter publications to topics or to selectively get messages from queues. Message properties can be used to include business data or state information without having to store it in the message data.

Another benefit of message properties is that applications do not have to access data in the WebSphere MQ Message Descriptor (MQMD) or RFH headers because fields in these data structures can be accessed as message properties using the Message Queue Interface functions.

Message handles

- Used to reference the message properties of a message
- Use **MQCRTMH** to create a message handle
- Use **MQDLTMH** to delete a message handle
- Specify the created handle on **MQPUT**, **MQPUT1** or **MQGET**
  - Association to message is by PutMsgOps or GetMsgOpts

Figure 5-20. Message handles

Notes:

The message handle can be used on the MQPUT and MQPUT1 calls to associate the properties of the message handle with those of the message being put. Similarly, by specifying a message handle on the MQGET call, the properties of the message being retrieved can be accessed using the message handle when the MQGET call completes.

There are new Message Queue Interface (MQI) functions to create and delete message handles (MQCRTMH and MQDTLMH). The message handle is used on the MQPUT and MQPUT1 calls to associate message properties to the message being put. Similarly a message handle in the MQGET function is the reference to all the message properties when the MQGET is complete.

The scope of message handles is between the MQCRTMH function call and the MQDLTMH function call or the unit of processing that defines the handle.

Using message properties – Example Part 1

This is a router application: get a message and work with it

```
// 1. Initial setup and read input

MQCRTMH(hConn, &CrtMsgHOpts, &hRequestMsg, &CC, &RC);
GetMsgOpts.MsgHandle = hRequestMsg;
MQGET(hConn, hObj, &MD, &GetMsgOpts, BufferLen, Buffer,
  &DataLen, &CC, &RC);

// 2. Forward request unchanged to server application
//    named in message

PutMsgOpts.Action = MQPACT_FORWARD;
PutMsgOpts.OriginalHandle = hRequestMsg;
MQPUT(hConn, hServerObj, &MD, &PutMsgOpts, DataLen,
  &Buffer, &CC, &RC);
```


Figure 5-21. Using message properties – Example Part 1 WM203 / VM2032.0

Notes:

An outline of a C program that routes messages and uses message properties. There are two steps:

1. A message handle is created. This handle is tied to the already open queue manager connection in hConn. The handle is returned in the hRequestMsg variable. A new get message options (MQGMO) structure is then connected to the message handle. A message is then retrieved from a source queue.
2. A put message options (MQPMO) structure is set to indicate the message are forwarded. The OriginalHandle field is set to the message handle created in step 1. The message is then put to another queue to be serviced by another application.

Using message properties – Example Part 2

This is a router application: get a message and work with it

```
// 3. Tell that the requester message has been dealt with
//    by updating the existing property

Name.VSPtr = "RequestStatus";
Name.VSLength = MQVS_NULL_TERMINATED;

MQSETMP(hConn, hRequestMsg, &SetPropOpts, &Name,
  &PropDesc, MQTYPE_STRING, "REQUEST_RECEIVED", 16, &CC,
  &RC);

PutMsgOpts.Action = MQPACT_REPLY;
PutMsgOpts.OriginalMsgHandle = hRequestMsg;

MQPUT(hConn, hReplyObj, &MD, &PutMsgOpts, DataLen,
  &Buffer, &CC, &RC);
```


Figure 5-22. Using message properties – Example Part 2 WM203 / VM2032.0

Notes:

An outline of a C program that routes messages and uses message properties – continued. The final step is:

3. WebSphere MQ V7 has a new type of variable known as variable length strings. Here **Name** is of type MQCHARV and is set to have the value "RequestStatus".

MQSETMP is then called to set the property RequestStatus = "REQUEST_RECEIVED".

The PMO Action is set to MQPACT_REPLY indicating the message is a reply and the message handle is connected to the PMO.

The message is then put to the reply-to queue.

Asynchronous message reception - CallBack

- Allows identification of a function to be invoked when a message arrives on a queue
  - Not the same as triggering or MQGET with wait
  - MQGET is not used at all
- MQCB call to register and deregister; suspend and resume consumers
- MQCTL call to start and stop; suspend and resume

Figure 5-23. Asynchronous message reception - CallBack WM203 / VM2032.0

Notes:

Asynchronous message reception or CallBack is a new MQI that enable consuming messages from multiple queues or topics. This facilitates the implementation of a message driven processing model (other than using message triggering or message dispatchers based on MQGET with wait) or to have programs that are unaware of the message size that is received.

This is a new programming style that can simplify the design and implementation of new applications. Message processing functions are registered with the MQCB MQI call. The function is then invoked by the queue manager when there are messages available that match the selection criteria. The control function (MQCTL) indicates to the queue manager when to start dispatching the callback functions to pass messages to them. The callback functions receive and process messages in asynchronous mode.

Selectors

- Use a SQL92 clause to select messages by properties
  - Includes MQMD fields
  - Does not include message body
- Already exists for JMS applications
- Filter specified on MQOPEN and MQSUB calls
  - On MQOD for MQOPEN
  - On MQSD for MQSUB
- Example:

```
"JMSType = 'car' AND color = 'blue' AND weight > 2500"
```


Figure 5-24. Selectors WM203 / VM2032.0

Notes:

A message selector is a criteria to filter messages during MQGET or MQCB function calls. The selection criteria is specified during the MQOPEN or MQSUB function calls. To change the selection criteria the object or subscription has to be closed and opened again.

Message selectors existed in Java Message Service (JMS) before WebSphere MQ V7.0. Now any application program that uses Message Queue Interface functions calls can use message selectors.

Message selectors act on message properties defined in a message. Fields in the MQMD data structure can be considered as message properties in message selectors.

A message selector is a variable length string (MQCHARV) field in MQOD or MQSD data structures. The syntax of the message selector string is based on a subset of the SQL92 conditional expressions.

When message selectors are used with point to point messaging applications the selection string is specified in the field SelectionString in the MQOD that is a parameter of the MQOPEN function call. Any subsequent MQGET function calls for the object handle

returned in the MQOPEN filters the messages based on the selection criteria described in the selection string.

When message selectors are used with publish/subscribe applications the selection string is specified in the field SelectionString in the MQSD that is a parameter of the MQSUB function call. Any subsequent MQGET function calls for the object handle returned in the MQSUB filters the publications based on the selection criteria described in the selection string.

A key requirement for message selection to work is that message producer applications (point to point or publishers) set the message properties required by the message selectors.

Checkpoint questions

1. True or false - A temporary dynamic queue can hold persistent messages.

2. True or false: The correlation identifier is normally used to provide an application with a means of matching a reply or report message with the original message.

3. An application can retrieve a message from a queue based on:
   a. Message ID
   b. Correlation ID
   c. Next available message based on the MsgDeliverySequence value of the queue
   d. An SQL-92 like selector
   e. All of the above

Figure 5-25. Checkpoint questions WM203 / VM2032.0

Notes:

Unit summary

Having completed this unit, you should be able to:
- Recognize MQI calls in a program
- Describe the purpose of each MQI parameter
- Explain the purpose of each field in the message descriptor

Figure 5-26. Unit summary

Notes: