Legal Information

Microsoft Message Queue Server Programmer's Reference

Finding What You Need

The Microsoft® Message Queue Server (MSMQ) Programmer's Reference is a set of online documents that accompany Microsoft Message Queue Server (MSMQ). This software development kit is available through the Microsoft® Platform SDK, as well as on the disks shipped with MSMQ.

For a description of the content of different sections, see:

- Using MSMQ
- MSMQ Guide
- MSMQ Reference

What's in "Using MSMQ"

"Using MSMQ" contains a work-flow listing of code examples needed to perform the basic functions provided by MSMQ. Use this section to find examples and information about how to complete specific tasks.

Note For information on a specific MSMQ function, property, structure, and ActiveX component, refer to the "MSMQ Reference" section.

The following topics are in "Using MSMQ:"

Using MSMQ API Functions

- Creating a Queue
- Locating a Public Queue
- Opening a Queue
- Closing a Queue
- Deleting a Queue
- Sending Messages To a Queue
- Sending Messages That Request Acknowledgments
- Sending Messages That Request a Response
- Sending Private Messages
- Sending Messages Using an Internal Transaction
- Sending Messages Using an MS DTC External Transaction
- Reading Messages Synchronously
- Reading Messages Asynchronously
- Reading Messages Using a Cursor
- Reading Messages in a Dead Letter Queue
- Reading Messages in a Machine Journal
- Reading Messages in a Queue Journal
- Returning an Acknowledgment Message by a Connector Application
- Retrieving a Queue's Properties Using API Functions
- Setting a Queue's Properties Using API Functions
- Authenticating Messages Using API Functions
- Setting Access Control Security for a Queue
- Using Transactions

Using ActiveX Components

- Creating a Queue
- Locating a Public Queue
- Opening a Queue
- Sending Messages To a Queue
- Sending Messages that Request Acknowledgments
- Sending Messages that Request a Response
- Sending Private Messages
- Reading Messages In a Queue
- Retrieving a Queue's Properties Using ActiveX Components

- Setting a Queue's Properties Using ActiveX Components

What's in "MSMQ Guide"

Use this section to get conceptual information about the different queues that can be used, the different messages that are available, how queues and messages are defined by properties (see object properties), in addition to information on other MSMQ services.

The following topics are in the "MSMQ Guide:"

What's in "MSMQ Reference"

This reference describes the functions, properties, structures, error and warning codes, and ActiveX components provided by the Microsoft Message Queue Server (MSMQ) SDK.

Note For code examples of basic MSMQ functions such as creating a queue, sending message, receiving messages, and so on, refer to "Using MSMQ." For background information on MSMQ concepts, refer to "MSMQ Guide."

The following topics are in the "MSMQ Reference:"

Finding a Function, Property, Structure, or ActiveX Component

Information on MSMQ SDK and MSMQ Mail SDK functions, properties, structures, and ActiveX components can be found in several places. The following information provides one way to get information on these topics. Once you become familiar with the Microsoft® Message Queue Server Programmer's Reference, you will develop your own methods to quickly find answers to your questions.

First, look in "MSMQ Reference" to find information on the specific item. A reference page for each item is included in this section of the software development kit.

From the reference page, use the jumps to look at examples in "Using MSMQ," or look at background information on the MSMQ concepts in the "MSMQ Guide."

For example, to find information on **MQCreateQueue** you could first go to **MQCreateQueue** in the reference section, then go to Creating a Queue for an example, or go to MSMQ Queues for background information on queues.

Finding Examples

To find an example you can go directly to "Using MSMQ," or you can go to any topic in "MSMQ Reference" and use the jumps provided there. In addition to the examples in "Using MSMQ," several smaller examples are provided for the ActiveX properties, methods, and events on their individual reference pages.

Using the MSMQ API Functions

This section contains complete examples of the basic tasks your application may need to perform.

Note For information on a specific MSMQ function, property, or structure, refer to its reference page in the "MSMQ Reference."

The following tasks are described:

- Creating a Queue
- Locating a Public Queue
- Opening a Queue
- Closing a Queue
- Deleting a Queue
- Sending Messages to a Queue
- Sending Messages that Request Acknowledgments
- Sending Messages that Request a Response
- Sending Private Messages
- Sending Messages Using an Internal Transaction
- Sending Messages Using an MS DTC External Transaction
- Reading Messages Synchronously
- Reading Messages Asynchronously
- Reading Messages Using a Cursor
- Reading Messages in a Dead Letter Queue
- Reading Messages in a Machine Journal
- Reading Messages in a Queue Journal
- Returning an Acknowledgment Message by a Connector Application
- Retrieving a Queue's Properties Using API Functions
- Setting a Queue's Properties Using API Functions
- Authenticating Messages Using API Functions
- Setting Access Control Security for a Queue
- Using Transactions

Creating a Queue

All queues, both public and private, are created by calling **MQCreateQueue**. For a description of public and private queues, see Message Queues.

The only property required to create a queue is PROPID_Q_PATHNAME. This property tells MSMQ where to store the queue's messages, if the queue is public or private, and the name of the queue. Once the queue is created, the format name returned in the *lpwcsFormatName* parameter is used to open the queue. For a description of MSMQ pathnames and queue format names, see Referencing a Queue.

To create a queue

1. Determine which computer will hold the messages for the queue. The computer's machine name is part of the queue's MSMQ pathname (PROPID_Q_PATHNAME). For private queues, the local computer must be specified.
2. Determine whether the queue should be public or private. This tells MSMQ where to register the queue: public queues are registered in the MQIS and private queues are registered on the local machine (private queues can only be registered on the local machine). This information is part of the queue's MSMQ pathname (PROPID_Q_PATHNAME).
3. Determine the name for the queue. The queue's name is part of the queue's MSMQ pathname (PROPID_Q_PATHNAME).

   Note The MSMQ pathname must be unique in the MSMQ enterprise. This applies to public and private queues.
4. Determine what queue properties must be set. If a queue property is not specified when calling **MQCreateQueue**, its default value is used. For a complete list of the queue properties that can be set when a queue is created, see the following Queue Properties section.
5. Specify the **MQQUEUEPROPS** structure.

```
MQQUEUEPROPS QueueProps;
PROPVARIANT aVariant[2];
QUEUEPROPID aPropId[2];
DWORD PropIdCount = 0;
HRESULT hr;
DWORD dwFormatNameBufferLength = 256;
WCHAR szFormatNameBuffer[256];
PSECURITY_DESCRIPTOR pSecurityDescriptor;
```

6. Fill in the **MQQUEUEPROPS** structure. PROPID_Q_PATHNAME is required; it indicates if the queue is public or private.

```
//Set the PROPID_Q_PATHNAME property.
aPropId[PropIdCount] = PROPID_Q_PATHNAME;      //PropId
aVariant[PropIdCount].vt = VT_LPWSTR;           //Type
aVariant[PropIdCount].pwszVal = L".\\MyPublicQueue";

PropIdCount++;

//Set the PROPID_Q_LABEL property.
aPropId[PropIdCount] = PROPID_Q_LABEL;      //PropId
aVariant[PropIdCount].vt = VT_LPWSTR;       //Type
aVariant[PropIdCount].pwszVal = L"MyPublicQueue";

PropIdCount++;
```

```
//Set the MQQUEUEPROPS structure.
QueueProps.cProp = PropIdCount;              //No of properties
QueueProps.aPropID = aPropId;                //Ids of properties
QueueProps.aPropVar = aVariant;              //Values of properties
QueueProps.aStatus = NULL;                   //No error reports
```

7. Set the queue's security descriptor. The following line of code specifies the default security descriptor.

```
pSecurityDescriptor = NULL;
```

8. Call MQCreateQueue.

```
hr = MQCreateQueue(
      pSecurityDescriptor,              //Security
      &QueueProps,                      //Queue properties
      szFormatNameBuffer,               //Output: Format Name
      &dwFormatNameBufferLength         //Output: Format Name len
      );
```

Examples

The following two examples show the code used to specify the MSMQ pathname and label for a public queue and a private queue, plus a call to **MQCreateQueue** to create the queue.

Note In these examples, a "." is used to indicate the local machine in PROPID_Q_PATHNAME. For MSMQ servers and independent clients, the local machine is the local computer. However, for MSMQ-dependent clients the local machine is the client's MSMQ server.

For a public queue

```
////////////////////////////
//   Define the MQQUEUEPROPS
//   structure.
////////////////////////////

MQQUEUEPROPS QueueProps;
PROPVARIANT aVariant[2];
QUEUEPROPID aPropId[2];
DWORD PropIdCount = 0;
HRESULT hr;
DWORD dwFormatNameBufferLength = 256;
TCHAR szFormatNameBuffer[ 256];
PSECURITY_DESCRIPTOR pSecurityDescriptor;


//////////////////////////////////////
// Specify the queue properties. Add
// additional properties as needed
//////////////////////////////////////

//Set the PROPID_Q_PATHNAME property.
aPropId[PropIdCount] = PROPID_Q_PATHNAME;      //PropId
aVariant[PropIdCount].vt = VT_LPWSTR;          //Type
aVariant[PropIdCount].pwszVal = L".\\MyPublicQueue";
```

```
PropIdCount++;

//Set the PROPID_Q_LABEL property.
aPropId[PropIdCount] = PROPID_Q_LABEL;      //PropId
aVariant[PropIdCount].vt = VT_LPWSTR;       //Type
aVariant[PropIdCount].pwszVal = L"MyPublicQueue";

PropIdCount++;


//Set the MQQUEUEPROPS structure.
QueueProps.cProp = PropIdCount;              //No of properties
QueueProps.aPropID = aPropId;                //Ids of properties
QueueProps.aPropVar = aVariant;              //Values of properties
QueueProps.aStatus = NULL;                   //No error reports

//Set security to default security descriptor
pSecurityDescriptor = NULL;


////////////////////////////
//Create the queue.
////////////////////////////
hr = MQCreateQueue(
      pSecurityDescriptor,              //Security
      &QueueProps,                      //Queue properties
      szFormatNameBuffer,               //Output: Format Name
      &dwFormatNameBufferLength         //Output: Format Name len
      );
```

For a private queue

```
MQQUEUEPROPS QueueProps;
PROPVARIANT aVariant[2];
QUEUEPROPID aPropId[2];
DWORD PropIdCount = 0;
HRESULT hr;
DWORD dwFormatNameBufferLength = 256;
WCHAR szFormatNameBuffer[ 256];
PSECURITY_DESCRIPTOR pSecurityDescriptor;


//////////////////////////////////////////
// Specify the queue properties. Add
// additional properties as needed
//////////////////////////////////////////

//Set the PROPID_Q_PATHNAME property.
aPropId[PropIdCount] = PROPID_Q_PATHNAME;      //PropId
aVariant[PropIdCount].vt = VT_LPWSTR;          //Type
aVariant[PropIdCount].pwszVal = L".\\private$\\MyPrivateQueue";

PropIdCount++;

//Set the PROPID_Q_LABEL property.
```

```
aPropId[PropIdCount] = PROPID_Q_LABEL;      //PropId
aVariant[PropIdCount].vt = VT_LPWSTR;       //Type
aVariant[PropIdCount].pwszVal = L"MyPrivateQueue";

PropIdCount++;

//Set the MQQUEUEPROPS structure.
QueueProps.cProp = PropIdCount;              //No of properties
QueueProps.aPropID = aPropId;                //Ids of properties
QueueProps.aPropVar = aVariant;              //Values of properties
QueueProps.aStatus = NULL;                   //No error reports

//Set security to default security descriptor.
pSecurityDescriptor = NULL;

//////////////////////
//Create the queue.
//////////////////////
hr = MQCreateQueue(
      pSecurityDescriptor,              //Security
      &QueueProps,                      //Queue properties
      szFormatNameBuffer,               //Output: Format Name
      &dwFormatNameBufferLength         //Output: Format Name len
      );
```

Queue Properties

The following optional queue properties can be set by the application when creating the queue:

PROPID_Q_AUTHENTICATE

PROPID_Q_BASEPRIORITY

PROPID_Q_JOURNAL

PROPID_Q_JOURNAL_QUOTA

PROPID_Q_LABEL

PROPID_Q_PRIV_LEVEL

PROPID_Q_QUOTA

PROPID_Q_TRANSACTION

PROPID_Q_TYPE

The following properties are set by MSMQ when it creates the queue:

PROPID_Q_CREATE_TIME

PROPID_Q_INSTANCE (for public queues)

PROPID_Q_MODIFY_TIME

Locating a Public Queue

Public queues can be located by running a query on the queue information registered in the MQIS. To run the query, the following three Locate functions are used: **MQLocateBegin**, **MQLocateNext**, and **MQLocateEnd**.

MQLocateBegin uses two sets of properties: One set specifies the properties used to locate the queues and the other set specifies the properties that will be included in the query results. For example, you may want to locate all the queues with a specific service type (PROPID_Q_TYPE) and only return their labels (PROPID_Q_LABEL). **MQLocateBegin** returns a handle to the query results.

Note If MSMQ finds a queue but the application does not have the access rights required to get the queue's properties, that queue is not included in the results of the query.

Once the results are available, **MQLocateNext** is called (as many times as needed) to navigate through the results. Finally, after the application is done using the query, **MQLocateEnd** is called to release the resources used for the query.

To run a query

1. Determine the search criteria for the query and what properties you want to retrieve.
2. Specify the search criteria using MQPROPERTYRESTRICTION and MQRESTRICTION.

```
// Set queue restriction to PROPID_Q_TYPE = PRINTER_SERVICE_TYPE.
PropertyRestriction.rel = PREQ;
PropertyRestriction.prop = PROPID_Q_TYPE;
PropertyRestriction.prval.vt = VT_CLSID;
PropertyRestriction.prval.puuid = &PRINTER_SERVICE_TYPE;

// Specify a one property restriction.
Restriction.cRes = 1;
Restriction.paPropRes = &PropertyRestriction;
```

3. Specify the properties to retrieve using MQCOLUMNSET.

```
MQCOLUMNSET     Column;
QUEUEPROPID     aPropId[2];       // only two properties to retrieve.
DWORD           dwColumnCount = 0;

aPropId[dwColumnCount] = PROPID_Q_INSTANCE;
dwColumnCount++;

aPropId[dwColumnCount] = PROPID_Q_CREATE_TIME;
dwColumnCount++;

Column.cCol = dwColumnCount;
Column.aCol = aPropId;
```

4. Call **MQLocateBegin**, to start the query.

```
HANDLE          hEnum;
hr = MQLocateBegin(
     NULL,            //start search at the top.
     &Restriction,    //Search criteria.
     &Column,         //Properties to return.
     NULL,            //No sort order
     &hEnum           //Enumeration Handle
     );
```

5. Call **MQLocateNext** to look at the query results.

```
hr = MQLocateNext(
      hEnum,        // Handle returned by MQLocateBegin.
      &cProps,      // Size of aPropVar array.
      aPropVar      // An array of PROPVARIANT for results.
      );
```

6. Call **MQLocateEnd** to close the query.

```
hr = MQLocateEnd(hEnum);    //Handle returned by MQLocateBegin.
```

Example

The following example shows the code used to locate all the queues of a specific type and return their queue identifier (PROPID_Q_INSTANCE) and when they were created (PROPID_Q_CREATE_TIME).

```
#define Max_PROPERTIES 13          //13 possible queue properties
CLSID PRINTER_SERVICE_TYPE =      //dummy GUID
      {0x1, 0x2, 0x3, 0x4, {0x5, 0x6, 0x7, 0x8, 0x9, 0xa}};
HRESULT hr;
MQPROPERTYRESTRICTION PropertyRestriction;
MQRESTRICTION   Restriction;


//////////////////////////////////////////
// Set search criteria according to the
// type of service provided by the queue.
//////////////////////////////////////////

// Set queue restriction to PROPID_Q_TYPE = PRINTER_SERVICE_TYPE.
PropertyRestriction.rel = PREQ;
PropertyRestriction.prop = PROPID_Q_TYPE;
PropertyRestriction.prval.vt = VT_CLSID;
PropertyRestriction.prval.puuid = &PRINTER_SERVICE_TYPE;

// Specify a one property restriction.
Restriction.cRes = 1;
Restriction.paPropRes = &PropertyRestriction;


//////////////////////////////////////////
// Set MQCOLUMNSET structure to specify
// the properties to be returned:
// PROPID_Q_INSTANCE and PROPID_Q_CREATE_TIME.
//////////////////////////////////////////

MQCOLUMNSET     Column;
QUEUEPROPID     aPropId[2];       // only two properties to retrieve.
DWORD           dwColumnCount = 0;

aPropId[dwColumnCount] = PROPID_Q_INSTANCE;
dwColumnCount++;

aPropId[dwColumnCount] = PROPID_Q_CREATE_TIME;
dwColumnCount++;
```

```
Column.cCol = dwColumnCount;
Column.aCol = aPropId;


////////////////////////////////////
// Call MQLocateBegin to start query.
////////////////////////////////////

HANDLE             hEnum;
hr = MQLocateBegin(
      NULL,               //start search at the top.
      &Restriction,     //Search criteria.
      &Column,            //Properties to return.
      NULL,               //No sort order
      &hEnum              //Enumeration Handle
      );

if(FAILED(hr))
{
      //
      //   Error handling
      //
}


////////////////////////////////////
// Call MQLocateNext to examine results
// of query.
////////////////////////////////////

PROPVARIANT aPropVar[MAX_PROPERTIES];
DWORD cProps, index;

do
   {
    cProps = MAX_PROPERTIES;
    hr = MQLocateNext(
          hEnum,         // Handle returned by MQLocateBegin.
          &cProps,      // Size of aPropVar array.
          aPropVar      // An array of PROPVARIANT for results.
          );

        if (FAILED(hr))
        {
          break;
        }

        for (index = 0; index < cProps; index += dwColumnCount)

        {
            //Process properties of a queue stored in:
            //aPropVar[index], aPropVar[index+1], ...,
            //aPropVar[index+dwColumnCount-1].
        }
```

```
   } while (cProps > 0);


////////////////////////////////////
// Call MQLocateEnd to end query.
////////////////////////////////////
hr = MQLocateEnd(hEnum);     //Handle returned by MQLocateBegin.
if(FAILED(hr))
{
      //
      //Error handling
      //
}
```

Opening a Queue

Queues can be opened for sending messages to the queue, retrieving messages from the queue, or peeking at the messages in the queue without removing them. All queues, both public and private, are opened by calling **MQOpenQueue**.

MQOpenQueue returns a queue handle that is used to:

- Send messages to the queue (**MQSendMessage**).
- Retrieve and peek at messages in the queue (**MQReceiveMessage**).
- Create a format name for the queue (**MQHandleToFormatName**).
- Create a cursor for navigating through the queue (**MQCreateCursor**).
- Close the queue (**MQCloseQueue**).

When opening a queue, the application specifies the access mode and share mode of the queue. The queue's access mode indicates if the application is going to send messages to the queue, peek at the messages in the queue, or retrieve messages from the queue. The queue's share mode indicates who else can use the queue while the application is using the queue.

Before opening a queue, MSMQ verifies that the access mode requested by the application is not restricted by the access rights of the queue. For example, a queue may restrict those who can send messages to it. For a discussion of queue access rights, see Access Control.

To open a queue

1. Obtain the format name of the queue. If the format name of the queue is not known, you can obtain a format name by using one of the following format name translation functions:

   MQInstanceToFormatName,

   MQPathNameToFormatName.

2. Set the queue's access mode. Are messages going to be sent to the queue (*dwAccess* = MQ_SEND_ACCESS), retrieved from the queue(*dwAccess* = MQ_RECEIVE_ACCESS), or peeked at without removing them from the queue (*dwAccess* = MQ_PEEK_ACCESS)?

   When a queue is opened with receive access, the application can also peek at the queue's messages. However, the reverse is not true. When a queue is opened with peek access, the application cannot retrieve a message from the queue.

   ```
   DWORD dwAccess = MQ_RECEIVE_ACCESS;
   ```

3. Set the queue's share mode. If messages are going to be retrieved from the queue (*dwAccess* = MQ_RECEIVE_ACCESS), determine if the application should stop others from retrieving messages at the same time it is retrieving messages (*dwShareMode* = MQ_DENY_RECEIVE_SHARE). Using this setting does not stop other applications from peeking at the messages in the queue, it only prevents them from retrieving messages at the same time the calling application is retrieving messages.

   ```
   DWORD dwShareMode = MQ_DENY_RECEIVE_SHARE;
   ```

4. Call **MQOpenQueue**.

   ```
   hr = MQOpenQueue(
         szwFormatNameBuffer,      // Format Name of queue.
         dwAccess,                 // Access mode of queue.
         dwShareMode,              // Exclusive mode of queue.
         &hQueue                   // OUT: Handle to queue.
         );
   ```

Example

This example opens a queue for reading messages.

```
//////////////////////
// Set the access mode.
//////////////////////
DWORD dwAccess = MQ_RECEIVE_ACCESS;

//////////////////////
// Set share mode.
//////////////////////
DWORD dwShareMode = MQ_DENY_RECEIVE_SHARE;

//////////////////////
// Call MQOpenQueue.
//////////////////////
QUEUEHANDLE hQueue;

hr = MQOpenQueue(
      szwFormatNameBuffer,      // Format Name of queue.
      dwAccess,                 // Access mode of queue.
      dwShareMode,              // Exclusive receive mode.
      &hQueue                   // OUT: Handle to queue.
      );
```

Closing a Queue

A queue is closed when its handle is no longer needed and its resources may be freed. Closing a queue is done with a single call to **MQCloseQueue**.

Note When an application closes a queue, the queue handle becomes invalid, but the messages waiting in the queue remain in the queue. This includes any messages sent to the queue by the application closing the queue.

To close a queue

1. Call **MQCloseQueue** to close the queue. Use the queue handle returned by **MQOpenQueue**.

```
hr   = MQCloseQueue(hQueue);    //handle obtained from MQOpenQueue.

If (FAILED(hr))
{
 // MQCloseQueue error handler.
}
```

Deleting a Queue

Deleting a queue is done with a single call to **MQDeleteQueue**.

To delete a queue

1. Obtain the format name of the queue. It is returned by **MQCreateQueue** when the queue is created. If the format name of the queue is not known, you can obtain a format name by using one of the following format name translation functions:

   MQHandleToFormatName

   MQInstanceToFormatName

   MQPathNameToFormatName.

2. Call **MQDeleteQueue** to delete the queue.

```
hr = MQDeleteQueue(szwFormatName);

If (FAILED(hr))
{
    // MQDeleteQueue error handler.
}
```

Example

The following example deletes the queue.

```
//////////////////
// Delete queue.
//////////////////
hr = MQDeleteQueue(szwFormatName);

If (FAILED(hr))
    {
    // MQDeleteQueue error handler.
}
```

Sending Messages to a Queue

Sending messages is typically a two-function operation: a single call to **MQOpenQueue** to open the queue and then one or more calls to **MQSendMessage**. After the queue is opened, the application can send any number of messages to the queue.

For examples of sending different types of messages, see:

- Sending Messages That Request Acknowledgments
- Sending Messages That Request a Response
- Sending Private Messages

Note Transaction messages can only be sent to transaction queues and non-transaction messages can only be sent to non-transaction queues.

Message are defined by a set of properties specified in an **MQMSGPROPS** structure. This structure provides the number of properties that will be sent with the message, identifiers for each property, and the value of each property.

Message properties

The following are general message properties that can be set by the application (length properties that are associated with another property are not included):

PROPID_M_ACKNOWLEDGE

PROPID_M_ADMIN_QUEUE

PROPID_M_APPSPECIFIC

PROPID_M_BODY

PROPID_M_BODY_TYPE

PROPID_M_CONNECTOR_TYPE

PROPID_M_CORRELATIONID

PROPID_M_DELIVERY

PROPID_M_EXTENSION

PROPID_M_JOURNAL

PROPID_M_LABEL

PROPID_M_PRIORITY

PROPID_M_RESP_QUEUE

PROPID_M_TIME_TO_BE_RECEIVED

PROPID_M_TIME_TO_REACH_QUEUE

PROPID_M_TRACE

PROPID_M_XACT_STATUS_QUEUE

Security message properties

Security properties are used for authenticating and encrypting messages. The following security message properties can be set by the application:

PROPID_M_AUTH_LEVEL

PROPID_M_DEST_SYMM_KEY

PROPID_M_ENCRYPTION_ALG

PROPID_M_HASH_ALG

PROPID_M_PRIV_LEVEL

PROPID_M_PROV_NAME

PROPID_M_PROV_TYPE

PROPID_M_SECURITY_CONTEXT

PROPID_M_SENDER_CERT

MSMQ-generated messages properties

The following message properties are set by MSMQ and should not be specified by the sending application. They are attached to the message when it is sent.

Message Property	Description
PROPID_M_ARRIVEDTIME	Indicates when the message arrived at its target queue.
PROPID_M_AUTHENTICATED	Indicates if the message was authenticated.
PROPID_M_CLASS	Indicates the type of message (such as normal, acknowledgment, or report).
PROPID_M_DEST_QUEUE	Indicates the destination of the message.
PROPID_M_MSGID	Indicates the message's MSMQ-generated identifier.
PROPID_M_SENDERID	Indicates the identifier of the sending application.
PROPID_M_SENDERID_TYPE	Indicates the type of identifier found by MSMQ.
PROPID_M_SENTTIME	Indicates when the message was sent.
PROPID_M_SIGNATURE	Indicates the digital signature of the message.
PROPID_M_SRC_MACHINE_ID	Indicates the machine identifier of the computer from where the message was sent.
PROPID_M_VERSION	Indicates what version of MSMQ the sending application is running.

Sending Messages that Request Acknowledgments

To request acknowledgment messages, the sending application must indicate the type of acknowledgment(s) it wants to receive and the administration queue where the acknowledgment message will be placed. It can receive positive, negative, or a combination of positive and negative acknowledgment messages by setting PROPID_M_ACKNOWLEDGE and attaching it to the messages it sends.

Acknowledgment messages are generated by MSMQ or by connector applications (when sending messages to foreign queues), and are returned to the administration queue specified by PROPID_M_ADMIN_QUEUE.

To send a message requesting acknowledgment messages

1. Call **MQOpenQueue** to open the queue with send access.

```
//////////////////////////////
// Open the destination queue
// with send access.
//////////////////////////////

QUEUEHANDLE hQueue;
hr = MQOpenQueue(szFormatName, MQ_SEND_ACCESS, 0, &hQueue);
if (FAILED(hr))
{
    fprintf(stderr, "Failed in MQOpenQueue, error = 0x%x\n", hr);
    return -1;
}
```

2. Set PROPID_M_ACKNOWLEDGE. This property specifies the type of acknowledgment messages that will be sent. The example below requests full-receive acknowledgments.

```
//////////////////////////////
// Set PROPID_M_ACKNOWLEDGE.
//////////////////////////////
aPropId[PropIdCount] = PROPID_M_ACKNOWLEDGE;              //PropId
aVariant[PropIdCount].vt = VT_UI1;                        //Type
aVariant[PropIdCount].bVal = MQMSG_ACKNOWLEDGMENT_FULL_RECEIVE;//Value

PropIdCount++;
```

3. Set PROPID_M_ADMIN_QUEUE. This property specifies the administration queue where the acknowledgment messages will be sent.

```
//////////////////////////////////////////
// Set the PROPID_M_ADMIN_QUEUE property.
//////////////////////////////////////////
aPropId[PropIdCount] = PROPID_M_ADMIN_QUEUE;            //PropId
aVariant[PropIdCount].vt = VT_LPWSTR;                   //Type
aVariant[PropIdCount].pwszVal = szwAdminFormatName;   //An already obtained format
name of the administration queue.

PropIdCount++;
```

4. Set other message properties, such as the message's body and its label.
5. Set the MQMSGPROPS structure.

```
//////////////////////////////
// Set the MQMSGPROPS structure.
```

```
////////////////////////////////
MsgProps.cProp = PropIdCount;          //Number of properties.
MsgProps.aPropID = aPropId;            //Ids of properties.
MsgProps.aPropVar = aVariant;          //Values of properties.
MsgProps.aStatus   = NULL;              //No Error report.
```

6. Call **MQSendMessage** to send the message to the queue.

```
/////////////////
// Send message.
/////////////////
hr = MQSendMessage(
      hQueue,                    // Handle to the Queue.
      &MsgProps,                 // Message properties to be sent.
      MQ_NO_TRANSACTION          // No transaction
      );
```

Example

The following example sends a message requesting full-receive acknowledgments.

```
MQMSGPROPS MsgProps;
PROPVARIANT aVariant[10];
MSGPROPID aPropId[10];
DWORD PropIdCount = 0;

HRESULT hr;

QUEUEHANDLE hQueue;

  ////////////////////////////////
  // Open the destination queue
  // with send access.
  ////////////////////////////////

  QUEUEHANDLE hQueue;
  hr = MQOpenQueue(szFormatName, MQ_SEND_ACCESS, 0, &hQueue);
  if (FAILED(hr))
  {
      fprintf(stderr, "Failed in MQOpenQueue, error = 0x%x\n", hr);
      return -1;
  }

////////////////////////////
// Set PROPID_M_ACKNOWLEDGE.
////////////////////////////
aPropId[PropIdCount] = PROPID_M_ACKNOWLEDGE;              //PropId
aVariant[PropIdCount].vt = VT_UI1;                        //Type
aVariant[PropIdCount].bVal = MQMSG_ACKNOWLEDGMENT_FULL_RECEIVE;//Value

PropIdCount++;

//////////////////////////////////////
// Set the PROPID_M_ADMIN_QUEUE property.
//////////////////////////////////////
aPropId[PropIdCount] = PROPID_M_ADMIN_QUEUE;              //PropId
```

```
aVariant[PropIdCount].vt = VT_LPWSTR;                    //Type
aVariant[PropIdCount].pwszVal = szwAdminFormatName;   //An already obtained format
name of the administration queue.

PropIdCount++;

////////////////////////////////////////
// Set other message properties, such
// as PROPID_M_BODY and PROPID_M_LABEL.
////////////////////////////////////////

//////////////////////////////////
// Set the MQMSGPROPS structure.
//////////////////////////////////
MsgProps.cProp = PropIdCount;          //Number of properties.
MsgProps.aPropID = aPropId;            //Ids of properties.
MsgProps.aPropVar = aVariant;          //Values of properties.
MsgProps.aStatus   = NULL;              //No Error report.

//////////////////
// Send message.
//////////////////
hr = MQSendMessage(
       hQueue,                     // handle to the Queue.
       &MsgProps,                  // Message properties to be sent.
       MQ_NO_TRANSACTION           // No transaction
       );

if (FAILED(hr))
    {
     //
     // Handle error condition
     //
     }
```

Sending Messages that Request a Response

To request response messages, the sending application must supply a response queue for the returned messages. Response messages are application-defined, and generated by the application reading the message.

To send a message that returns a response message

1. Call **MQOpenQueue** to open the queue with send access.

```
//////////////////////////////
// Open the destination queue
// with send access.
//////////////////////////////

QUEUEHANDLE hQueue;
hr = MQOpenQueue(szFormatName, MQ_SEND_ACCESS, 0, &hQueue);
if (FAILED(hr))
{
    fprintf(stderr, "Failed in MQOpenQueue, error = 0x%x\n", hr);
    return -1;
}
```

2. Set PROPID_M_RESP_QUEUE. This property specifies the response queue where the response messages will be sent.

```
//////////////////////////////////////
// Set the PROPID_M_RESPONSE_QUEUE property.
//////////////////////////////////////
aPropId[PropIdCount] = PROPID_M_RESPONSE_QUEUE;       //PropId
aVariant[PropIdCount].vt = VT_LPWSTR;                 //Type
aVariant[PropIdCount].pwszVal = szwRespFormatName;   //An already obtained format
name of the response queue.

PropIdCount++;
```

3. Set other message properties such as the message's body and its label.
4. Set the MQMSGPROPS structure.

```
//////////////////////////////
// Set the MQMSGPROPS structure.
//////////////////////////////
MsgProps.cProp = PropIdCount;          //Number of properties.
MsgProps.aPropID = aPropId;           //Ids of properties.
MsgProps.aPropVar = aVariant;         //Values of properties.
MsgProps.aStatus   = NULL;             //No Error report.
```

5. Call **MQSendMessage** to send the message to the queue.

```
/////////////////
// Send message.
/////////////////
hr = MQSendMessage(
      hQueue,                    // handle to the Queue.
      &MsgProps,                 // Message properties to be sent.
      MQ_NO_TRANSACTION          // No transaction
      );
```

Example

The following example sends a message requesting full receive acknowledgments.

```
MQMSGPROPS MsgProps;
PROPVARIANT aVariant[10];
MSGPROPID aPropId[10];
DWORD PropIdCount = 0;

HRESULT hr;

  ////////////////////////////////
  // Open the destination queue
  // with send access.
  ////////////////////////////////

  QUEUEHANDLE hQueue;
  hr = MQOpenQueue(szFormatName, MQ_SEND_ACCESS, 0, &hQueue);
  if (FAILED(hr))
  {
       fprintf(stderr, "Failed in MQOpenQueue, error = 0x%x\n", hr);
       return -1;
  }

QUEUEHANDLE hQueue;

//////////////////////////////////////////
// Set the PROPID_M_RESP_QUEUE property.
//////////////////////////////////////////
aPropId[PropIdCount] = PROPID_M_RESP_QUEUE;             //PropId
aVariant[PropIdCount].vt = VT_LPWSTR;                       //Type
aVariant[PropIdCount].pwszVal = szwRespFormatName;   //An already obtained format
name of the response queue.

PropIdCount++;

//////////////////////////////////////
// Set other message properties, such
// as PROPID_M_BODY and PROPID_M_LABEL.
//////////////////////////////////////

////////////////////////////////
// Set the MQMSGPROPS structure.
////////////////////////////////
MsgProps.cProp = PropIdCount;          //Number of properties.
MsgProps.aPropID = aPropId;            //Ids of properties.
MsgProps.aPropVar = aVariant;          //Values of properties.
MsgProps.aStatus   = NULL;              //No Error report.

/////////////////
// Send message.
/////////////////
hr = MQSendMessage(
       hQueue,                      // handle to the Queue.
       &MsgProps,                   // Message properties to be sent.
       MQ_NO_TRANSACTION            // No transaction
```

```
        );

if (FAILED(hr))
    {
     //
     // Handle error condition
     //
     }
```

Sending Private Messages

To send a private message , the sending application must set the privacy level of the message. Once the privacy level is set, the functions used to send the message are no different from those used to send other messages. When sending (and receiving) private messages, the application has no part in encrypting (or decrypting) the message.

Note For information on how MSMQ encrypts and decrypts messages, see Private Messages.

To send a private message

1. Obtain the format name of the destination queue. The example below calls **MQPathNameToFormatName** to obtain a format name from a known pathname.

```
WCHAR szFormatName[128];
DWORD dwFormatNameLen = sizeof(szFormatName) / sizeof(WCHAR);

hr = MQPathNameToFormatName(L"dest_machine\\secrets",
                            szFormatName,
                            &dwFormatNameLen);
```

2. Call **MQOpenQueue** to open the destination queue with send access.

```
QUEUEHANDLE hQueue;
hr = MQOpenQueue(szFormatName, MQ_SEND_ACCESS, 0, &hQueue);
```

3. Set message properties. The value of PROPID_M_PRIV_LEVEL indicates that the message is private.

```
#define NMSGPROPS 10
MSGPROPID aMsgPropId[NMSGPROPS];
MQPROPVARIANT aMsgPropVar[NMSGPROPS];
HRESULT aMsgStatus[NMSGPROPS];
MQMSGPROPS MsgProps;
DWORD PropIdCount = 0;

//Set PROPID_M_LABEL
aMsgPropId[PropIdCount] = PROPID_M_LABEL;              //PropId
aMsgPropVar[PropIdCount].vt = VT_LPWSTR;               //Type
aMsgPropVar[PropIdCount].pwszVal = L"Hash hash";       //Value
PropIdCount++

//Set PROPID_M_BODY
#define MESSAGE_BODY L"Secret matters"
aMsgPropId[PropIdCount] = PROPID_M_BODY;               //PropId
aMsgPropVar[PropIdCount].vt = VT_VECTOR | VT_UI1;
aMsgPropVar[PropIdCount].caub.pElems = (LPBYTE)MESSAGE_BODY;
aMsgPropVar[PropIdCount].caub.cElems = sizeof(MESSAGE_BODY);
PropIdCount++

//Set PROPID_M_PRIV_LEVEL
aMsgPropId[PropIdCount] = PROPID_M_PRIV_LEVEL;         //PropId
aMsgPropVar[PropIdCount].vt = VT_UI4;                  //Type
aMsgPropVar[PropIdCount].ulVal = MQMSG_PRIV_LEVEL_BODY;
 PropIdCount++

//Optional. Set PROPID_M_ENCRYPTION_ALG.
```

```
aMsgPropId[PropIdCount] = PROPID_M_ENCRYPTION_ALG;    //PropId
aMsgPropVar[PropIdCount].vt = VT_UI4;                 //Type
aMsgPropVar[PropIdCount].ulVal = CALG_RC4;     //Default is RC2.
 PropIdCount++

//Set the MQMSGPROPS structure.
MsgProps.cProp = PropIdCount;          //Number of properties.
MsgProps.aPropID = aMsgPropId;         //Id of properties.
MsgProps.aPropVar = aMsgVariant;       //Value of properties.
MsgProps.aStatus   = aMsgStatus;       //Error report.
```

4. Call **MQSendMessage** to send the message.

```
hr = MQSendMessage(hQueue, &MsgProps, NULL);
```

5. Call **MQCloseQueue** to close the destination queue.

```
hr = MQCloseQueue(hQueue);
```

Example

This example starts with a known pathname for the destination queue, translates the pathname to a format name, opens the destination queue with send access, sets the properties of the message (including the privacy level of the message), then sends the message to the destination queue.

```
#include <windows.h>
#include <wincrypt.h>
#include <stdio.h>
#include <mq.h>

int main(int argc, char *argv[])
{
    HRESULT hr;


    ////////////////////////////
    // Get the format name of
    // the destination queue.
    ////////////////////////////

    WCHAR szFormatName[128];
    DWORD dwFormatNameLen = sizeof(szFormatName) / sizeof(WCHAR);

    hr = MQPathNameToFormatName(L"dest_machine\\secrets",
                                szFormatName,
                                &dwFormatNameLen);
    if (FAILED(hr))
    {
        fprintf(stderr, "Failed in MQPathNameToFormatName, error = 0x%x\n", hr);
        return -1;
    }

    ////////////////////////////////
    // Open the destination queue
    // with send access.
    ////////////////////////////////
```

```
QUEUEHANDLE hQueue;
hr = MQOpenQueue(szFormatName, MQ_SEND_ACCESS, 0, &hQueue);
if (FAILED(hr))
{
     fprintf(stderr, "Failed in MQOpenQueue, error = 0x%x\n", hr);
     return -1;
}


////////////////////////////////////
// Define the MQMSGPROPS structure.
////////////////////////////////////

#define NMSGPROPS 10
MSGPROPID aMsgPropId[NMSGPROPS];
MQPROPVARIANT aMsgPropVar[NMSGPROPS];
HRESULT aMsgStatus[NMSGPROPS];
MQMSGPROPS MsgProps;
DWORD PropIdCount = 0;

//Set PROPID_M_LABEL
aMsgPropId[PropIdCount] = PROPID_M_LABEL;                //PropId
aMsgPropVar[PropIdCount].vt = VT_LPWSTR;                 //Type
aMsgPropVar[PropIdCount].pwszVal = L"Hash hash";       //Value
PropIdCount++

//Set PROPID_M_BODY
#define MESSAGE_BODY L"Secret matters"
aMsgPropId[PropIdCount] = PROPID_M_BODY;                  //PropId
aMsgPropVar[PropIdCount].vt = VT_VECTOR | VT_UI1;
aMsgPropVar[PropIdCount].caub.pElems = (LPBYTE)MESSAGE_BODY;
aMsgPropVar[PropIdCount].caub.cElems = sizeof(MESSAGE_BODY);
PropIdCount++

//Set PROPID_M_PRIV_LEVEL
aMsgPropId[PropIdCount] = PROPID_M_PRIV_LEVEL;          //PropId
aMsgPropVar[PropIdCount].vt = VT_UI4;                       //Type
aMsgPropVar[PropIdCount].ulVal = MQMSG_PRIV_LEVEL_BODY;
 PropIdCount++

//Optional. Set PROPID_M_ENCRYPTION_ALG.
aMsgPropId[PropIdCount] = PROPID_M_ENCRYPTION_ALG;    //PropId
aMsgPropVar[PropIdCount].vt = VT_UI4;                       //Type
aMsgPropVar[PropIdCount].ulVal = CALG_RC4;      //Default is RC2.
 PropIdCount++

//Set the MQMSGPROPS structure.
MsgProps.cProp = PropIdCount;          //Number of properties.
MsgProps.aPropID = aMsgPropId;         //Id of properties.
MsgProps.aPropVar = aMsgVariant;      //Value of properties.
MsgProps.aStatus   = aMsgStatus;       //Error report.

//////////////////
// Send the message
//////////////////
```

```
    hr = MQSendMessage(hQueue, &MsgProps, NULL);
    if (FAILED(hr))
    {
     fprint(stderr, "Failed in MQSendMessage, error = 0x%x\n", hr);
     return -1;
    }

    ////////////////////
    // Close the queue.
    ////////////////////
    MQCloseQueue(hQueue);

    if (FAILED(hr))
    {
      fprintf(stderr, "Failed in MQCloseQueue, error = 0x%x\n", hr);
      return -1;
    }
    printf("The private message was sent successfully\n");

    return 0;
}
```

Sending Messages Using an Internal Transaction

To send a message using an internal transaction, call **MQBeginTransaction** to initiate the transaction and call **MQSendMessage** to send the message.

To send a message using an internal transaction

1. Call **MQBeginTransaction** to initiate internal transaction.

```
ITransaction   *pTransaction;
hr = MQBeginTransaction (&pTransaction);     // Pointer to a pointer
                                             // to the transaction
                                             // object.
```

2. Call **MQSendMessage** to send message.

```
hr = MQSendMessage(h,              // Handle to destination queue.
                   &msgprops,     // Pointer to MQMSGPROPS structure.
                   pTransaction); // Pointer to transaction object.
```

3. Commit or abort the transaction.

```
hr = pTransaction->Commit(0, 0, 0);
-or-
hr = pTransaction->Abort(0, 0, 0);
```

4. Release the transaction object.

```
pTransaction->Release();
```

Example

This example sends a single message within an internal transaction.

```
void TransactSend(QUEUEHANDLE h, MQMSGPROPS * pMsgProps)
{
        HRESULT hr;
        printf ("\nStarting transaction...\n\n");

        ////////////////////////////////////////
        // Call MQBeginTransaction to initiate
        // the internal transaction.
        ////////////////////////////////////////
        ITransaction   *pTransaction;
        hr = MQBeginTransaction (&pTransaction);     // Pointer to a
                                                     // pointer to the
                                                     // transaction object

        if (FAILED(hr))
        {
            Error ("BeginTransaction",hr);
        }

        ////////////////////////////////////
        // Set default to commit the
        // transaction.
        ////////////////////////////////////
        BOOL fCommit = TRUE;

        ////////////////////////////////////
```

```
    // Within the transaction: Call
    // MQSendMessage to send the message to
    // the Receiver Side.
    ////////////////////////////////////
    hr = MQSendMessage(h,                   // Handle to destination
                                            // queue.
                       &msgprops,           // Pointer to MQMSGPROPS
                                            // structure.
                       pTransaction);   // Pointer to transaction
                                            // object.

    if (FAILED(hr))
    {
        printf("\nFailed in MQSendMessage(). hresult- %lxh\n", (DWORD) hr) ;
        fCommit = FALSE;   // Aborting as MQSendMessage failed.
    }

    ////////////////////////////////////
    // Commit or abort the transaction.
    ////////////////////////////////////
    if (fCommit)
    {
        printf ("Committing the transaction...     ");
        hr = pTransaction->Commit(0, 0, 0);
        if (FAILED(hr))
            printf ("Failed... Transaction aborted.\n\n");
        else
            printf ("Transaction committed successfully.\n\n");
    }
    else
    {
        printf ("Aborting the transaction...     ");
        hr = pTransaction->Abort(0, 0, 0);
        if (FAILED(hr))
            Error("Transaction Abort",hr);
        else
            printf ("Transaction aborted.\n\n");
    }

    ////////////////////////////
    // Release the transaction.
    ////////////////////////////
    pTransaction->Release();

}
```

Sending Messages Using an MS DTC External Transaction

To send a message using a Microsoft® Distributed Transaction Coordinator (MS DTC) external transaction, the application must work with all the resource managers that are needed to complete the transaction. In the example below, the only resource manager used is MSMQ.

To send a message using an MS DTC external transaction

1. Call DtcGetTransactionManager to get a transaction dispenser. For information on DtcGetTransactionManager, see the Microsoft Platform SDK.

```
hr = DtcGetTransactionManager(
        NULL,                           // pszHost
        NULL,                           // pszTmName
        IID_ITransactionDispenser,      // IID of interface
        0,                              // Reserved: must be null.
        0,                              // Reserved: must be null.
        0,                              // Reserved: must be null.
        (void **)&g_pTransactionDispenser   // Pointer to pointer
                                            // to requested
                                            // interface.
                                );
```

2. Initiate the transaction.

```
hr = g_pTransactionDispenser->BeginTransaction (
        0,                              // Must be null.
        ISOLATIONLEVEL_ISOLATED,    // Isolation level.
        ISOFLAG_RETAIN_DONTCARE,    // Isolation flags.
        0,                              // Pointer to transaction
                                        // options object.
        &pTransaction);                 // Pointer to a pointer to
                                        // transaction object.
```

3. Call **MQSendMessage** to send message.

```
hr = MQSendMessage(h,               // Handle to destination queue.
                   &msgprops,   // Pointer to MQMSGPROPS
                                // structure.
                    pTransaction); // Pointer to transaction
                                   // object.
```

4. Commit or abort the transaction.

```
hr = pTransaction->Commit(0, 0, 0);
-or-
hr = pTransaction->Abort(0, 0, 0);
```

5. Release the transaction object.

```
pTransaction->Release();
```

Example

This example sends a single message within an MS DTC external transaction.

```
ITransactionDispenser    *g_pTransactionDispenser;

BOOL InitCoordinatedTransactions()
```

```
{

    ////////////////////////////////
    // Get transaction dispenser.
    ////////////////////////////////

    // Obtain an interface pointer from MS DTC proxy
    hr = DtcGetTransactionManager(
            NULL,                                   // pszHost
            NULL,                                   // pszTmName
            IID_ITransactionDispenser,              // IID of interface
            0,                                      // Reserved: must be null.
            0,                                      // Reserved: must be null.
            0,                                      // Reserved: must be null.
            (void **)&g_pTransactionDispenser   // pointer to pointer to
                                                    // requested interface.
                                        );

    if (FAILED(hr))
    {
      //
      // No Connection to DTC.
      //
    return(FALSE);
    }

    return(TRUE);

}

void TransactSend(QUEUEHANDLE h, MQMSGPROPS * pMsgProps)
{

  ITransaction                  *pTransaction;
  printf ("\nStarting transaction...\n\n");

  ////////////////////////////////
  // Initiate a transaction.
  ////////////////////////////////

  hr = g_pTransactionDispenser->BeginTransaction (
          0,                                // Must be null.
          ISOLATIONLEVEL_ISOLATED,     // Isolation level.
          ISOFLAG_RETAIN_DONTCARE,     // Isolation flags.
          0,                                // Pointer to transaction
                                            // options object.
          &pTransaction);                 // Pointer to a pointer to
                                            // transaction object.

        if (FAILED(hr))
        {
            Error ("BeginTransaction",hr);
        }

        // Default is to commit transaction
```

```
    BOOL fCommit = TRUE;

    ////////////////////////////////////////////
    // Call MQSendMessage to send message to
    // the receiver side within the transaction.
    ////////////////////////////////////////////
    hr = MQSendMessage(h,               // Handle to destination queue
                       pMsgprops,     // Pointer to MQMSGPROPS
                                      // structure.
                       pTransaction); // Pointer to transaction
                                      // Object


    if (FAILED(hr))
    {
        printf("\nFailed in MQSendMessage(). hresult- %lxh\n", (DWORD) hr) ;
        fCommit = FALSE;        // Abort if MQSend failed
    }

    ////////////////////////////////////////////////
    // Here the application can call other resource
    // managers (such as SQL server) and enlist their
    // actions in the transaction pTransaction.   If
    // atomicity is required, set fCommit to FALSE.


    // Commit the transaction or abort it
    if (fCommit)
    {
        printf ("Committing the transaction...     ");

        hr = pTransaction->Commit(0, 0, 0);

        if (FAILED(hr))
            printf ("Failed... Transaction aborted.\n\n");
        else
            printf ("Transaction committed successfully.\n\n");

    }
    else
    {
        printf ("Aborting the transaction...     ");

        hr = pTransaction->Abort(0, 0, 0);

        if (FAILED(hr))
            Error("Transaction Abort",hr);
        else
            printf ("Transaction aborted.\n\n");
    }

    // Release the transaction
    pTransaction->Release();

}
```

```
void CleanupTransaction()
{
    ////////////////////////////////////////////////
    // Cleanup and release the transaction object.
    ////////////////////////////////////////////////

    g_pTransactionDispenser->Release();
}
```

Reading Messages in a Queue

Reading messages in a queue is a two-function operation if the application only reads the first message in the queue, and a three-function operation if the application needs to navigate through the queue. To read the first message in the queue, the application must call **MQOpenQueue** and **MQReceiveMessage**. However, to read messages that are not at the front of the queue, the application must call **MQOpenQueue**, **MQCreateCursor**, then **MQReceiveMessage**.

For examples, see:

- Reading Messages Synchronously
- Reading Messages Asynchronously

When reading messages in a queue, MSMQ can peek at the messages or retrieve them.

When reading messages, the application can retrieve all the message properties that are provided by MSMQ. It is the application's responsibility to specify which message properties it wants to read (this includes the body of the message).

Message properties include:

PROPID_M_ACKNOWLEDGE

PROPID_M_ADMIN_QUEUE

PROPID_M_ADMIN_QUEUE_LEN

PROPID_M_APPSPECIFIC

PROPID_M_ARRIVEDTIME

PROPID_M_AUTH_LEVEL

PROPID_M_AUTHENTICATED

PROPID_M_BODY

PROPID_M_BODY_SIZE

PROPID_M_BODY_TYPE

PROPID_M_CLASS

PROPID_M_CONNECTOR_TYPE

PROPID_M_CORRELATIONID

PROPID_M_DELIVERY

PROPID_M_DEST_QUEUE

PROPID_M_DEST_QUEUE_LEN

PROPID_M_DEST_SYMM_KEY

PROPID_M_DEST_SYMM_KEY_LEN

PROPID_M_ENCRYPTION_ALG

PROPID_M_EXTENSION

PROPID_M_EXTENSION_LEN

PROPID_M_HASH_ALG

PROPID_M_JOURNAL

PROPID_M_LABEL

PROPID_M_LABEL_LEN

PROPID_M_MSGID

PROPID_M_PRIORITY

PROPID_M_PRIV_LEVEL

PROPID_M_PROV_NAME

PROPID_M_PROV_NAME_LEN

PROPID_M_PROV_TYPE

PROPID_M_RESP_QUEUE

PROPID_M_RESP_QUEUE_LEN

PROPID_M_SECURITY_CONTEXT

PROPID_M_SENDER_CERT

PROPID_M_SENDER_CERT_LEN

PROPID_M_SENDERID

PROPID_M_SENDERID_LEN

PROPID_M_SENDERID_TYPE

PROPID_M_SENTTIME

PROPID_M_SIGNATURE

PROPID_M_SIGNATURE_LEN

PROPID_M_SRC_MACHINE_ID

PROPID_M_TIME_TO_BE_RECEIVED

PROPID_M_TIME_TO_REACH_QUEUE

PROPID_M_TRACE

PROPID_M_VERSION

PROPID_M_XACT_STATUS_QUEUE

PROPID_M_XACT_STATUS_QUEUE_LEN

Reading Messages Synchronously

When reading messages synchronously, all calls are blocked until the next message is available or timeout occurs.

To read a message synchronously

1. Open the queue with receive or peek access.

```
hr = MQOpenQueue(
      szwFormatNameBuffer,      // Format name of the queue.
      MQ_RECEIVE_ACCESS,        // Access rights to the Queue.
      0,                        // No receive Exclusive.
      &hQueue                   // OUT: handle to the opened Queue.
);
```

2. Specify the message properties to be retrieved. For example, if you only need to look at the body of the message, only specify PROPID_M_BODY.

```
//Set the message body property.
aPropId[PropIdCount] = PROPID_M_BODY;                 //PropId
aVariant[PropIdCount].vt = VT_VECTOR|VT_UI1;          //Type
aVariant[PropIdCount].caub.cElems = MSG_BODY_LEN;   //Buffer size
aVariant[PropIdCount].caub.pElems = ucMsgBody;       //Buffer

      PropIdCount++;
```

3. Set the MQMSGPROPS structure.

```
// Set the MQMSGPROPS structure.
MsgProps.cProp = PropIdCount;          //Number of properties.
MsgProps.aPropID = aPropId;            //Ids of properties.
MsgProps.aPropVar = aVariant;          //Values of properties.
MsgProps.aStatus   = NULL;             //No Error report.
```

4. Retrieve the message from the queue.

```
hr = MQReceiveMessage(
      hQueue,                 // Handle to the Queue.
      5 * 60 * 1000,          // Timeout value (msec) to wait for
                              // messages (5*60*1000=5 min.).
      MQ_ACTION_RECEIVE,     // Action.
      &MsgProps,              // Properties to retrieve.
      NULL,                   // Must be NULL for synchronous receive.
      NULL,                   // Must be NULL for synchronous receive.
      NULL,                   // No Cursor.
      MQ_NO_TRANSACTION      // No transaction.
                        );
```

Example

The following example opens a queue with receive access, specifies the body of the message as the only property to be retrieved, then reads the first message in the queue as a non-transactional, synchronous operation.

```
HRESULT hr

//Open Queue
WCHAR * szwFormatNameBuffer;   // Format Name of the queue to be opened
QUEUEHANDLE hQueue;
```

```
// Obtain format name of queue.

hr = MQOpenQueue(
      szwFormatNameBuffer,      // Format Name of the queue to be opened.
      MQ_RECEIVE_ACCESS,         // Access rights to the Queue.
      0,                              // No receive Exclusive.
      &hQueue                         // OUT: handle to the opened Queue.
                    );

if (FAILED(hr))
{
 // Error handler for MQOpenQueue.
}


MQMSGPROPS MsgProps;
MQPROPVARIANT aVariant[10];
MSGPROPID aPropId[10];
DWORD PropIdCount = 0;

//
// Prepare the message properties to be retrieved.
//

#define MSG_BODY_LEN 500
unsigned char ucMsgBody[MSG_BODY_LEN];

// Set the PROPID_M_BODY property.
aPropId[PropIdCount] = PROPID_M_BODY;                    //PropId
aVariant[PropIdCount].vt = VT_VECTOR|VT_UI1;          //Type
aVariant[PropIdCount].caub.cElems = MSG_BODY_LEN;   //Buffer size.
aVariant[PropIdCount].caub.pElems = ucMsgBody;       //Buffer

PropIdCount++;

//
// Set other properties.
//

// Set the MQMSGPROPS structure
MsgProps.cProp = PropIdCount;          //Number of properties.
MsgProps.aPropID = aPropId;            //Ids of properties.
MsgProps.aPropVar = aVariant;          //Values of properties.
MsgProps.aStatus   = NULL;              //No Error report.

//
// Retrieve the message.
//
hr = MQReceiveMessage(
      hQueue,                    // handle to the Queue.
       5 * 60 * 1000,         // Timeout value (msec) to wait for
                              // for messages (5*60*1000=5 min.).
      MQ_ACTION_RECEIVE,     // Action.
      &MsgProps,               // properties to retrieve.
```

```
        NULL,                   // No overlapped structure.
        NULL,                   // No callback function.
        NULL,                   // No Cursor.
        MQ_NO_TRANSACTION       // No transaction
                        );


if (FAILED(hr))
        {
        //   Error handler for MQReceiveMessage.
        }
```

Reading Messages Asynchronously

Applications can use a callback function, a Windows Event mechanism, or a Microsoft® Windows NT® completion port to read messages asynchronously. When reading messages asynchronously, the application is notified if a message is available or if a timeout has occurred.

For information on MQReceiveMessage, see **MQReceiveMessage**.

Note In the following two examples, the first uses a callback function to retrieve the messages and the second uses a Windows Event mechanism. Both examples read the first message in the queue.

To read a message asynchronously using a callback function

1. Write the callback function.

```
void APIENTRY ReceiveCallbackRoutine(
     HRESULT hrStatus,
     QUEUEHANDLE hSource,
     DWORD dwTimeout,
     DWORD dwAction,
     MQMSGPROPS* pMessageProps,
     LPOVERLAPPED lpOverlapped,
     HANDLE hCursor
)
 {
   // Process message.
 }
```

2. Open the queue with receive or peek access. If the queue is opened with receive access the application can still peek at the messages in the queue.

```
hr = MQOpenQueue(
     szwFormatNameBuffer,      // Format name of the queue.
     MQ_RECEIVE_ACCESS,        // Access rights to the queue.
     0,                        // No receive Exclusive.
     &hQueue                   // OUT: handle to the opened queue.
                 );
```

3. Specify the message properties to be retrieved, retrieving only those properties that are needed. For example, if you only need to look at the body of the message, only specify PROPID_M_BODY.

```
// Set the PROPID_M_BODY property.
paPropId[dwPropIdCount] = PROPID_M_BODY;                  //PropId
paVariant[dwPropIdCount].vt = VT_VECTOR|VT_UI1;           //Type
paVariant[dwPropIdCount].caub.cElems = MSG_BODY_LEN ;     //Value
paVariant[dwPropIdCount].caub.pElems = new unsigned char[ MSG_BODY_LEN];

 dwPropIdCount++;
```

4. Set the MQMSGPROPS structure.

```
// Set the MQMSGPROPS structure.
MsgProps.cProp = PropIdCount;         //Number of properties.
MsgProps.aPropID = aPropId;           //Ids of properties.
MsgProps.aPropVar = aVariant;         //Values of properties.
MsgProps.aStatus   = NULL;            //No Error report.
```

5. Read message from the queue, using ReceiveCallbackRoutine as the callback function.

```
   hr = MQReceiveMessage(
        hQueue,                        // handle to the Queue.
        5 * 60 * 1000,                 // Max time (msec) to wait.
        MQ_ACTION_RECEIVE,             // Action.
        pMsgProps,                     // Properties to retrieve.
        NULL,                          // No overlapped structure.
        ReceiveCallbackRoutine,        // Callback function.
        NULL,                          // No Cursor.
        NULL                           // No transaction
                    );
```

Callback Function Example

The following example specifies a callback function, opens a queue with receive access, specifies the body of the message as the only message property to retrieve, then reads the first message of the queue using the callback function.

```
//////////////////////////////
// Receive callback function.
//////////////////////////////

void APIENTRY ReceiveCallbackRoutine(
    HRESULT hr,
    QUEUEHANDLE hSource,
    DWORD dwTimeout,
    DWORD dwAction,
    MQMSGPROPS* pMessageProps,
    LPOVERLAPPED lpOverlapped,
    HANDLE hCursor
  )
 {
  if (FAILED(hr))
    {
     // Error handler for Callback routine.
    }
  else
  {
   // Process message.
  }
 }

//////////////
// Open Queue
//////////////

HRESULT hr;
QUEUEHANDLE hQueue;

hr = MQOpenQueue(
       szwFormatNameBuffer,      // Format Name of the queue to be opened.
       MQ_RECEIVE_ACCESS,        // Access rights to the Queue.
       0,                        // No receive Exclusive.
       &hQueue                   // OUT: handle to the opened Queue.
);

if (FAILED(hr))
```

```
{
 // Error handler for MQOpenQueue.
}


MQMSGPROPS * pMsgProps;
MQPROPVARIANT *paVariant;
MSGPROPID * paPropId;
DWORD dwPropIdCount = 0;

//
//   The output parameters to an asynchronous call to MQReceiveMessage
//   should be kept intact until the operation completes, you should
//   not free or reuse them until the operation is complete.
//
pMsgProps = new MQMSGPROPS;
paVariant = new MQPROPVARIANT[ 10];
paPropId = new MSGPROPID[ 10];


////////////////////////////////////////////////
// Prepare the message properties to be retrieved.
////////////////////////////////////////////////


// Set the PROPID_M_BODY property.
paPropId[dwPropIdCount] = PROPID_M_BODY;                    //PropId
paVariant[dwPropIdCount].vt = VT_VECTOR|VT_UI1;             //Type
paVariant[dwPropIdCount].caub.cElems = MSG_BODY_LEN ;      //Value
paVariant[dwPropIdCount].caub.pElems = new unsigned char[ MSG_BODY_LEN];

 dwPropIdCount++;

////////////////////////////////
// Set the MQMSGPROPS structure
////////////////////////////////
pMsgProps->cProp = dwPropIdCount;       //Number of properties.
pMsgProps->aPropID = paPropId;          //Ids of properties.
pMsgProps->aPropVar = paVariant;        //Values of properties.
pMsgProps->aStatus   = NULL;             //No Error report.


////////////////////////////////////////////
//   Receive the message using callback function
//   ReceiveCallbackRoutine.
////////////////////////////////////////////////

hr = MQReceiveMessage(
      hQueue,                       // handle to the Queue.
      5 * 60 * 1000,                // Max time (msec) to wait.
      MQ_ACTION_RECEIVE,              // Action.
      pMsgProps,                     // properties to retrieve.
      NULL,                         // No overlapped structure.
      ReceiveCallbackRoutine,       // Callback function.
      NULL,                         // No Cursor.
```

```
    NULL                            // No transaction
                );

if (FAILED(hr))
    {
      //   Error handler for MQReceiveMessage.
     }
```

To read a message asynchronously using a Windows Event mechanism

1. Open the queue with receive or peek access. The access mode used to open the queue does not determine how the messages are read from the queue. For example, if the queue is opened with receive access the application can still peek at the messages in the queue.

```
hr = MQOpenQueue(
      szwFormatNameBuffer,      // Format name of the queue.
      MQ_RECEIVE_ACCESS,        // Access rights to the Queue.
      0,                        // No receive Exclusive.
      &hQueue                   // OUT: handle to the opened Queue.
                  );
```

2. Specify the message properties to be retrieved. If you only need to look at the body of the message, only specify PROPID_M_BODY.

```
// Set the PROPID_M_BODY property.
paPropId[dwPropIdCount] = PROPID_M_BODY;                    //PropId
paVariant[dwPropIdCount].vt = VT_VECTOR|VT_UI1;             //Type
paVariant[dwPropIdCount].caub.cElems = MSG_BODY_LEN ;     //Value
paVariant[dwPropIdCount].caub.pElems = new unsigned char[ MSG_BODY_LEN];

 dwPropIdCount++;
```

3. Set the MQMSGPROPS structure.

```
// Set the MQMSGPROPS structure.
MsgProps.cProp = PropIdCount;          //Number of properties.
MsgProps.aPropID = aPropId;            //Ids of properties.
MsgProps.aPropVar = aVariant;          //Values of properties.
MsgProps.aStatus   = NULL;             //No Error report.
```

4. Create Event object using overlapped structure.

```
OVERLAPPED *pov = new OVERLAPPED ;
pov->hEvent = CreateEvent(0, TRUE, TRUE, 0);
```

5. Read message from the queue.

```
hr = MQReceiveMessage(
      hQueue,                   // handle to the Queue.
      5 * 60 * 1000,            // Max time (msec) to wait.
      MQ_ACTION_RECEIVE,        // Action.
      pMsgProps,                // Properties to retrieve.
      pov,                      // Overlapped structure.
      NULL,                     // Callback function.
      NULL,                     // No Cursor.
      NULL                      // No transaction
                  );
```

6. Write Windows Event handler (typically a separate thread) and close handle to overlapped structure.

```
if(hr == MQ_INFORMATION_OPERATION_PENDING)
{
    WaitForSingleObject(pov->hEvent, INFINITE);
//
// Parse recieved results
//
}

CloseHandle(pov->hEvent);
```

Windows Event Mechanism Example

The following example opens a queue with receive access, specifies the body of the message as the only message property to retrieve, then uses a Windows Event mechanism to read the first message of the queue.

```
///////////////
// Open Queue
///////////////

HRESULT hr;
QUEUEHANDLE hQueue;

hr = MQOpenQueue(
      szwFormatNameBuffer,      // Format Name of the queue to be opened.
      MQ_RECEIVE_ACCESS,        // Access rights to the Queue.
      0,                        // No receive Exclusive.
      &hQueue                   // OUT: handle to the opened Queue.
                );

if (FAILED(hr))
{
 // Error handler for MQOpenQueue.
}


MQMSGPROPS * pMsgProps;
MQPROPVARIANT *paVariant;
MSGPROPID * paPropId;
DWORD dwPropIdCount = 0;

//
//   The output parameters of an asynchronous call to MQReceiveMessage
//   should be kept intact until the operation completes, you cannot
//   free them or reuse them.
//
pMsgProps = new MQMSGPROPS;
paVariant = new MQPROPVARIANT[ 10];
paPropId = new MSGPROPID[ 10];


////////////////////////////////////////////////
// Prepare the message properties to be retrieved.
////////////////////////////////////////////////
```

```
// Set the PROPID_M_BODY property.
paPropId[dwPropIdCount] = PROPID_M_BODY;                    //PropId
paVariant[dwPropIdCount].vt = VT_VECTOR|VT_UI1;             //Type
paVariant[dwPropIdCount].caub.cElems = MSG_BODY_LEN ;      //Value
paVariant[dwPropIdCount].caub.pElems = new unsigned char[ MSG_BODY_LEN];

 dwPropIdCount++;


////////////////////////////////
// Set the MQMSGPROPS structure
////////////////////////////////

pMsgProps->cProp = dwPropIdCount;       //Number of properties.
pMsgProps->aPropID = paPropId;          //Ids of properties.
pMsgProps->aPropVar = paVariant;        //Values of properties.
pMsgProps->aStatus   = NULL;            //No Error report.


//////////////////////////////////////////
//   Create Event object using overlapped
//   structure.
//////////////////////////////////////////

OVERLAPPED *pov = new OVERLAPPED ;
pov->hEvent = CreateEvent(0, TRUE, TRUE, 0);


//////////////////////////////////////////
//   Retrieve the message using overlapped
//   structure.
//////////////////////////////////////////

hr = MQReceiveMessage(
     hQueue,                        // handle to the Queue.
     5 * 60 * 1000,                 // Max time (msec) to wait.
     MQ_ACTION_RECEIVE,             // Action.
     pMsgProps,                     // properties to retrieve.
     pov,                           // Overlapped structure.
     NULL,                          // No callback function.
     NULL,                          // No Cursor.
     NULL                           // No transaction
                );

if (FAILED(hr))
     {
          //   Error handler for MQReceiveMessage.
     }


//////////////////////////////
// Windows Event handler.
//////////////////////////////
```

```
if (hr == MQ_INFORMATION_OPERATION_PENDING)
      {
           WaitForSingleObject(pov->hEvent, INFINITE);
      //
      // Parse recieved results
      //
      }

CloseHandle(pov->hEvent);
delete paVariant;   //Free resources.
delete paPropId;
```

Reading Messages Using a Cursor

Cursors allow you to read a message that is not at the front of the queue. The cursor always maintains its position relative to the message to which it points.

The two functions used to manage the cursor are **MQCreateCursor** and **MQCloseCursor**. **MQCreateCursor** returns a cursor handle that is used in **MQReceiveMessage**, and **MQCloseCursor** releases the cursor's resources.

To find a specific message

1. Call **MQOpenQueue** to open the queue with either receive or peek access. Opening the queue for receive access allows you to peek at the messages as well as receive them.

```
hr = MQOpenQueue(
      wszFormatNameBuffer,
      MQ_RECEIVE_ACCESS,
      0,
      &hQueue
      );
```

2. Call **MQCreateCursor** to create the cursor.

```
hr = MQCreateCursor(
    hQueue,                 //Queue handle
   &hCursor
   );
```

3. Specify the message properties you want to retrieve.

```
MQMSGPROPS MsgProps;
MQPROPVARIANT aVariant[10];
MSGPROPID aPropId[10];
DWORD PropIdCount = 0;

#define MSG_BODY_LEN 500
unsigned char ucMsgBody[MSG_BODY_LEN];
DWORD dwAppspecificIndex;

// Set the PROPID_M_BODY property.
aPropId[PropIdCount] = PROPID_M_BODY;                  //PropId
aVariant[PropIdCount].vt = VT_VECTOR|VT_UI1;           //Type
aVariant[PropIdCount].caub.cElems = MSG_BODY_LEN;     //Value
aVariant[PropIdCount].caub.pElems = ucMsgBody;
PropIdCount++;

//Set the PROPID_M_APPSPECIFIC property.
aPropId[PropIdCount] = PROPID_M_APPSPECIFIC;          //PropId
aVariant[PropIdCount].vt = VT_UI4;                    //Type
dwAppspecificIndex = PropIdCount;
PropIdCount++;

//Set the MQMSGPROPS structure.
MsgProps.cProp = PropIdCount;          //Number of properties.
MsgProps.aPropID = aPropId;            //Ids of properties.
MsgProps.aPropVar = aVariant;          //Values of properties.
MsgProps.aStatus   = NULL;             //No Error report.
```

4. Call **MQReceiveMessage** until the message is found.

```
DWORD dwAction = MQ_ACTION_PEEK_CURRENT;   //Peek at first msg.
do
 {
 hr = MQReceiveMessage(
      hQueue,                 // handle to the Queue.
      5 * 60 * 1000,         // Max time (msec) to wait for msg.
      dwAction,               // Action.
      &MsgProps,               // properties to retrieve.
      NULL,                   // No overlapped structure.
      NULL,                   // No callback function.
      hCursor,                // Cursor handle.
      NULL                    // No transaction.
 );
 if (FAILED(hr))
 {
 //   Handle failure
 }

 dwAction = MQ_ACTION_PEEK_NEXT;   //Peek at next message.
} while (MsgProps.aPropVar[dwAppspecificIndex].ulVal != 1);
```

5. Call **MQCloseCursor** to release the cursor handle's resources.

```
hr = MQCloseCursor(
   hCursor           //Cursor handle
   );
```

Examples

The following example locates a message whose application-specific property is equal to 1. It uses MQ_PEEK_CURRENT to look at the first message in the queue, then uses MQ_PEEK_NEXT to look at the next message in the queue. MQ_PEEK_NEXT looks at the next message, and then moves the cursor.

```
HRESULT hr;
QUEUEHANDLE hQueue;


////////////////////////////////
// Open queue with receive access.
////////////////////////////////
hr = MQOpenQueue(
      wszFormatNameBuffer,
      MQ_RECEIVE_ACCESS,
      0,
      &hQueue
      );
if (FAILED(hr))
{
//
//   Handle failure
//
}


////////////////////////////////
```

```
// Create the cursor.
////////////////////////////////
HANDLE hCursor;

hr = MQCreateCursor(
    hQueue,                    //Queue handle
   &hCursor
   );
if (FAILED(hr))
{
//
//   Handle failure
//
}


//////////////////////////////////////
// Specify the message properties you
// want to retrieve.
//////////////////////////////////////

MQMSGPROPS MsgProps;
MQPROPVARIANT aVariant[10];
MSGPROPID aPropId[10];
DWORD PropIdCount = 0;

#define MSG_BODY_LEN 500
unsigned char ucMsgBody[MSG_BODY_LEN];
DWORD dwAppspecificIndex;

// Set the PROPID_M_BODY property.
aPropId[PropIdCount] = PROPID_M_BODY;                    //PropId
aVariant[PropIdCount].vt = VT_VECTOR|VT_UI1;            //Type
aVariant[PropIdCount].caub.cElems = MSG_BODY_LEN;     //Value
aVariant[PropIdCount].caub.pElems = ucMsgBody;
PropIdCount++;

//Set the PROPID_M_APPSPECIFIC property.
aPropId[PropIdCount] = PROPID_M_APPSPECIFIC;          //PropId
aVariant[PropIdCount].vt = VT_UI4;                       //Type
dwAppspecificIndex = PropIdCount;                        //Value
PropIdCount++;

//Set the MQMSGPROPS structure.
MsgProps.cProp = PropIdCount;          //Number of properties.
MsgProps.aPropID = aPropId;            //Ids of properties.
MsgProps.aPropVar = aVariant;          //Values of properties.
MsgProps.aStatus   = NULL;             //No Error report.


////////////////////////////////////
// Peek until you find the message
// where APPSPECIFIC = 1.
////////////////////////////////////
```

```
DWORD dwAction = MQ_ACTION_PEEK_CURRENT;   //Peek at first msg.
do
 {
 hr = MQReceiveMessage(
     hQueue,                  // handle to the Queue.
     5 * 60 * 1000,          // Max time (msec) to wait for msg.
     dwAction,                // Action.
     &MsgProps,                // properties to retrieve.
     NULL,                    // No overlapped structure.
     NULL,                    // No callback function.
     hCursor,                 // Cursor handle.
     NULL                      // No transaction.
 );
 if (FAILED(hr))
 {
 //   Handle failure
 break;
 }

 dwAction = MQ_ACTION_PEEK_NEXT;   //Peek at next message.
} while (MsgProps.aPropVar[dwAppspecificIndex].ulVal != 1);

hr = MQCloseCursor(
       hCursor
      );
```

Reading Messages in a Dead Letter Queue

Reading messages in a dead letter queue is typically a three-function operation: a call to **MQGetMachineProperties** to retrieve the local computer identifier (its machine GUID), a call to **MQOpenQueue** to open the queue with receive access, and a call to **MQReceiveMessage** to read the message.

To read a message in a dead letter queue

1. Define an MQQMProps structure that includes PROPID_QM_MACHINE_ID.

```
MQQMPROPS QMProps;
MQPROPVARIANT Variant;
MSGPROPID PropId;
GUID      guidMachineId;

PropId = PROPID_QM_MACHINE_ID;                  //PropId
Variant.vt = VT_CLSID;                          //Type
Variant.puuid = &guidMachineId;                 //Value

QMProps.cProp = 1;                  //Number of properties.
QMProps.aPropID = &PropId;          //Id of property.
QMProps.aPropVar = &Variant;        //Value of property.
QMProps.aStatus   = NULL;           //No Error report.
```

2. Call **MQGetMachineProperties** to retrieve the machine identifier (PROPID_QM_MACHINE_ID) of the local computer.

```
hr = MQGetMachineProperties(
          NULL,
          NULL,
          &QMProps
          );
if (FAILED(hr))
```

3. Translate the machine GUID into a string.

```
WCHAR wszMachineGuid[40];
TBYTE* pszUuid = 0;
if(UuidToString(&guidMachineId, &pszUuid) != RPC_S_OK)
{
  //   Handle failure
}
else
{
     wcscpy( wszMachineGuid, pszUuid );
     RpcStringFree(&pszUuid);
}
```

4. Prepare the format name of the dead letter queue.

```
wsprintf( wszFormatNameBuffer,
            L"MACHINE=%s%s",
            wszMachineGuid,
            L";DEADLETTER"
            );
```

5. Call **MQOpenQueue** and open the queue with receive access.

```
QUEUEHANDLE hQueue;
```

```
hr = MQOpenQueue(
      wszFormatNameBuffer,
      MQ_RECEIVE_ACCESS,
      0,
      &hQueue
      );
if (FAILED(hr))
```

6. Define an MQMSGProps structure for the message properties to be retrieved.

```
MQMSGPROPS MsgProps;
MQPROPVARIANT aVariant[10];
MSGPROPID aPropId[10];
DWORD PropIdCount = 0;

// Prepare property array (PROPVARIANT).
#define MSG_BODY_LEN 500
unsigned char ucMsgBody[MSG_BODY_LEN];
DWORD dwAppspecificIndex;

// Set the PROPID_M_BODY property.
aPropId[PropIdCount] = PROPID_M_BODY;                    //PropId
aVariant[PropIdCount].vt = VT_VECTOR|VT_UI1;             //Type
aVariant[PropIdCount].caub.cElems = MSG_BODY_LEN;        //Value
aVariant[PropIdCount].caub.pElems = ucMsgBody;

PropIdCount++;

// Set the MQMSGPROPS structure
MsgProps.cProp = PropIdCount;          //Number of properties.
MsgProps.aPropID = aPropId;            //Ids of properties.
MsgProps.aPropVar = aVariant;          //Values of properties.
MsgProps.aStatus   = NULL;             //No Error report.
```

7. Call MQReceiveMessage and retrieve the messages in the queue. The following call retrieves the first message in the queue.

```
hr = MQReceiveMessage(
       hQueue,                  // handle to the Queue.
       5 * 60 * 1000,           // Max time (msec) to wait for message.
       MQ_ACTION_RECEIVE,       // Action.
       &MsgProps,               // properties to retrieve.
      NULL,                     // No overlapped structure.
      NULL,                     // No callback function.
      NULL,                     // NO cursor.
      NULL                      // No transaction
      );
if (FAILED(hr))
```

Example

The following example opens a dead letter queue and retrieves the message body of the first message in the queue.

```
  HRESULT hr;
```

```
#define FORMAT_NAME_LEN 80
WCHAR wszFormatNameBuffer[ FORMAT_NAME_LEN];

//////////////////////////////////
//   Define an MQQMPROPS structure
//   for PROPID_QM_MACHINE_ID.
//////////////////////////////////

MQQMPROPS QMProps;
MQPROPVARIANT Variant;
MSGPROPID PropId;
GUID      guidMachineId;

// Set the PROPID_QM_MACHINE_ID property.
PropId = PROPID_QM_MACHINE_ID;                    //PropId
Variant.vt = VT_CLSID;                           //Type
Variant.puuid = &guidMachineId;                   //Value

// Set the MQQMPROPS structure
QMProps.cProp = 1;                    //Number of properties.
QMProps.aPropID = &PropId;            //Id of properties.
QMProps.aPropVar = &Variant;          //Value of properties.
QMProps.aStatus   = NULL;             //No Error report.


//////////////////////////////////////
//   Retrieving the identifier of the
//   local computer (machine GUID).
//////////////////////////////////////
hr = MQGetMachineProperties(
          NULL,
          NULL,
          &QMProps
          );
if (FAILED(hr))
{
     //
     //   Handle failure
     //
}


//////////////////////////////////
//   Translating the machine GUID
//   into a string
//////////////////////////////////
WCHAR wszMachineGuid[40];
WBYTE* pszUuid = 0;
if(UuidToString(&guidMachineId, &pszUuid) != RPC_S_OK)
{
     //
     //   Handle failure
     //
}
else
```

```
{
    wcscpy( wszMachineGuid, pszUuid );
    RpcStringFree(&pszUuid);
}


////////////////////////////////
//   Preparing the format name of
//   the dead letter queue.
////////////////////////////////
wsprintf( wszFormatNameBuffer,
            L"MACHINE=%s%s",
            wszMachineGuid,
            L";DEADLETTER"
            );


////////////////////////////////
//   Open the queue for receive
////////////////////////////////

QUEUEHANDLE hQueue;

hr = MQOpenQueue(
    wszFormatNameBuffer,
    MQ_RECEIVE_ACCESS,
    0,
    &hQueue
    );
if (FAILED(hr))
{
    //
    //   Handle failure
    //
}

////////////////////////////////////
//   Define an MQMSGPROPS structure
//   for the message properties to
//   be retrieved.
////////////////////////////////////

MQMSGPROPS MsgProps;
MQPROPVARIANT aVariant[10];
MSGPROPID aPropId[10];
DWORD PropIdCount = 0;

// Prepare property array (PROPVARIANT).
#define MSG_BODY_LEN          500
unsigned char ucMsgBody[MSG_BODY_LEN];
DWORD dwAppspecificIndex;

// Set the PROPID_M_BODY property.
aPropId[PropIdCount] = PROPID_M_BODY;                    //PropId
aVariant[PropIdCount].vt = VT_VECTOR|VT_UI1;             //Type
```

```
aVariant[PropIdCount].caub.cElems = MSG_BODY_LEN;       //Value
aVariant[PropIdCount].caub.pElems = ucMsgBody;

PropIdCount++;

// Set the MQMSGPROPS structure
MsgProps.cProp = PropIdCount;          //Number of properties.
MsgProps.aPropID = aPropId;            //Ids of properties.
MsgProps.aPropVar = aVariant;          //Values of properties.
MsgProps.aStatus   = NULL;              //No Error report.


////////////////////////////////////
// Read first message in dead
// letter queue.
////////////////////////////////////

hr = MQReceiveMessage(
       hQueue,                    // handle to the Queue.
       5 * 60 * 1000,           // Max time (msec) to wait for message.
       MQ_ACTION_RECEIVE,      // Action.
       &MsgProps,                // properties to retrieve.
     NULL,                       // No overlapped structure.
     NULL,                       // No callback function.
     NULL,                       // NO cursor.
     NULL                        // No transaction
     );
if (FAILED(hr))
{
//
//   Handle failure
//
}
```

Reading Messages in a Machine Journal

To read the messages in a machine journal is typically a three-function operation: a call to **MQGetMachineProperties** to retrieve the local computer identifier (its machine GUID), a call to **MQOpenQueue** to open the queue with receive access, and a call to **MQReceiveMessage** to read the message.

To read messages in a machine journal

1. Prepare the format name of the queue. This includes defining an MQQMProps structure, retrieving the machine identifier, translating the machine identifier into a string, and constructing the format name.
2. Define an MQQMProps structure that includes PROPID_QM_MACHINE_ID.

```
MQQMPROPS QMProps;
MQPROPVARIANT Variant;
MSGPROPID PropId;
GUID      guidMachineId;

PropId = PROPID_QM_MACHINE_ID;                  //PropId
Variant.vt = VT_CLSID;                          //Type
Variant.puuid = &guidMachineId;                 //Value

QMProps.cProp = 1;                  //Number of properties.
QMProps.aPropID = &PropId;          //Id of properties.
QMProps.aPropVar = &Variant;        //Value of properties.
QMProps.aStatus   = NULL;           //No Error report.
```

3. Call **MQGetMachineProperties** to retrieve the machine identifier (PROPID_QM_MACHINE_ID) of the local computer.

```
hr = MQGetMachineProperties(
          NULL,
          NULL,
          &QMProps
          );
if (FAILED(hr))
```

4. Translate the machine GUID into a string.

```
WCHAR wszMachineGuid[40];
TBYTE* pszUuid = 0;
if(UuidToString(&guidMachineId, &pszUuid) != RPC_S_OK)
{
  //   Handle failure
}
else
{
     wcscpy( wszMachineGuid, pszUuid );
     RpcStringFree(&pszUuid);
}
```

5. Construct the format name of the queue.

```
wsprintf( wszFormatNameBuffer,
          L"MACHINE=%s;JOURNAL",
          wszMachineGuid
          );
```

6. Call **MQOpenQueue** and open the queue with receive access.

```
QUEUEHANDLE hQueue;

hr = MQOpenQueue(
      wszFormatNameBuffer,
      MQ_RECEIVE_ACCESS,
      0,
      &hQueue
      );
if (FAILED(hr))
```

7. Define an MQMSGProps structure for the message properties to be retrieved.

```
MQMSGPROPS MsgProps;
MQPROPVARIANT aVariant[10];
MSGPROPID aPropId[10];
DWORD PropIdCount = 0;

// Prepare property array (PROPVARIANT).
#define MSG_BODY_LEN 500
unsigned char ucMsgBody[MSG_BODY_LEN];
DWORD dwAppspecificIndex;

// Set the PROPID_M_BODY property.
aPropId[PropIdCount] = PROPID_M_BODY;                    //PropId
aVariant[PropIdCount].vt = VT_VECTOR|VT_UI1;             //Type
aVariant[PropIdCount].caub.cElems = MSG_BODY_LEN;        //Value
aVariant[PropIdCount].caub.pElems = ucMsgBody;

PropIdCount++;

// Set the MQMSGPROPS structure
MsgProps.cProp = PropIdCount;          //Number of properties.
MsgProps.aPropID = aPropId;            //Ids of properties.
MsgProps.aPropVar = aVariant;          //Values of properties.
MsgProps.aStatus   = NULL;             //No Error report.
```

8. Call MQReceiveMessage and retrieve the messages in the queue. The following call retrieves the first message in the queue.

```
hr = MQReceiveMessage(
       hQueue,                  // handle to the Queue.
       5 * 60 * 1000,           // Max time (msec) to wait for message.
       MQ_ACTION_RECEIVE,       // Action.
       &MsgProps,               // properties to retrieve.
      NULL,                     // No overlapped structure.
      NULL,                     // No callback function.
      NULL,                     // NO cursor.
      NULL                      // No transaction
      );
if (FAILED(hr))
```

Example

The following example opens a machine journal queue and retrieves the message body of the first message in the queue.

```
HRESULT hr;
#define FORMAT_NAME_LEN 80
WCHAR wszFormatNameBuffer[ FORMAT_NAME_LEN];

//////////////////////////////////
//   Define an MQQMPROPS structure
//   for PROPID_QM_MACHINE_ID.
//////////////////////////////////

MQQMPROPS QMProps;
MQPROPVARIANT Variant;
MSGPROPID PropId;
GUID      guidMachineId;

// Set the PROPID_QM_MACHINE_ID property.
PropId = PROPID_QM_MACHINE_ID;                    //PropId
Variant.vt = VT_CLSID;                            //Type
Variant.puuid = &guidMachineId;                   //Value

// Set the MQQMPROPS structure
QMProps.cProp = 1;                    //Number of properties.
QMProps.aPropID = &PropId;            //Id of properties.
QMProps.aPropVar = &Variant;          //Value of properties.
QMProps.aStatus   = NULL;             //No Error report.


//////////////////////////////////////
//   Retrieving the identifier of the
//   local computer (machine GUID).
//////////////////////////////////////
hr = MQGetMachineProperties(
          NULL,
          NULL,
          &QMProps
          );
if (FAILED(hr))
{
     //
     //   Handle failure
     //
}


//////////////////////////////////
//   Translating the machine GUID
//   into a string
//////////////////////////////////
WCHAR wszMachineGuid[40];
TBYTE* pszUuid = 0;
if(UuidToString(&guidMachineId, &pszUuid) != RPC_S_OK)
{
     //
     //   Handle failure
     //
}
```

```
else
{
     wcscpy( wszMachineGuid, pszUuid );
     RpcStringFree(&pszUuid);
}


////////////////////////////////
//   Prepare the format name
//   of the machine journal.
////////////////////////////////
wsprintf( wszFormatNameBuffer,
              L"MACHINE=%s%s",
              wszMachineGuid,
              L";JOURNAL"
              );


////////////////////////////////
// Open the queue for receive.
////////////////////////////////
QUEUEHANDLE hQueue;

hr = MQOpenQueue(
     wszFormatNameBuffer,
     MQ_RECEIVE_ACCESS,
     0,
     &hQueue
     );
if (FAILED(hr))
{
     //
     //   Handle failure
     //
}


////////////////////////////////////
// Define an MQMSGPROPS structure
// for the message properties to
// be retrieved.
////////////////////////////////////

MQMSGPROPS MsgProps;
MQPROPVARIANT aVariant[10];
MSGPROPID aPropId[10];
DWORD PropIdCount = 0;

//Prepare the property array (PROPVARIANT)
#define MSG_BODY_LEN          500
unsigned char ucMsgBody[MSG_BODY_LEN];
DWORD dwAppspecificIndex;

// Set the PROPID_M_BODY property.
aPropId[PropIdCount] = PROPID_M_BODY;                    //PropId
```

```
aVariant[PropIdCount].vt = VT_VECTOR|VT_UI1;         //Type
aVariant[PropIdCount].caub.cElems = MSG_BODY_LEN   //Value
aVariant[PropIdCount].caub.pElems = ucMsgBody;

PropIdCount++;

// Set the MQMSGPROPS structure
MsgProps.cProp = PropIdCount;          //Number of properties.
MsgProps.aPropID = aPropId;            //Ids of properties.
MsgProps.aPropVar = aVariant;          //Values of properties.
MsgProps.aStatus   = NULL;              //No Error report.


////////////////////////////
// Read first message in
// machine journal queue.
////////////////////////////

hr = MQReceiveMessage(
      hQueue,                  // handle to the Queue.
      5 * 60 * 1000,         // Max time (msec) to wait for message.
      MQ_ACTION_RECEIVE,   // Action.
      &MsgProps,              // properties to retrieve.
      NULL,                    // No overlapped structure.
      NULL,                    // No callback function.
      NULL,                    // NO cursor.
      NULL                     // No transaction
      );
if (FAILED(hr))
{
//
//   Handle failure
//
}
```

Reading Messages in a Queue Journal

The functions used to read messages in a queue journal are the same as those used to read messages in other queues. The only difference is that the format name used to open the queue journal has a special format.

To read messages in a queue journal

1. Obtain the queue journal's format name.

```
wsprintf( wszFormatNameBuffer,
          L"%s;JOURNAL",
          QueueFormatName
          );
```

2. Call **MQOpenQueue** to open the queue with receive access.

```
QUEUEHANDLE hQueue;
hr= MQOpenQueue(
     wszFormatNameBuffer,
     MQ_RECEIVE_ACCESS,
     0,
     &hQueue
     );
if (FAILED(hr))
{
 //   Handle failure
}
```

3. Specify the message properties to be retrieved.

```
MQMSGPROPS MsgProps;
MQPROPVARIANT aVariant[10];
MSGPROPID aPropId[10];
DWORD PropIdCount = 0;
#define MSG_BODY_LEN 500
unsigned char ucMsgBody[MSG_BODY_LEN];
DWORD dwAppspecificIndex;

// Set the PROPID_M_BODY property.
aPropId[PropIdCount] = PROPID_M_BODY;                 //PropId
aVariant[PropIdCount].vt = VT_VECTOR|VT_UI1;        //Type
aVariant[PropIdCount].caub.cElems = MSG_BODY_LEN; //Value
aVariant[PropIdCount].caub.pElems = ucMsgBody;
PropIdCount++;

// Set the MQMSGPROPS structure
MsgProps.cProp = PropIdCount;          //Number of properties.
MsgProps.aPropID = aPropId;            //Ids of properties.
MsgProps.aPropVar = aVariant;          //Values of properties.
MsgProps.aStatus   = NULL;              //No Error report.
```

4. Call **MQReceiveMessage** to read the first message in the queue.

```
hr = MQReceiveMessage(
       hQueue,                  // handle to the Queue.
       5 * 60 * 1000,          // Max time (msec) to wait for msg.
       MQ_ACTION_RECEIVE,     // Action.
       &MsgProps,               // properties to retrieve.
```

```
    NULL,                   // No overlapped structure.
    NULL,                   // No callback function.
    NULL,                   // NO cursor.
    NULL                    // No transaction.
  );
```

Example

This example reads the first message in the queue journal. It takes the queue's identifier (GUID), translates it into a string and, prepares the format name of the queue using the translated string, then opens the queue and reads the first message.

```
HRESULT hr;
#define FORMAT_NAME_LEN 80
WCHAR wszFormatNameBuffer[ FORMAT_NAME_LEN];
DWORD dwFormatLen = FORMAT_NAME_LEN;


wsprintf( wszFormatNameBuffer,
        L"%s;JOURNAL",
        QueueFormatName
        );

////////////////////////////////////
// Open queue with receive access.
////////////////////////////////////

QUEUEHANDLE hQueue;
hr= MQOpenQueue(
     wszFormatNameBuffer,
     MQ_RECEIVE_ACCESS,
     0,
     &hQueue
     );
if (FAILED(hr))
{
 //   Handle failure
}


////////////////////////////////////
// Specify the message properties
// you want to receive.
////////////////////////////////////

MQMSGPROPS MsgProps;
MQPROPVARIANT aVariant[10];
MSGPROPID aPropId[10];
DWORD PropIdCount = 0;
#define MSG_BODY_LEN         500
unsigned char ucMsgBody[MSG_BODY_LEN];
DWORD dwAppspecificIndex;

// Set the PROPID_M_BODY property.
aPropId[PropIdCount] = PROPID_M_BODY;                  //PropId
```

```
aVariant[PropIdCount].vt = VT_VECTOR|VT_UI1;         //Type
aVariant[PropIdCount].caub.cElems = MSG_BODY_LEN; //Value
aVariant[PropIdCount].caub.pElems = ucMsgBody;
PropIdCount++;

// Set the MQMSGPROPS structure
MsgProps.cProp = PropIdCount;          //Number of properties.
MsgProps.aPropID = aPropId;            //Ids of properties.
MsgProps.aPropVar = aVariant;          //Values of properties.
MsgProps.aStatus   = NULL;              //No Error report.


//////////////////////////
// Read a message.
//////////////////////////

hr = MQReceiveMessage(
       hQueue,                  // handle to the Queue.
       5 * 60 * 1000,          // Max time (msec) to wait for msg.
       MQ_ACTION_RECEIVE,     // Action.
       &MsgProps,               // properties to retrieve.
       NULL,                   // No overlapped structure.
       NULL,                   // No callback function.
       NULL,                   // NO cursor.
       NULL                     // No transaction.
     );
if (FAILED(hr))
{
 //Handle failure
}
```

Returning an Acknowledgment Message by a Connector Application

Acknowledgment messages returned by the connector application must be sent to the administration queue specified by the original message. Several properties must be set to the acknowledgment message values shown below and others should be set to the values specified by the original message.

The following properties must be set to acknowledgment message values shown below.

Property	Setting
PROPID_M_ACKNOWLEDGE	MQMSG_ACKNOWLEDGMENT_NONE
PROPID_M_AUTH_LEVEL	MQMSG_AUTH_LEVEL_NONE
PROPID_M_BODY	For positive acknowledgments set to NULL. For negative acknowledgments (with the exception of encrypted message bodies), set to the body of the original message.
PROPID_M_CORRELATIONID	Message identifier of original message.
PROPID_M_CLASS	Appropriate acknowledgment class.
PROPID_M_JOURNAL	MQMSG_JOURNAL_NONE
PROPID_M_RESP_QUEUE	Destination queue of the original message.
PROPID_M_TIME_TO_BE_RECEIVED	INFINITE
PROPID_M_TIME_TO_REACH_QUEUE	INFINITE

All the other properties are set to the same values as the original message.

To send an acknowledgment message

1. Open the administration queue specified by the original application.

```
hr = MQOpenQueue(lpstrAdminQueue,MQ_SEND_ACCESS,0, &hQueue);
if (FAILED(hr))
{
// Handle failure
// Return hr;
}
```

2. Set the acknowledgment default value required by MSMQ.

```
aPropID[cProp] = PROPID_M_CLASS;
aPropVar[cProp].vt = VT_UI2;
aPropVar[cProp].uiVal = AckValue;
cProp++;

aPropID[cProp] = PROPID_M_ACKNOWLEDGE;
aPropVar[cProp].vt = VT_UI1;
aPropVar[cProp].bVal = MQMSG_ACKNOWLEDGMENT_NONE;
cProp++;
```

```
aPropID[cProp] = PROPID_M_TIME_TO_BE_RECEIVED;
aPropVar[cProp].vt = VT_UI4;
aPropVar[cProp].ulVal = INFINITE;
cProp++;

aPropID[cProp] = PROPID_M_TIME_TO_REACH_QUEUE;
aPropVar[cProp].vt = VT_UI4;
aPropVar[cProp].ulVal = INFINITE;
cProp++;

aPropID[cProp] = MQMSG_JOURNAL_NONE;
aPropVar[cProp].vt = VT_UI1;
aPropVar[cProp].bVal = pMsgProps->aPropVar[i].bVal;
cProp++;
```

3. Set the remaining message properties. Set PROPID_M_BODY, PROPID_M_CORRELATIONID, PROPID_M_CLASS, and PROPID_M_RESP_QUEUE to the values specified above.
4. Set PROPID_M_CONNECTION_TYPE. This tells the application reading the acknowledgment message that the message was not created by MSMQ.

```
aPropID[cProp] = PROPID_M_CONNECTOR_TYPE;
aPropVar[cProp].vt = VT_CLSID;
aPropVar[cProp].puuid = &g_gConnectorType;
cProp++;
```

5. Call **MQSendMessage** to send the message.

```
hr = MQSendMessage(hQueue, &SendMsgProp, NULL);
```

Example

The following example creates an acknowledgment message and sends it back to the administration queue specified by the original message. It assumes that the format name of the administration queue (*lpstrAdimQueue*), all original message properties (*pMsgProps*), and the value of the acknowledgment (*ackValue*) are all available.

```
HRESULT CreateAck(LPWSTR lpstrAdminQueue,
                  USHORT AckValue,
                  MQMSGPROPS* pMsgProps
                  )

{
MQMSGPROPS SendMsgProp;
MSGPROPID       aPropID[40];
MQPROPVARIANT   aPropVar[40];
HRESULT         aStatus[40];
HRESULT         hr;
HANDLE hQueue = NULL;
DWORD cProp = 0;

//////////////////////////////////
// Open the administration queue
// specified by the original message.
//////////////////////////////////
hr = MQOpenQueue(lpstrAdminQueue,MQ_SEND_ACCESS,0, &hQueue);
if (FAILED(hr))
{
```

```
// Handle failure
// Return hr;
}

DWORD dwExtensionMsgSize = 0;
DWORD dwSenderidLen = 0;
DWORD dwBodySize = 0;
BOOL fEncrypted = FALSE;
DWORD i;


//////////////////////////////////////////////////
// Set the acknowledgment message default values.
//////////////////////////////////////////////////
aPropID[cProp] = PROPID_M_CLASS;
aPropVar[cProp].vt = VT_UI2;
aPropVar[cProp].uiVal = AckValue;
cProp++;

aPropID[cProp] = PROPID_M_ACKNOWLEDGE;
aPropVar[cProp].vt = VT_UI1;
aPropVar[cProp].bVal = MQMSG_ACKNOWLEDGMENT_NONE;
cProp++;

aPropID[cProp] = PROPID_M_TIME_TO_BE_RECEIVED;
aPropVar[cProp].vt = VT_UI4;
aPropVar[cProp].ulVal = INFINITE;
cProp++;

aPropID[cProp] = PROPID_M_TIME_TO_REACH_QUEUE;
aPropVar[cProp].vt = VT_UI4;
aPropVar[cProp].ulVal = INFINITE;
cProp++;

aPropID[cProp] = MQMSG_JOURNAL_NONE;
aPropVar[cProp].vt = VT_UI1;
aPropVar[cProp].bVal = pMsgProps->aPropVar[i].bVal;
cProp++;


////////////////////////////////////////
// Set all other message properties to
// the values of the original message.
////////////////////////////////////////


// Get size and length of properties.

for (i = 0; i < pMsgProps->cProp ; i++)
{
     switch(pMsgProps->aPropID[i])
     {
          case PROPID_M_EXTENSION:
               dwExtensionMsgSize = pMsgProps->aPropVar[i].ulVal;
               break;
```

```
                case PROPID_M_SENDERID_LEN:
                        dwSenderidLen = pMsgProps->aPropVar[i].ulVal;
                        break;

                case PROPID_M_BODY_SIZE:
                        dwBodySize = pMsgProps->aPropVar[i].ulVal;
                        break;

                case PROPID_M_PRIV_LEVEL:
                        fEncrypted = (pMsgProps->aPropVar[i].ulVal ==
MQMSG_PRIV_LEVEL_BODY);
                        break;

                        default:
                        break;
          }
     }

      for (i = 0; i < pMsgProps->cProp ; i++)
      {
      switch (pMsgProps->aPropID[i])
      {

                 // Set correlation identifier
                case PROPID_M_MSGID:
                     aPropID[cProp] = PROPID_M_CORRELATIONID;
                     aPropVar[cProp].vt = VT_UI1|VT_VECTOR;
                     aPropVar[cProp].caub.cElems = pMsgProps->aPropVar[i].caub.cElems;
                     aPropVar[cProp].caub.pElems = pMsgProps->aPropVar[i].caub.pElems;
                     cProp++;
                     break;

                // Set message priority.
                case PROPID_M_PRIORITY:
                     aPropID[cProp] = PROPID_M_PRIORITY;
                     aPropVar[cProp].vt = VT_UI1;
                     aPropVar[cProp].bVal = pMsgProps->aPropVar[i].bVal;
                     cProp++;
                     break;

                // Set delivery mode.
                case PROPID_M_DELIVERY:
                     aPropID[cProp] = PROPID_M_DELIVERY;
                     aPropVar[cProp].vt = VT_UI1;
                     aPropVar[cProp].bVal = pMsgProps->aPropVar[i].bVal;
                     cProp++;
                     break;

                // Set application specific information
               case PROPID_M_APPSPECIFIC:
                     aPropID[cProp] = PROPID_M_APPSPECIFIC;
                     aPropVar[cProp].vt = VT_UI4;
                     aPropVar[cProp].ulVal = pMsgProps->aPropVar[i].ulVal;
                     cProp++;
```

```
            break;

        // Set message label to the same
        case PROPID_M_LABEL:
            aPropID[cProp] = PROPID_M_LABEL;
            aPropVar[cProp].vt = VT_LPWSTR;
            aPropVar[cProp].pwszVal = pMsgProps->aPropVar[i].pwszVal;
            cProp++;
            break;
        //

        // Set extension information.
       case PROPID_M_EXTENSION:
            aPropID[cProp] = PROPID_M_EXTENSION;
            aPropVar[cProp].vt = VT_UI1|VT_VECTOR;
            aPropVar[cProp].caub.cElems = dwExtensionMsgSize;
            aPropVar[cProp].caub.pElems = pMsgProps->aPropVar[i].caub.pElems;
            cProp++;
            break;

        // Set acknowledge message response queue to
        // the destination queue of the original message.
       case PROPID_M_DEST_QUEUE:
            aPropID[cProp] = PROPID_M_RESP_QUEUE;
            aPropVar[cProp].vt = VT_LPWSTR;
            aPropVar[cProp].pwszVal = pMsgProps->aPropVar[i].pwszVal;
            cProp++;
            break;

        ////////////////////////////////////////////
        // Set message body. If acknowledge is negative
        // and the original message isn't encrypted, add
        // message body of original message.
        ////////////////////////////////////////////
        case PROPID_M_BODY:
            if (MQCLASS_NACK(AckValue) && ! fEncrypted)
            {
                aPropID[cProp] = PROPID_M_BODY;
                aPropVar[cProp].vt = VT_UI1|VT_VECTOR;
                aPropVar[cProp].caub.cElems = dwBodySize;
                aPropVar[cProp].caub.pElems = pMsgProps->aPropVar[i].caub.pElems;
                cProp++;
            }
            break;

        default:
            break;
    }
}


////////////////////////////////////////////////
// Set the connector type identifier (GUID) to
// indicate who generated the acknowledge message.
////////////////////////////////////////////////
```

```
    aPropID[cProp] = PROPID_M_CONNECTOR_TYPE;
    aPropVar[cProp].vt = VT_CLSID;
    aPropVar[cProp].puuid = &g_gConnectorType;
    cProp++;

    SendMsgProp.aStatus = aStatus;
    SendMsgProp.aPropID = aPropID;
    SendMsgProp.aPropVar = aPropVar;
    SendMsgProp.cProp =cProp;


    ////////////////////////////////////////////////
    // Send the acknowledge message back to the
    // administration queue specified by the
    // original message.
    ////////////////////////////////////////////////
    hr = MQSendMessage(hQueue, &SendMsgProp, NULL);

    MQCloseQueue(hQueue);

    return hr;
}
```

Retrieving a Queue's Properties Using API Functions

After a queue is created, its properties can be retrieved at any time by calling **MQGetQueueProperties**. All queue properties can be retrieved; however, you can only retrieve the properties of private queues if they are located on your local computer.

Note Properties of public queues can also be retrieved by doing a query on the MQIS. For details, see Locating a Public Queue.

In most cases, any application can retrieve a queue's properties. However, if MQ_ERROR_ACCESS_DENIED is returned to the **MQGetQueueProperties** call, the queue's access control is blocking the application from retrieving its properties. For information setting a queue's access rights, see Setting Access Control Security for a Queue.

To retrieve a queue's properties

1. Obtain the format name of the queue. If the format name of the queue is not known, you can obtain its format name by using one of the following format name translation functions:

   MQHandleToFormatName

   MQInstanceToFormatName

   MQPathNameToFormatName.
2. Specify the properties you want to retrieve.
3. Call **MQGetQueueProperties**.

For more information about setting properties, see Setting a Queue's Properties Using API Functions.

Queue Properties

All properties of a queue can be retrieved. However, you can only retrieve the properties of a private queue if it is located on your local machine.

PROPID_Q_AUTHENTICATE

PROPID_Q_BASEPRIORITY (public queues only)

PROPID_Q_CREATE_TIME

PROPID_Q_INSTANCE (public queues only)

PROPID_Q_JOURNAL

PROPID_Q_JOURNAL_QUOTA

PROPID_Q_LABEL

PROPID_Q_MODIFY_TIME

PROPID_Q_PATHNAME

PROPID_Q_PRIV_LEVEL

PROPID_Q_QUOTA

PROPID_Q_TRANSACTION

PROPID_Q_TYPE

Setting a Queue's Properties Using API Functions

The properties of a queue can be dynamically set by calling **MQSetQueueProperties**. All open instances of the queue are immediately affected when **MQSetQueueProperties** is called.

Note In most cases, any application can set a queue's properties. However, if MQ_ERROR_ACCESS_DENIED is returned to the **MQSetQueueProperties** call, the queue's access control is blocking the application from setting its properties. For information about access rights, see Access Control.

Not all properties can be set by **MQSetQueueProperties**. The following tables indicates which properties can be set and which cannot.

Property	Set by MQSetQueueProperties
PROPID_Q_AUTHENTICATE	Yes
PROPID_Q_BASEPRIORITY	Yes
PROPID_Q_CREATE_TIME	No (set by MSMQ when queue created)
PROPID_Q_INSTANCE	No (set by MSMQ when public queue is created)
PROPID_Q_JOURNAL	Yes
PROPID_Q_JOURNAL_QUOTA	Yes
PROPID_Q_LABEL	Yes
PROPID_Q_MODIFY_TIME	No (set by MSMQ)
PROPID_Q_PATHNAME	No (set when queue is created)
PROPID_Q_PRIV_LEVEL	Yes
PROPID_Q_QUOTA	Yes
PROPID_Q_TRANSACTION	No (set when queue is created)
PROPID_Q_TYPE	Yes

To set a queue's properties

1. Determine the format name of the queue. If the format name of the queue is not known, you can obtain its format name by using one of the following format name translation functions:
   MQHandleToFormatName,
   MQInstanceToFormatName,
   MQPathNameToFormatName.
2. Specify the properties you want to retrieve.
3. Call **MQSetQueueProperties**.

Authenticating Messages Using API Functions

MSMQ provides security services that allow applications to authenticate messages using an internal or external certificate. Using an internal certificate allows an application to verify the security identifier (SID) of the user sending the message but nothing more. Using an external certificate allows an application to verify the sender's SID, plus other user information provided in the certificate.

For information on what MSMQ does to authenticate messages, see How MSMQ Authenticates Messages.

For information on using an internal certificate, see Authenticating Messages Using an Internal Certificate.

For information on using an external certificate, see Authenticating Messages Using an External Certificate.

Authenticating Messages Using an Internal Certificate

From an application perspective, authenticating messages using an internal certificate requires registering the internal certificate with MSMQ, and setting the appropriate message properties. An internal certificate is created the first time the MSMQ Control Panel utility is run.

The following procedures highlight what must be done by the sending computer to request authentication using an internal certificate, and what the receiving application can do to determine if MSMQ was able to authenticate the message.

To request authentication using an internal certificate

1. Register the internal certificate using the MSMQ Control Panel option.
2. Set PROPID_M_AUTH_LEVEL to MQMSG_AUTH_LEVEL_ALWAYS in the message properties.
3. Make sure PROPID_M_SENDER_CERT is not specified.
4. If you want to change the hash algorithm MSMQ uses to authenticate the message, set PROPID_M_HASH_ALG (the default algorithm is CALG_MD5).
5. If you want MSMQ to return an acknowledgment to show that the message reached the queue or was retreived, set PROPID_M_ACKNOWLEDGE to MQMSG_ACKNOWLEDGMENT_FULL_REACH_QUEUE | MQMSG_ACKNOWLEDGMENT_NACK_REACH_QUEUE | MQMSG_ACKNOWLEDGMENT_FULL_RECEIVE | MQMSG_ACKNOWLEDGMENT_NACK_RECEIVE.
6. Send the message.

After the message is sent, the remaining work is done by MSMQ. For information on what MSMQ does to authenticate messages, see How MSMQ Authenticates Messages.

To receive an authenticated message

When reading the message in the queue, verify that PROPID_M_AUTHENTICATED is set to 1. If it is set to 0, the message was not signed and it is up to the receiving application to decide if it wants to use the message.

When a message is authenticated (PROPID_M_AUTHENTICATED = 1) using an internal certificate, MSMQ guarantees that the sender identifier in PROPID_M_SENDERID is correct, and that no one tampered with the message.

Authenticating Messages Using an External Certificate

From an application perspective, authenticating messages using an external certificate is very easy. It simply requires getting the external certificate from a certificate authority, retrieving information from the certificate, and setting the appropriate message properties.

To send an authenticated message

1. Obtain a certificate from an authorized certificate authority. A common way to obtain a certificate is to request a class 1 certificate from VeriSign Commercial Software Publishers CA, using Microsoft® Internet Explorer (version 3.0 or later).
2. Place the certificate in the Microsoft Internet Explorer personal certificate store (if Internet Explorer was used to obtain the certificate, this is done automatically). MSMQ can only use certificates placed in this store.
3. If you want to use a sender identifier in addition to the certificate information, register the certificate using the MSMQ Control Panel. This step is not required to authenticate the message.
4. Set PROPID_M_AUTH_LEVEL to MQMSG_AUTH_LEVEL_ALWAYS.
5. If the certificate is only going to be used once, set PROPID_M_SENDER_CERT. If the same certificate is going to be used several times, call **MQGetSecurityContext** to retrieve security information from the certificate and set PROPID_M_SECURITY_CONTEXT.
6. If you want to change the hash algorithm MSMQ uses to authenticate the message, set PROPID_M_HASH_ALG (the default algorithm is CALG_MD5).
7. If you want MSMQ to return an acknowledgment to show that the message reached the queue or was retreived, set PROPID_M_ACKNOWLEDGE to MQMSG_ACKNOWLEDGMENT_FULL_REACH_QUEUE | MQMSG_ACKNOWLEDGMENT_NACK_REACH_QUEUE | MQMSG_ACKNOWLEDGMENT_FULL_RECEIVE | MQMSG_ACKNOWLEDGMENT_NACK_RECEIVE
8. Send the message.

After the message is sent, the remaining work is done by MSMQ. For information on what MSMQ does to authenticate the message, see How MSMQ Authenticates Messages.

To receive an authenticated message

- When reading the message in the queue, verify that PROPID_M_AUTHENTICATED is set to 1. If it is set to 0, the message was not signed and it is up to the receiving application to decide whether to use the message.

When a message is authenticated (PROPID_M_AUTHENTICATED = 1) using an external certificate, MSMQ guarantees that the owner of the certificate (as specified in the certificate) sent the message, that no one tampered with the message, and that the SID is correct if the SID (PROPID_M_SENDERID) was passed with the message.

Note MSMQ does not validate the external certificate. The receiving application must perform any validation requirements on the certificate before using an authenticated message. MSMQ generates the digital signature of a message when it is sent and verifies the digital signature when the message is received, but does not validate the certificate itself.

Sending Messages While Impersonating Another User

Sending messages while impersonating another user requires that the security information must be retrieved from the certificate using **MQGetSecurityContext**, even if you are using the internal certificate supplied by MSMQ.

To send an authenticated message

1. Call **MQGetSecurityContext**. When using an external certificate, specify the certificate you want to use. When using the internal certificate provided by MSMQ, specify NULL. **MQGetSecurityContext** must always be used when impersonating a user.
2. Set PROPID_M_SECURITY_CONTEXT to the returned handle.

After the message is sent, the remaining work is done by MSMQ. For information on what MSMQ does to authenticate the message, see How MSMQ Authenticates Messages.

Setting Access Control Security for a Queue

The access control of a queue is first set when the queue is created (see Creating a Queue). However, MSMQ also provides two functions for managing access control after the queue is created: **MQGetQueueSecurity** and **MQSetQueueSecurity**.

Access control determines who can perform specific operations on the queue. Operations that can be set include creating, deleting, and opening the queue; sending messages to and reading messages from the queue; getting and setting the queue's properties; and getting and setting the queue's security descriptor. These operations can be restricted to a specific user or group of users.

For information on how MSMQ uses these settings, see Access Control.

Using Transactions

When an MSMQ application performs a transaction, it must work with all the transactional services, such as Microsoft® Distributed Transaction Coordinator (MS DTC) and all resource managers needed to complete the transaction. This includes all the resource mangers associated with the transaction, including MSMQ and MS DTC as the transaction manager.

From a system perspective, the process for performing a transaction starts with the transaction application asking MS DTC or MSMQ for a new transaction object. Once a transaction object is available, the application can then make any number of transactional calls, as well as any number of non-transactional calls, to all the resource managers needed by the application.

The application must check the return values of all the functions called. If all calls succeed, the application can vote to commit the transaction. This does not mean the transaction is committed; it only means that the application is ready to commit.

MS DTC now starts a two-phase commit procedure, asking each participating resource manager to prepare and to inform MS DTC if it is ready to commit. If all the resource managers commit, MS DTC commits the transaction. If any one resource manager does not commit, the transaction is aborted.

All this activity by MS DTC is invisible to the transaction application. The application only sees the return value of the **Commit** function it calls. When a successful **Commit** is returned to the application, the transaction is completed.

Note When a transaction is completed, it does not mean the work is completed. When the transaction manager commits the transaction, it only means that each resource manager guarantees that it will do its part of the transaction at some later time.

Programming Transactions

The following table lists the elements of MSMQ that require special attention when used as part of a transaction:

Element	Description
MQCreateQueue:	Creates a transaction queue (see PROPID_Q_TRANSACTION) .
MQSendMessage:	Sends transactional messages (see *pTransaction* parameter) to a transaction queue.
MQReceiveMessage	Retrieves transactional messages (see *pTransaction* parameter) from a transaction queue.
PROPID_M_PRIORITY	MSMQ sets the message priority of all transactional messages to 0.
PROPID_M_DELIVERY	MSMQ sets the delivery property of all transaction messages to MQMSG_DELIVERY_RECOVERABLE.
PROPID_M_TIME_TO_BE_RECEIVED	MSMQ sets the message's time-to-be-received property to the setting of the first transaction message sent.
PROPID_M_TIME_TO_REACH_QUEUE	MSMQ sets the message's time-to-reach-queue property to the setting of the first transaction message sent.
PROPID_M_JOURNAL	If set to MQMSG_DEADLETTER, MSMQ automatically sends the transactional messages to the transaction dead letter queue (DEADXACT) on the source machine if the message is not delivered.

Transaction Programming Considerations

The following issues are unique to MSMQ transactions:

- There are five ways to send and receive transactional messages: MTS transactions, MS DTC external transactions, MSMQ internal transactions, XA-compliant Transactions, and single-message transactions.
- When sending messages, the application must get a successful return code from the call to **MQSendMessage** and a successful return code from its commit call before it can be assured that the message will be sent.
- If some operations in a transaction fail, it is the application's responsibility to decide whether to terminate the entire transaction (by calling the transaction object's abort member function) or commit the transaction anyway (if the failures are such that the transaction is still viable). If the application does commit to a transaction where some operations have failed, the failed operations will not be part of the transaction.
- There is no limit to the number of messages sent, the number of messages retrieved, or the number of queues used in a single transaction. However, an application cannot send a message to a queue and then try to retrieve it during the same transaction.
- Messages can be sent to a local or remote transaction queue, but messages can only be received from a local transaction queue.
- Calling **MQSendMessage** does not actually send the message within the transaction. The actual sending is done at some time after MS DTC commits the transaction. When MS DTC returns a successful commit return value, the sending application is guaranteed that the message will be sent. If a transaction is aborted, all MSMQ transaction operations are rolled back: no messages are sent, and all retrieved messages are returned to their original place in the queue.
- MSMQ guarantees exactly-once-delivery. This means that all messages sent to a queue will arrive once and only once. MSMQ takes special measures to prevent any message duplication or loss.
- MSMQ guarantees that all messages sent to a specific transaction queue will arrive in the order they were sent by the transaction. This means that if transaction T1 sends messages M1 and M2 to queue Q1, M1 will arrive before M2.

  However, there is no guarantee if two transactions are sending messages to the same queue. If transaction T1 sends messages M1 and M2 to Q1, and a second transaction T2 sends messages M3 and M4 to Q1, MSMQ only guarantees that M1 will arrive before M2, and that M3 will arrive before M4. In order to guarantee that M1 and M2 will arrive before M3 and M4, the application must commit to T2 only after getting a successful return code from T1.

  MSMQ does not guarantee order of delivery to different queues, nor does it guarantee order of delivery from different computers.
- One may receive messages from a transaction queue using non-transactional receive operations.

Using the ActiveX Components

This section contains complete examples of the basic tasks your application may need to perform.

Note For information on a specific MSMQ ActiveX component method, property, or event, refer to its reference page in the "MSMQ Reference."

The following tasks are described:

- Creating a Queue
- Locating a Public Queue
- Opening a Queue
- Sending Messages To a Queue
- Sending Messages that Request Acknowledgments
- Sending Messages that Request a Response
- Sending Private Messages
- Sending Messages Using an Internal Transaction
- Sending Messages Using an Internal Transaction
- Sending Messages Using an MS DTC External Transaction
- Reading Messages In a Queue
- Retrieving a Queue's Properties Using ActiveX Components
- Setting a Queue's Properties Using ActiveX Components

Creating a Queue

All queues, either public or private, are created by calling the **MSMQQueueInfo** object's **Create** method. For a description of public and private queues, see Message Queues.

The only queue property required to create the queue is **PathName**. This property tells 1) MSMQ where to store the queue's messages, 2) whether the queue is public or private, and 3) the name of the queue. Once the queue is created, the **MSMQQueueInfo** object's returned **FormatName** property is used to open the queue. For a description of MSMQ pathnames and queue format names, see Referencing a Queue.

To create a queue

1. Determine which computer will hold the messages for the queue. The computer's machine name is part of the queue's MSMQ pathname (**PathName**). For private queues, the local computer must be specified.
2. Determine whether the queue should be public or private. This tells MSMQ where to register the queue: public queues are registered in MQIS and private queues are registered on the local machine (private queues can only be registered on the local machine). This information is part of the queue's MSMQ pathname (**PathName**).
3. Determine the name for the queue. The queue's name is part of the queue's MSMQ pathname (**PathName**).

   Note The MSMQ pathname must be unique in the MSMQ enterprise. This applies to public and private queues.

4. Determine which queue properties must be set. If a queue property is not specified before calling **Create**, its default value is used. For a complete list of the queue properties that can be set when a queue is created, see the following Queue Properties section.
5. Set the queue properties. The only property required to create a queue is **PathName**, all other queue properties are optional. The **PathName** property specifies where the queue's messages are stored, if the queue is public or private, and the local name of the queue. To modify properties after the queue is created, see **Update**.

   ```
   qinfo.PathName = "machinename\localname"
   or
   qinfo.PathName = "machinename\PRIVATE$\localname"
   ```

   Where *qinfo* is an MSMQQueueInfo object.
6. Call **Create**.

   ```
   qinfo.Create
   ```

Examples

The following two examples show functions used to create a public queue and a private queue. In these examples, two queue properties are specified: **PathName** and **Label**.

Note In these examples, a "." is used to indicate the local machine in **PathName**. For MSMQ servers and Independent clients, the local machine is the local computer. However, for MSMQ dependent clients the local machine is the client's MSMQ server.

For a public queue

```
Function CreatePublicQueue() As MSMQQueueInfo
    Dim qinfo As New MSMQQueueInfo
    qinfo.PathName = ".\MyQueue" 'Created on local computer.
    qinfo.Label = "Public Queue"
```

```
    On Error GoTo ErrorHandler
    qinfo.Create
    Set CreatePublicQueue = qinfo
    Exit Function
ErrorHandler:
    MsgBox "Couldn't create queue. Error: " + Str$(Err.Number)
    MsgBox "Reason:   " + Err.Description
    Set CreatePublicQueue = Nothing
End Function

Sub Test()
     Dim qinfo As MSMQQueueInfo
     Set qinfo = CreatePublicQueue
     If Not qinfo Is Nothing Then
        MsgBox "Queue's format name is: " = qinfo.FormatName
     End If
End Sub
```

For a private queue

```
Function CreatePrivateQueue() As MSMQQueueInfo
    Dim qinfo As New MSMQQueueInfo
    qinfo.PathName = ".\Private$\MyQueue"           'On local computer.
    qinfo.Label = "Private Queue"
    On Error GoTo ErrorHandler
    qinfo.Create
    Set CreatePrivateQueue = qinfo
    Exit Function
ErrorHandler:
    MsgBox "Couldn't create queue. Error: " + Str$(Err.Number)
    MsgBox "Reason:   " + Err.Description
    Set CreatePrivateQueue = Nothing
End Function

Sub Test()
     Dim qinfo As MSMQQueueInfo
     Set qinfo = CreatePrivateQueue
     If Not qinfo Is Nothing Then
        MsgBox "Queue's format name is: " = qinfo.FormatName
     End If
End Sub
```

Queue Properties

The following optional queue properties can be set by the application when creating the queue:

Authenticate

BasePriority

Journal

JournalQuota

Label

PrivLevel

Quota

ServiceTypeGuid

The following properties are set by MSMQ when it creates the queue. To read these properties, the application must explicitly call the **MSMQQueueInfo** object's **Refresh** method before they can be read.

CreateTime (public queues only)

FormatName (public and private queues)

IsTransactional (public and private queues)

ModifyTime (public queues only)

QueueGuid (public queues only)

The following properties are set by MSMQ when it creates the queue. To read these properties, the application must explicitly call the **MSMQQueueInfo** object's **Refresh** method before they can be read.

Locating a Public Queue

Public queues can be located by running a query on the queues registered in MQIS. A query can be based on the queue's identifier, its label, the type of service it provides, when it was created, or the last time the queue's properties were modified.

A query is made by calling the **MSMQQuery** object's **LookupQueue** method. When the query is finished, the returned **MSMQQueueInfos** object references all the queues located by the query.

To run a query

1. Determine the search criteria for the query. When locating queues by label, type of service, create time, or modify time you can also specify a relationship parameter. For example, when using the queue's create time as the search criteria, you can also use the create time's relationship parameter (*RelCreateTime*) to locate all the queues that were created before, after, or on a specific date.
2. Call **LookupQueue**.

   ```
   Set qinfos = query.LookupQueue(Label:="Test Queue")
   ```

3. Call **Reset** on the returned **MSMQQueueInfos** object. Although this is not required, it guarantees that the cursor is looking at the first queue in the returned set.

   ```
   qinfos.Reset
   ```

4. Look at the queues in the query. In the example below, this is done by calling **Next** to point to the first queue in the query, followed by a common While loop containing another call to **Next**.

   ```
   Set qinfo = qinfos.Next

   While Not qinfo Is Nothing
       MsgBox "I found a Test Queue! its Format name is: " + qinfo.FormatName
       Set qinfo = qinfos.Next
   Wend
   ```

Example

This example assumes that at least one queue whose label is "Test Queue" already exists. It runs a query for the test queues, displaying the format name of each queue it finds.

To try this example using Microsoft® Visual Basic® (version 5.0), paste the code into the Declaration section of a form, and then run the example and click the form.

```
Dim query As New MSMQQuery
Dim qinfos As MSMQQueueInfos
Dim qinfo As MSMQQueueInfo
Dim Response As String

Private Sub Form_Click()

    Set qinfos = query.LookupQueue(Label:="Test Queue")
    qinfos.Reset
    Set qinfo = qinfos.Next

    While Not qinfo Is Nothing
        MsgBox "I found a Test Queue! its Format name is: " + qinfo.FormatName
        Set qinfo = qinfos.Next
    Wend

End Sub
```

Opening a Queue

Queues can be opened by calling the **MSMQQueueInfo** object's **Open** method. The **Open** method returns an **MSMQQueue** object that can be used for:

- Sending messages to the queue (**Send**).
- Enabling notification for reading messages asynchronously (**EnableNotification**)
- Reading messages in the queue (**Peek**, **PeekNext**, **Receive**)
- Closing the queue (**Close**)

Note The **MSMQQueue** object exposes a queue handle that can be used to call MSMQ API functions directly. For example, in Microsoft® Visual Basic®, MSMQ functions can be called directly using the Declare Function facility.

When opening a queue, the application specifies the access rights and share mode of the queue. The queue's access rights indicate if the application is going to send messages to the queue, peek at the messages in the queue, or retrieve messages from the queue. The queue's share mode indicates who else can use the queue while the application is using the queue.

In most cases, a queue can be opened without checking its access rights. However, if MQ_ERROR_ACCESS_DENIED is returned to the **Open** call, the queue's access control is blocking the application from opening the queue. A queue's access control can block sending messages, retrieving messages, or peeking at messages. For information about access rights, see Access Control.

The properties of the opened queue are based on the current properties of the **MSMQQueueInfo** object used to open the queue. While the queue is opened, the application can always see the current properties of the queue by calling the **MSMQQueue** object's **queueInfo** property.

To open a queue

1. Determine the queue's access mode. Are messages going to be sent to the queue (*lAccess* = MQ_SEND_ACCESS), retrieved from the queue(*lAccess* = MQ_RECEIVE_ACCESS), or peeked at without removing them from the queue (*lAccess* = MQ_PEEK_ACCESS)?

   When a queue is opened with receive access, the application can also peek at the queue's messages. However, the reverse is not true. When a queue is opened with peek access, the application cannot retrieve a message from the queue.
2. Determine the queue's share mode. If messages are going to be retrieved from the queue (*lAccess* = MQ_RECEIVE_ACCESS), determine if the application should not allow others to retrieve messages at the same time it is retrieving messages(*ShareMode* = MQ_DENY_RECEIVE_SHARE). Using this setting does not stop other applications from peeking at the messages in the queue, it only prevents them from retrieving messages at the same time the calling application is retrieving messages.
3. Verify that the queue information (**MSMQQueueInfo**) object exists and that a queue object (**MSMQQueue**) is available.
4. Call **Open**, setting the queue's access mode and share mode to the appropriate value.

Example: Opening a queue for sending messages

This example creates a public queue, then opens the queue for sending messages. To try this example using Microsoft® Visual Basic® (version 5.0), paste the code into the Declaration section of a form, and then run the example and click the form.

```
Dim qinfo As New MSMQQueueInfo
Dim q As New MSMQQueue
```

```
Private Sub Form_Click()

  Set qinfo = New MSMQQueueInfo
  qinfo.PathName = ".\SendTest"
  qinfo.Label = "Test Queue"
  qinfo.Create

  Set q = qinfo.Open(MQ_SEND_ACCESS, MQ_DENY_NONE)
  If q.IsOpen Then
     MsgBox "The queue is open for sending messages."
     Else
        MsgBox "The queue is not open!"
  End If

End Sub
```

Example: Opening a queue for reading messages

This example creates a public queue, then opens the queue for retrieving messages. To try this example using Microsoft Visual Basic (version 5.0), paste the code into the Declaration section of a form, and then run the example and click the form.

```
Dim qinfo As New MSMQQueueInfo
Dim q As New MSMQQueue

Private Sub Form_Click()

  Set qinfo = New MSMQQueueInfo
  qinfo.PathName = ".\ReceiveTest"
  qinfo.Label = "Test Queue"
  qinfo.Create

  Set q = qinfo.Open(MQ_RECEIVE_ACCESS, MQ_DENY_NONE)
  If q.IsOpen Then
     MsgBox "The queue is open to receive messages."
     Else
        MsgBox "The queue is not open!"
  End If

End Sub
```

Sending Messages to a Queue

Sending messages is an asynchronous operation that requires opening the queue and sending the message.

The body of an MSMQ message can be a string, an array of bytes, or any persistent ActiveX object that supports **IDispatch** and **IPersist** (**IPersistStream** or **IPersistStorage**).

To transmit an object, the sending application specifies the object in the message's **Body** property. After the message arrives, the receiving application can deserialize the object using the message's **Body** property.

For examples of sending messages using ActiveX components, see:

- Sending Messages that Request Acknowledgments
- Sending Messages that Request a Response
- Sending Private Messages

The following is a complete list of message properties that can be set when sending a message.

Message Properties

Property	Description
Ack	Determines the type of acknowledgment messages (default is none) MSMQ will generate and send back to the administration queue.
AdminQueueInfo	Provides the **MSMQQueueInfo** object that MSMQ uses for sending acknowledgment messages.
CorrelationId	Determines the application-defined identifier for the message.
Delivery	Determines how MSMQ delivers the message: express or guaranteed delivery.
EncryptAlgorithm	Determines which encryption algorithm is used when sending private messages.
Label	Provides an application-defined label for the message.
MaxTimeToReachQueue	Determines how long the message has to reach the queue.
MaxTimeToReceive	Determines how long before the message must be removed from the queue.
Priority	Determines the message's priority: effects routing and placement in the queue.
PrivLevel	Determines if the message is sent as a private (encrypted) message.
ResponseQueueInfo	Provides the **MSMQQueueInfo** object used for returning response messages.

Sending Messages that Request Acknowledgments

Acknowledgment messages are returned by MSMQ whenever the sending application requests them. To receive acknowledgment messages, the sending application must request the type of acknowledgment messages it wants returned (**Ack**), and an administration queue where MSMQ can send the acknowledgment messages (**AdminQueueInfo**).

The administration queue is maintained by the sending application. It is the sending application's responsibility to create the queue, to read the messages in the queue, and to perform whatever actions are required as a result of the type of acknowledgment message returned by MSMQ.

When reading the messages in the administration queue, the application can check the message's **Class** property to determine what type of acknowledgment was returned. Not all acknowledgment messages contain the same information. For example, although negative acknowledgment messages contain the message body of the original messages, positive acknowledgment messages do not. For a complete description of what is in the various types of acknowledgments, see Acknowledgment Messages.

To send messages that request acknowledgments

1. Determine what type of acknowledgment messages need to be returned and which queue will be used as the administration queue.
2. Locate the administration queue, creating one if it doesn't exist.

```
Set qinfos = query.LookupQueue(Label:="Administration Queue")
qinfos.Reset
Set qinfoAdmin = qinfos.Next
If qinfoAdmin Is Nothing Then
    Set qinfoAdmin = New MSMQQueueInfo
    qinfoAdmin.PathName = ".\AdminQueue"
    qinfoAdmin.Label = "Administration Queue"
    qinfoAdmin.Create
End If
```

3. Locate the destination queue, creating one if it doesn't exist.

```
Set qinfos = query.LookupQueue(Label:="Destination Queue")
qinfos.Reset
Set qinfoDest = qinfos.Next
If qinfoDest Is Nothing Then
    Set qinfoDest = New MSMQQueueInfo
    qinfoDest.PathName = ".\DestQueue"
    qinfoDest.Label = "Destination Queue"
    qinfoDest.Create
End If
```

4. Open the destination queue and send the message.

```
Set qDest = qinfoDest.Open(MQ_SEND_ACCESS, MQ_DENY_NONE)
msgSent.Label = "Test Message"
msgSent.Body = "This message tests acknowledgment messages."
msgSent.Ack = MQMSG_ACKNOWLEDGMENT_FULL_RECEIVE
Set msgSent.AdminQueueInfo = qinfoAdmin
msgSent.Send qDest

MsgBox "The message was sent. Check the MSMQ Explorer to see the messages in the
queue."

qDest.Close
```

5. Open the administration queue and read the acknowledgment messages returned by MSMQ. Messages are placed in the administration queue by their priority. The priority level of an acknowledgment message is set to the priority level of its original message.

```
Set qAdmin = qinfoAdmin.Open(MQ_RECEIVE_ACCESS, MQ_DENY_NONE)
Set msgAdmin = qAdmin.Receive
```

6. Perform whatever actions are needed as a result of the returned acknowledgment message. The acknowledgment message's **Class** property specifies the type of acknowledgment returned.

```
If msgAdmin.Class = MQMSG_CLASS_ACK_RECEIVE Then
        MsgBox "The message was removed from the queue."
ElseIF msgAdmin.Class = MQMSG_CLASS_NACK_RECEIVE_TIMEOUT Then
        MsgBox "The message was not removed from the queue in time."
Else
        MsgBox "The returned acknowledgment message (" + CStr(msgAdmin.Class) + ")
is not valid for this example."
End If
```

Example

This example uses an administration queue to see if a message has been retrieved from its destination queue. First a message is sent to its destination queue with its *time-to-live-timer* (**MaxTimeToReceive**) set to 60 seconds. Then the application reads the acknowledgment message returned to the administration queue to see if the original message was retrieved from the queue. The destination and administration queues are created if they don't exist.

To try this example using Microsoft® Visual Basic® (version 5.0), paste the code into the code window of a form, and then run the example and click the form.

```
Dim query As New MSMQQuery
Dim qinfoDest As MSMQQueueInfo
Dim qinfoAdmin As MSMQQueueInfo
Dim qDest As MSMQQueue
Dim qAdmin As MSMQQueue
Dim msgSent As New MSMQMessage

Private Sub Form_Click()
    '****************************
    ' Locate administration queue
    '(create one if one doesn't
    ' exist).
    '****************************
    Set qinfos = query.LookupQueue(Label:="Administration Queue")
    qinfos.Reset
    Set qinfoAdmin = qinfos.Next
    If qinfoAdmin Is Nothing Then
        Set qinfoAdmin = New MSMQQueueInfo
        qinfoAdmin.PathName = ".\AdminQueue"
        qinfoAdmin.Label = "Administration Queue"
        qinfoAdmin.Create
    End If

    '****************************
    ' Locate destination queue
    '(create one if one doesn't
```

```
    ' exist).
    '***************************
    Set qinfos = query.LookupQueue(Label:="Destination Queue")
    qinfos.Reset
    Set qinfoDest = qinfos.Next
    If qinfoDest Is Nothing Then
        Set qinfoDest = New MSMQQueueInfo
        qinfoDest.PathName = ".\DestQueue"
        qinfoDest.Label = "Destination Queue"
        qinfoDest.Create
    End If

    '***************
    ' Open destination queue.
    '***************
    Set qDest = qinfoDest.Open(MQ_SEND_ACCESS, MQ_DENY_NONE)

    '***************
    ' Send Message.
    '***************
    msgSent.Label = "Test Message"
    msgSent.Body = "This message tests acknowledgment messages."
    msgSent.Ack = MQMSG_ACKNOWLEDGMENT_FULL_RECEIVE
    msgSent.MaxTimeToReceive = 60
    Set msgSent.AdminQueueInfo = qinfoAdmin
    msgSent.Send qDest
    qDest.Close

    '********************************
    ' Read Acknowledgment message in
    ' administration queue.
    '********************************
    Results = MsgBox("The message was sent to the queue. Click YES to remove the
message from the queue and return a positive acknowledgment. Wait for 60 seconds and
click NO to return a negative acknowledgment.", 4)
    If (Results = vbYes) Then
        Set qDest = qinfoDest.Open(MQ_RECEIVE_ACCESS, MQ_DENY_NONE)
        Set msgDest = qDest.Receive
    End If

    Set qAdmin = qinfoAdmin.Open(MQ_RECEIVE_ACCESS, MQ_DENY_NONE)
    Set msgAdmin = qAdmin.Receive

    If msgAdmin.Class = MQMSG_CLASS_ACK_RECEIVE Then
        MsgBox "The message was removed from the queue."
    ElseIf msgAdmin.Class = MQMSG_CLASS_NACK_RECEIVE_TIMEOUT Then
        MsgBox "The message was not removed from the queue in time."
    Else
        MsgBox "The returned acknowledgment message (" + CStr(msgAdmin.Class) + ") is
not valid for this example."
    End If

End Sub
```

Sending Messages that Request a Response

By sending the **MSMQQueueInfo** object of a second queue along with a message, the sending application indicates that it expects a response message from the receiving application. The **MSMQQueueInfo** object is passed in the **ResponseQueueInfo** property of the message.

When the receiving application reads the message and sees that a response is requested (**ResponseQueueInfo** is not NULL), it should then return a response message to the queue specified by **ResponseQueueInfo**. For more information on response queues, see Response Queues.

To request a response

1. Determine what type of response message is needed. For example, the response message could be a message that only indicates a message arrived, or it could include a message body with complete instructions on what to do. Both the sending and receiving application must be able to understand the message.
2. Locate the response queue, creating one if it doesn't exist.

```
Set qinfos = query.LookupQueue(Label:="Response Queue")
qinfos.Reset
Set qinfoResp = qinfos.Next
If qinfoResp Is Nothing Then
    Set qinfoResp = New MSMQQueueInfo
    qinfoResp.PathName = ".\RespQueue"
    qinfoResp.Label = "Response Queue"
    qinfoResp.Create
End If
```

3. Locate the destination queue, creating one if it doesn't exist.

```
Set qinfos = query.LookupQueue(Label:="Destination Queue")
qinfos.Reset
Set qinfoDest = qinfos.Next
If qinfoDest Is Nothing Then
    Set qinfoDest = New MSMQQueueInfo
    qinfoDest.PathName = ".\DestQueue"
    qinfoDest.Label = "Destination Queue"
    qinfoDest.Create
End If
```

4. Open the destination queue and send the message. The example below asks if a response message is wanted.

```
Set q = qinfoDest.Open(MQ_SEND_ACCESS, MQ_DENY_NONE)

msgSent.Label = "Test Message"
msgSent.Body = "This message tests the response queue."

response = MsgBox("Do you want a response message?", 4)
If response = 6 Then
    Set msgSent.ResponseQueueInfo = qinfoResp
End If

msgSent.Send q
```

5. Read the message in the destination queue, and return a response message if one is requested. In the example below, the original message's identifier is included in the response message's correlation identifier. This provides a way to match the response message with the original message

read from the queue.

```
Set q = qinfoDest.Open(MQ_RECEIVE_ACCESS, MQ_DENY_NONE)
Set msgRead = q.Receive

If Not msgRead.ResponseQueueInfo Is Nothing Then
    Set qResp = msgRead.ResponseQueueInfo.Open(MQ_SEND_ACCESS,
MQ_DENY_NONE)
    msgResp.Label = "Response Message"
    msgResp.Body = "This is a response message"
    msgResp.CorrelationId = msgRead.id
    msgResp.Send qResp
    MsgBox "A response message was returned to :" +
msgRead.ResponseQueueInfo.PathName
Else
    MsgBox "No response was requested."
End If
```

Example

This example locates the response queue and the destination queue (creating them if needed), displays a message box that allows you to request a response message, then sends the message depending on your response.

Next, the example reads the message from the queue, returning a response message if one is requested. If the response message is requested, its correlation identifier, **CorrelationId**, is set to the message identifier of the original message and the response message is sent.

To try this example using Microsoft® Visual Basic® (version 5.0), paste the code into the code window of a form, and then run the example and click the form.

```
Dim query As New MSMQQuery
Dim qinfos As MSMQQueueInfos
Dim qinfoResp, qinfoDest As MSMQQueueInfo
Dim qRead, qResp As New MSMQQueue
Dim msgSent As New MSMQMessage
Dim msgRead As New MSMQMessage
Dim msgResp As New MSMQMessage

Private Sub Form_Click()

    '**********************************
    ' Locate response queue (create one
    ' if one doesn't exist).
    '**********************************
    Set qinfos = query.LookupQueue(Label:="Response Queue")
    qinfos.Reset
    Set qinfoResp = qinfos.Next
    If qinfoResp Is Nothing Then
        Set qinfoResp = New MSMQQueueInfo
        qinfoResp.PathName = ".\RespQueue"
        qinfoResp.Label = "Response Queue"
        qinfoResp.Create
    End If
    '**********************************
    ' Locate destination queue
    '(create one if one doesn't exist).
```

```
    '*********************************
    Set qinfos = query.LookupQueue(Label:="Destination Queue")
    qinfos.Reset
    Set qinfoDest = qinfos.Next
    If qinfoDest Is Nothing Then
        Set qinfoDest = New MSMQQueueInfo
        qinfoDest.PathName = ".\DestQueue"
        qinfoDest.Label = "Destination Queue"
        qinfoDest.Create
    End If

    '************************
    ' Open destination queue.
    '************************
    Set q = qinfoDest.Open(MQ_SEND_ACCESS, MQ_DENY_NONE)

    '**************
    ' Send Message.
    '**************
    msgSent.Label = "Test Message"
    msgSent.Body = "This message tests the response queue."

    response = MsgBox("Do you want a response message?", 4)
    If response = 6 Then
        Set msgSent.ResponseQueueInfo = qinfoResp
    End If

    msgSent.Send q

    MsgBox "The message was sent. Check the MSMQ Explorer to see the messages in the
queue."
    q.Close

    '***********************************
    ' Read the message in the destination
    ' queue and send response message if
    ' one is requested.
    '***********************************
    Set q = qinfoDest.Open(MQ_RECEIVE_ACCESS, MQ_DENY_NONE)
    Set msgRead = q.Receive

    If Not msgRead.ResponseQueueInfo Is Nothing Then
        Set qResp = msgRead.ResponseQueueInfo.Open(MQ_SEND_ACCESS,
MQ_DENY_NONE)
        msgResp.Label = "Response Message"
        msgResp.Body = "This is a response message"
        msgResp.CorrelationId = msgRead.id
        msgResp.Send qResp
        MsgBox "A response message was returned to :" +
msgRead.ResponseQueueInfo.PathName
    Else
        MsgBox "No response was requested."
    End If

End Sub
```

Sending Private Messages

Sending private messages requires setting the privacy level of the message, and, as an option, setting the privacy level of the queue where the message is sent, before the message is sent.

The properties used to set the privacy level of the message are the **MSMQMessage** object's **PrivLevel** and **EncryptAlgorithm** properties. The property used to set the privacy level of the queue is the **MSMQQueueInfo** object's **PrivLevel** property.

Note The actual call to send and read private messages is the same as the call to send and read non-private messages.

To send private messages

1. Optional. Verify that the queue can receive private messages. The **MSMQQueueInfo** object's **PrivLevel** must be set to MQ_PRIV_LEVEL_BODY or MQ_PRIV_LEVEL_OPTIONAL. If set to MQ_PRIV_LEVEL_BODY, the queue can only accept private messages. Non-private messages will be ignored.
   ```
   qinfo.PrivLevel = MQ_PRIV_LEVEL_BODY
   ```
2. Open the queue for sending messages.
   ```
   Set q = qinfo.Open(MQ_SEND_ACCESS, MQ_DENY_NONE)
   ```
3. Set the **MSMQMessage** object's **PrivLevel** property to MQMSG_PRIV_LEVEL_BODY.
   ```
   msg.PrivLevel = MQMSG_PRIV_LEVEL_BODY
   ```
4. Optional. Set the encryption algorithm used to encrypt the message.
   ```
   msg.EncryptAlgorithm = MQMSG_CALG_RC4
   ```
5. Send the message.
   ```
   msg.Send q
   ```

Example

```
Dim qinfo As New MSMQQueueInfo
Dim q As New MSMQQueue
Dim msg As New MSMQMessage

Private Sub Form_Click()
  '**********************
  ' Create queue
  '**********************
  Set qinfo = New MSMQQueueInfo
  qinfo.PathName = ".\PrivacyTest"
  qinfo.Label = "Test Queue"
  qinfo.PrivLevel = MQ_PRIV_LEVEL_BODY
  qinfo.Create
  '**********************
  ' Open queue.
  '**********************
  Set q = qinfo.Open(MQ_SEND_ACCESS, MQ_DENY_NONE)
  '**********************
  ' Send message.
  '**********************
```

```
  msg.Label = "Test Message"
  msg.Body = "This is a private message."
  msg.PrivLevel = MQMSG_PRIV_LEVEL_BODY
  msg.EncryptAlgorithm = MQMSG_CALG_RC4
  msg.Send q
  MsgBox "The message " + msg.Label + " was sent."
  q.Close
  '**********************
  ' Receive message.
  '**********************
  Set q = qinfo.Open(MQ_RECEIVE_ACCESS, MQ_DENY_NONE)
  Set msg = q.Receive
  If msg.PrivLevel = MQMSG_PRIV_LEVEL_BODY Then
      MsgBox "Message " + msg.Label + " is private."
  Else
      MsgBox "Message " + msg.Label + " is not private."
  End If

    MsgBox "Encryption alogithm is: " + CStr(msg.EncryptAlgorithm)


End Sub
```

Sending Messages Using an Internal Transaction

To send a message within an internal transaction, the **MSMQTransactionDispenser** is used to create the transaction object. After the transaction object is created, make sure it is referenced in the call to **Send**.

To send a message using an internal transaction

1. Specify the type of transaction dispenser you want to use. For internal transactions, specify the **MSMQTransactionDispenser** object.

   ```
   Dim xdispenser as New MSMQTransactionDispenser
   ```

2. Call **BeginTransaction**.

   ```
   Set xact = xdispenser.BeginTransaction
   ```

3. Open the queue with send access.

   ```
   Set qSend = qinfo.Open(MQ_SEND_ACCESS, MQ_DENY_NONE)
   ```

4. Create and send the message. Make sure the call to Send is associated with the transaction.

   ```
   msg.Label = "MyTransaction message"
   msg.Body = "Message 1 Body"
   msg.Send qSend, xact                    'Associates send with xact.
   ```

Example

This example sends a single message within an internal transaction.

```
Dim xdispenser as New MSMQTransactionDispenser
Dim xact as MSMQTransaction

Dim qSend as MSMQQueue
Dim msg as New MSMQMessage

Set xact = xdispenser.BeginTransaction

'Assumes queue already exists and is transactional.
Set qSend = qinfo.Open(MQ_SEND_ACCESS, MQ_DENY_NONE)
msg.Label = "MyTransaction message"
msg.Body = "Message 1 Body"
msg.Send qSend, xact                    'Associates send with xact.
```

Sending Messages Using an MS DTC External Transaction

To send a message within an MS DTC external transaction, the **MSMQCoordinatedTransactionDispenser** is used to create the transaction object. After the transaction object is created, make sure it is referenced in the call to **Send**.

To send a message using an external transaction

1. Specify the type of transaction dispenser you want to use. For external transactions, specify the **MSMQCoordinatedTransactionDispenser** object.

   ```
   Dim xdispenser as New MSMQCoordinatedTransactionDispenser
   ```

2. Call **BeginTransaction**.

   ```
   Set xact = xdispenser.BeginTransaction
   ```

3. Open the queue with send access.

   ```
   Set qSend = qinfo.Open(MQ_SEND_ACCESS, MQ_DENY_NONE)
   ```

4. Create and send the message. Make sure the call to Send is associated with the transaction.

   ```
   msg.Label = "MyTransaction message"
   msg.Body = "Message 1 Body"
   msg.Send qSend, xact                    'Associates send with xact.
   ```

Example

This example sends a single message within an external transaction.

```
Dim xdispenser as New MSMQCoordinatedTransactionDispenser
Dim xact as MSMQTransaction

Dim qSend as MSMQQueue
Dim msg as New MSMQMessage

Set xact = xdispenser.BeginTransaction

'Assumes queue already exists and is transactional.
Set qSend = qinfo.Open(MQ_SEND_ACCESS, MQ_DENY_NONE)
msg.Label = "MyTransaction message"
msg.Body = "Message 1 Body"
msg.Send qSend, xact                    'Associates send with xact.
```

Reading Messages in a Queue

Reading messages in a queue can be done synchronously or asynchronously.

When reading messages synchronously, execution is blocked until a message is available, or the message time-out timer expires.

When reading messages asynchronously, execution continues until an Arrived or ArrivedError event is fired. The Arrived and ArrivedError events are provided by the **MSMQEvent** object.

Cursors are never explicitly defined when reading messages in a queue. However, a single implicit cursor can be used to navigate through the queue. The methods used to navigate through the queue are all defined in terms of this implicit cursor.

For examples of reading messages, see:

- Reading Messages Synchronously.
- Reading Messages Asynchronously.

For a description of how the messages in a queue are moved about when an application peeks at or retrieves a message, see Reading Messages.

Reading Messages Synchronously

When reading messages synchronously, all calls are blocked until the next message is available or timeout occurs.

To read a message synchronously

1. Open the queue with receive or peek access.

```
Set qDest = qinfoDest.Open(MQ_RECEIVE_ACCESS, MQ_DENY_NONE)
```

2. Call **Receive** or **Peek** to read each message in the queue. The example below uses **Receive** to remove each message in the queue. (Using **Peek** in the code below create an infinite loop.)

```
Do While True
    Set msgDest = qDest.Receive(ReceiveTimeout:=1000)
    If msgDest Is Nothing Then Exit Do
    MsgBox msgDest.Label + " is removed from the queue."
Loop
```

Example

This example reads all messages in a queue, removing each message as it is read. An error handler is added to trap any errors generated as a result of the **Receive** call.

To try this example using Microsoft® Visual Basic® (version 5.0), paste the code into the Code window of a form, and then run the example and click the form.

```
Option Explicit
Dim qinfoDest As MSMQQueueInfo
Dim qDest As MSMQQueue
Dim msgDest As MSMQMessage

Private Sub Form_Click()

    '**********************
    ' Removes all messages
    ' in the queue.
    '**********************

    Set qDest = qinfoDest.Open(MQ_RECEIVE_ACCESS, MQ_DENY_NONE)

    On Error GoTo Handler

    Do While True
        Set msgDest = qDest.Receive(ReceiveTimeout:=1000)
        If msgDest Is Nothing Then Exit Do
        MsgBox msgDest.Label + " is removed from the queue."
    Loop

    Exit Sub

    '***************
    ' Error Handler
    '***************

Handler:
    If (Err = MQ_ERROR_IO_TIMEOUT) Then
        MsgBox "All messages are removed from the queue."
```

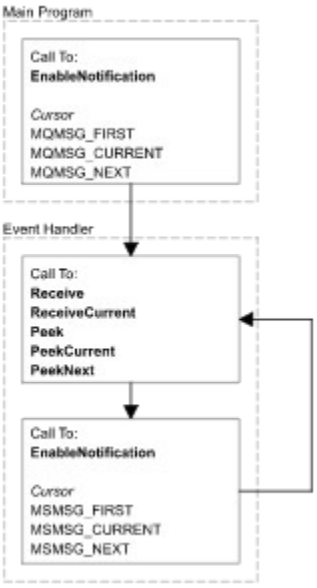
```
        Exit Sub
    Else
        MsgBox "Unexpected error!"
    End If

End Sub
```

Reading Messages Asynchronously

When reading messages asynchronously using ActiveX components, **EnableNotification** must be called for each message that is read from the queue.

Typically, **EnableNotification** is called for the first message to be read, and then again in the event handler after you read the arrived message. The subsequent calls to **EnableNotification** are needed to read the next message in the queue. Notification returns a single Arrived event for each message found in the queue.

The following diagram shows the basic programming model used for reading messages asynchronously. It includes two calls to **EnableNotification**, plus a call to read the message from the queue.



The model in the diagram shows that there are many ways to combine the two calls to **EnableNotification** and the call to read the message in the queue. For example, to purge all the messages in a queue you could call **EnableNotification** with Cursor set to MQMSG_FIRST, call **Receive**, then call **EnableNotification** with Cursor set to MQMSG_FIRST.

However, not all combinations necessarily make good programming "sense." For instance, it is possible to write an event handler that would only read every other message in the queue.

The initial call to **EnableNotification** and the subsequent call from the event handler can tell MSMQ to check if a message is in the queue at all (*Cursor* = MQMSG_FIRST), if a message is at the current cursor location (*Cursor* = MQMSG_CURRENT), or if a message is at the next position after the cursor (*Cursor* = MQMSG_NEXT). The default is to trigger the Arrived event when MSMQ finds any message in the queue (*Cursor* = MQMSG_FIRST).

The calls to read the message in the queue include: **Receive**, **ReceiveCurrent**, **Peek**, **PeekCurrent**, or **PeekNext**.

Note Each call to **EnableNotification**, plus the calls to **ReceiveCurrent**, **PeekCurrent**, or **PeekNext**, provide numerous ways to navigate through the queue. Each call can affect how the implied cursor is moved through the queue.

To find a specific message asynchronously

1. Call **Open** to open queue with receive access and **EnableNotification** to start notification.

```
Set queue = qInfo.Open(MQ_RECEIVE_ACCESS, MQ_DENY_NONE)
```

```
queue.EnableNotification Event:=Event, Cursor:=MQMSG_CURRENT,
ReceiveTimeout:=1000
```

2. Write Arrived event handler. The following event handler calls **PeekCurrent** to look at the current message, then uses **EnableNotification** (*Cursor* = MQMSG_NEXT) to start notification with the cursor pointing to the next location.

```
Private Sub TheEvent_Arrived(ByVal Queue As Object, ByVal Cursor As Long)
Dim msgrec As MSMQMessage
On Error GoTo Error_TheEvent_Arrived
    Set msgrec = queue.PeekCurrent(ReceiveTimeout:=0)
    If msgrec.AppSpecific = 34 Then
        Set msgrec = queue.ReceiveCurrent(ReceiveTimeout:=0)
        MsgBox "Found a message with AppSpecific = 34", vbOKOnly, "Inside the
Arrived Event handler"
    Else
        queue.EnableNotification TheEvent, MQMSG_NEXT, 1000
    End If
    Exit Sub
Error_TheEvent_Arrived:
    MsgBox Err.Description + " in TheEvent_Arrived sub"
End Sub
```

3. Write ArrivedError event handler.

```
Private Sub TheEvent_ArrivedError(ByVal Queue As Object, ByVal ErrorCode As Long,
ByVal Cursor As Long)
    MsgBox Err.Description + " in TheEvent_ArrivedError sub"
End Sub
```

Example

This example sends several messages with different application-specific identifiers to a queue, then searches the queue for the message whose application-specific identifier equals 34.

```
Option Explicit
Dim queue As MSMQQueue
Dim WithEvents TheEvent As MSMQEvent

Private Sub Form_Click()
Dim qinfo As New MSMQQueueInfo
Dim msgSend As New MSMQMessage
Dim i As Integer
On Error Resume Next
    Set TheEvent = New MSMQEvent
    qInfo.PathName = ".\AsyncSearchDemo"
    qInfo.Create
On Error GoTo Error_Form_Click
    Set queue = qinfo.Open(MQ_SEND_ACCESS, MQ_DENY_NONE)
    msgSend.AppSpecific = 24
    msgSend.Send queue
    msgSend.AppSpecific = 34
    msgSend.Send queue
    msgSend.AppSpecific = 44
    msgSend.Send queue
    msgSend.AppSpecific = 54
```

```
    msgSend.Send queue
    msgSend.AppSpecific = 64
    msgSend.Send queue
    queue.Close

'*****************************
'* Open queue and start
'* notification.
'*****************************
    Set queue = qinfo.Open(MQ_RECEIVE_ACCESS, MQ_DENY_NONE)
    queue.EnableNotification Event:=Event, Cursor:=MQMSG_CURRENT,
ReceiveTimeout:=1000
    Exit Sub
Error_Form_Click:
    MsgBox Err.Description
End Sub

'*****************************
'* Define Arrived event handler.
'*****************************
Private Sub TheEvent_Arrived(ByVal Queue As Object, ByVal Cursor As Long)
Dim msgrec As MSMQMessage
On Error GoTo Error_TheEvent_Arrived
    Set msgrec = queue.PeekCurrent(ReceiveTimeout:=0)
    If msgrec.AppSpecific = 34 Then
        Set msgrec = queue.ReceiveCurrent(ReceiveTimeout:=0)
        MsgBox "Found a message with AppSpecific = 34", vbOKOnly, "Inside the Arrived
Event handler"
    Else
        queue.EnableNotification TheEvent, MQMSG_NEXT, 1000
    End If
    Exit Sub
Error_TheEvent_Arrived:
    MsgBox Err.Description + " in TheEvent_Arrived sub"
End Sub

Private Sub TheEvent_ArrivedError(ByVal Queue As Object, ByVal ErrorCode As Long, ByVal
Cursor As Long)
    MsgBox Err.Description + " in TheEvent_ArrivedError sub"
End Sub
```

Retrieving a Queue's Properties Using ActiveX Components

After a queue is created, the **MSMQQueueInfo** object's properties can be reset at any time by calling the object's **Refresh** method.

Refresh has two uses. It is required before an application can read any queue property that is set by MSMQ (MSMQ-generated properties such as **QueueGuid** cannot be used until **Refresh** is called). Second, it can also be used to refresh the properties of the **MSMQQueueInfo** object when another application resets a queue's properties. When **Refresh** is called, it updates all of the queue's properties, not just those set by the application. However, you can only refresh the properties of private queues if the queue is located on your local computer. To see a list of queue properties, scroll down to the bottom of this topic.

Note Properties of public queues can also be retrieved by doing a query on MQIS. For details, see Locating a Public Queue.

To retrieve a queue's properties

1. Determine whether the sending application has the access rights to look at the queue's properties. If the application does not have MQSEC_GET_QUEUE_PROPERTIES access rights, an MQ_ERROR_ACCESS_DENIED error is returned to **Refresh**. For a complete list of queue access rights, see Access Control.
2. Call **Refresh**.

Example

This example creates a public queue, then uses **Refresh** to update the MSMQQueueInfo so it can display the queue's identifier (**QueueGuid**). To try this example using Microsoft® Visual Basic® (version 5.0), paste the code into the Declaration section of a form that has a single text box, and then run the example and click the form.

```
Dim qinfo As New MSMQQueueInfo

Private Sub Form_Click()

  Set qinfo = New MSMQQueueInfo
  qinfo.PathName = ".\RefreshTest"
  qinfo.Label = "Test Queue"
  qinfo.Create

  qinfo.Refresh             'Required to update QueueGuid
  Text1.Text = "quidQueue = " + CStr(qinfo.QueueGuid)

End Sub
```

Queue Properties

The following queue properties can be retrieved:

Authenticate

BasePriority

Journal

JournalQuota

Label

PrivLevel

Quota

ServiceTypeGuid

The following queue properties are set by MSMQ when it creates the queue. To read these properties, the application must explicitly call **Refresh** before they can be read.

CreateTime (public queues only)

FormatName (public and private queues)

IsTransactional (public and private queues)

ModifyTime (public queues only)

QueueGuid (public queues only)

Setting a Queue's Properties Using ActiveX Components

The properties of a queue can be dynamically set by calling the **MSMQQueueInfo** object's **Update** method. **Update** can only called on queues that exist. It cannot be called on an **MSMQQueueInfo** object before the queue is created or after the queue is deleted.

When **Update** is called, MQIS (public queues) and the local computer (private queues) are updated with the current settings of the **MSMQQueueInfo** object's properties.

Note Properties for private queues can only be updated if the queue is located on the local computer.

To set a queue's properties

1. Determine whether the application has the access rights needed to set the queue's properties. If the application does not have MQSEC_SET_QUEUE_PROPERTIES access rights, an MQ_ERROR_ACCESS_DENIED error is returned to **Update**. For a complete list of queue access rights, see Access Control.
2. Set the queue properties that need to be changed to a new value.
3. Call **Update**.

Example

This example creates a public queue and then uses **Update** to change the queue's label. To try this example using Microsoft® Visual Basic® (version 5.0), paste the code into the Declaration section of a form that has a single text box, and then run the example and click the form.

```
Dim qinfo As New MSMQQueueInfo

Private Sub Form_Click()

  Set qinfo = New MSMQQueueInfo
  qinfo.PathName = ".\UpdateTest1"
  qinfo.Label = "Test Queue"
  qinfo.Create
  MsgBox "The queue's label is: " + qinfo.Label

  qinfo.Label = "New Queue Label"
  qinfo.Update
  MsgBox "The queue's new label is: " + qinfo.Label

End Sub
```

Queue properties

The following queue properties can be set by calling **Update**:

Authenticate

BasePriority

Journal

JournalQuota

Label

PrivLevel

Quota

ServiceTypeGuid

The following properties cannot be set by **Update**:

CreateTime (set by MSMQ)

IsTransactional (set when queue is created)

ModifyTime (set by MSMQ)

PathName (set when queue is created)

QueueGuid (set by MSMQ)

Using Transactions

When an MSMQ application performs a transaction, it must work with all the transactional services (Microsoft® Distributed Transaction Coordinator [MS DTC] and all resource managers) needed to complete the transaction. This includes all the resource mangers associated with the transaction, including MSMQ and MS DTC as the transaction manager.

From a system perspective, the process for performing a transaction starts with the transaction application asking MS DTC or MSMQ for a new transaction object. Once a transaction object is available, the application can then make any number of transactional calls, as well as any number of non-transactional calls, to all the resource managers needed by the application.

It is the application's responsibility to check the return values of all the functions called. If all calls succeed, the application can call the **Commit** method of the transaction object. This does not mean the transaction is committed; it only means that the application is ready to commit.

MS DTC now starts a two-phase commit procedure, asking each participating resource manager to prepare itself and to inform MS DTC if it is ready to commit. If all the resource managers commit, MS DTC commits the transaction. If any one resource manager does not commit, the transaction is aborted.

All this activity by MS DTC is invisible to the transaction application. The application only sees the return value of the **Commit** function it calls. When a successful **Commit** is returned to the application, the transaction is completed.

Note When a transaction is completed, it does not mean the work is completed. When the transaction manager commits the transaction, it only means that each resource manager guarantees that it will do its part of the transaction at some later time.

About the MSMQ Guide

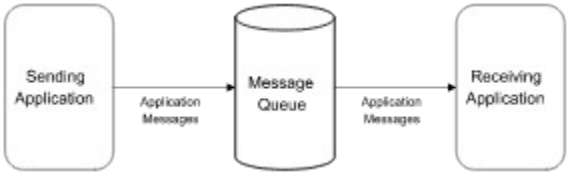
The Microsoft® Message Queue Server Guide outlines conceptual information on different queues that can be used, different messages that are available, how queues and messages are defined by properties, as well as information on other MSMQ services.

The following topics are in the "MSMQ Guide:"

About MSMQ

With the trend toward distributed computing in enterprise environments, it is important to have flexible and reliable communication among applications. Businesses often require independent applications that are running on different systems to communicate with each other and exchange messages even though the applications may not be running at the same time.

Microsoft® Message Queue Server (MSMQ) technology enables applications running at different times to communicate across heterogeneous networks and systems that may be temporarily offline. Within an MSMQ enterprise, applications send messages to queues and read messages from queues. The following illustration shows how queues hold the messages used by both the sending and receiving application.



MSMQ ensures that all messages eventually reach their destination, whether a message is sent to a queue or a message is read from a queue. MSMQ provides guaranteed message delivery, efficient routing, security, and priority-based messaging.

MSMQ is different from remote procedure calls (RPC), Windows Sockets, and messaging API (MAPI). Because MSMQ is a connectionless message service, where applications do not need to maintain a session, it is different from RPC where applications are required to maintain sessions. And although Windows Sockets provides low-level functions for writing applications, Windows Sockets does not allow applications to run at different times in the way that MSMQ does. MSMQ is also different from MAPI (an e-mail oriented service) in that it uses a more general-purpose message queuing model than MAPI.

MSMQ Objects

There are various objects available in Microsoft® Message Queue Server (MSMQ), including machines, queues, and messages. For information on all MSMQ objects, refer to the *Microsoft Message Queue Server Administrator's Guide*.

MSMQ machine, queue, and message objects are all defined by their properties.

Using the MSMQ API, you can:

- Create queue objects
- Locate queue objects
- Open and delete queue objects
- Set and get queue object properties
- Send and receive message objects
- Get machine object properties

For more information see: MSMQ Queues, MSMQ Messages, MSMQ Properties, and MSMQ Computers.

MSMQ Queues

MSMQ queues are all based on the same model. However, who creates the queue, who sends messages to the queue, and how the messages are used varies between the different types of queues.

Queues can be created by applications (message, administration, and response queues) or by MSMQ (journal, dead letter, and report queues). Queues created by applications are application queues and those created by MSMQ are system queues.

Queues can receive their messages from applications (message and response queues) or from MSMQ (administration, journal, dead letter and report queues).

Regardless of who creates the queue, who sends messages to the queue, or how messages are used by an application, the method for sending and receiving messages is the same. An application uses the same method of sending messages to message and response queues, and uses the same method of reading messages from message, response, administration, journal, dead letter, and report queues.

The following table lists the different kinds of queues, whether they are application or system queues, and a brief description of how they can be used by your applications.

Queue	Queue Type	Description
Message queues	Application	Application-generated messages can be sent to and read from these queues. They can be *public* (they are defined in the MSMQ information store and can be located using MSMQ Lookup functions) or *private* (they are defined on the local machine and cannot be located using MSMQ Lookup functions).
Administration queues	Application	Local queue used to store MSMQ-generated positive and negative acknowledgments when sending messages.
Response queues	Application	Used to return application-generated response messages from the application reading the messages in a queue.
Journal queues	System	Used to store copies of application-generated messages. There are two types of journal queues: machine journals and queue journals.
Dead Letter queues	System	Used to store application-generated messages that cannot be delivered. There are two dead letter queues, one for transaction messages and the other for non-transaction messages.
Report queues	System	Used to track the progress of your messages as they move through your enterprise. Report queues receive MSMQ-generated report messages.

Application Queues

Application queues include message queues, administration queues, and response queues. These queues are referred to as application queues because they are created by applications.

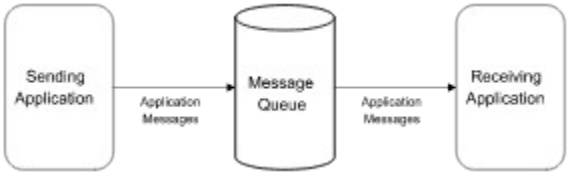
Message queues and response queues are used to store messages that are sent and read by applications. Administration queues are used to store acknowledgment messages sent by MSMQ.

For information on a specific type of queue, see:

- Message Queues
- Administration Queues
- Response Queues

Message Queues

Message queues provide applications with a way to exchange information through messages. Applications can send messages to these queues and read the messages they contain. The following illustration shows a single message queue with one application sending messages to the queue and another application reading its messages.



However, this illustration does not indicate where messages are stored. A queue's messages are stored on the computer that was designated when the queue is created.

To an application, the most important classifications of these queues are whether they are public or private. Although both public and private queues have significant advantages, the choice to make a queue public or private depends on whether or not you want others to be able to locate the queue.

The advantage of public queues is that they are registered in the MQIS information store (MSMQ) so they can be located by any MSMQ application. Public queues are persistent and their registration information can be backed up on the MSMQ enterprise, making them good for long-term use.

Private queues are registered on the local computer and typically cannot be seen by other applications. To register a private queue, MSMQ stores a description of the queue, plus cached information about any local public queues, in the LQS (local queue storage) directory on the local computer (the default LQS directory is \program files\msmq\storage\lqs). Private queues do have the advantage of no MQIS overhead (faster to create, no latency, and no replication), and they can be created and deleted when the MQIS is not working.

Private queues can be exposed to other applications if the queue's location is sent to the other application. This is done by sending the private queue's format name to the other application. (For applications using API functions, see PROPID_M_RESP_QUEUE. For applications using ActiveX components, see **ResponseQueueInfo**).

Examples

- Creating A Queue (using API functions)
- Creating A Queue (using ActiveX components)
- Locating a Public Queue (using API functions)
- Locating a Public Queue (using ActiveX components)
- Opening a Queue (using API functions)
- Sending Messages To a Queue (using API functions)
- Sending Messages To a Queue (using ActiveX components)

Administration Queues

Administration queues are specified by the sending application when it sends its messages. They receive MSMQ-generated acknowledgment messages that indicate whether the messages sent by the application arrived (a positive acknowledgment) or whether an error occurred (a negative acknowledgment).

The sending application must specify the queue it wants used as the administration queue and the type of acknowledgment messages it wants returned. Any available queue can be specified as the administration queue.

Note When an administration queue and a response queue are needed, their functionality can be combined into a single queue. For information on response queues, see Response Queues.

For an example of how administration queues can be used, see:

- Sending Messages that Request Acknowledgments (using API functions).
- Sending Messages that Request Acknowledgments (using ActiveX components)

Response Queues

Response queues are specified by the sending application when it sends its messages. They receive application-generated messages that are sent back to the sending application when the receiving application reads the message from the queue.

Any available queue can be specified as the response queue.

Note The message properties used for specifying a response queue can also be used to send the location of a private queue to another application.

When a response queue and an administration queue are needed, their functionality can be combined into a single queue. For information on administration queues, see Administration Queues.

For an example of how administration queues can be used, see:

- Sending Messages To a Queue (using API functions).
- Sending Messages To a Queue (using ActiveX).

System Queues

System queues include journal, dead letter, and report queues. MSMQ or the MSMQ Administrator creates these queues, and only MSMQ can send messages to them. Applications can read only messages from these queues.

Journal and dead letter queues are used to store copies of application messages. Report queues contain report messages that track the path of a specific application message.

For information on a specific type of queue, see:

- Journal Queues
- Dead Letter Queues
- Report Queues

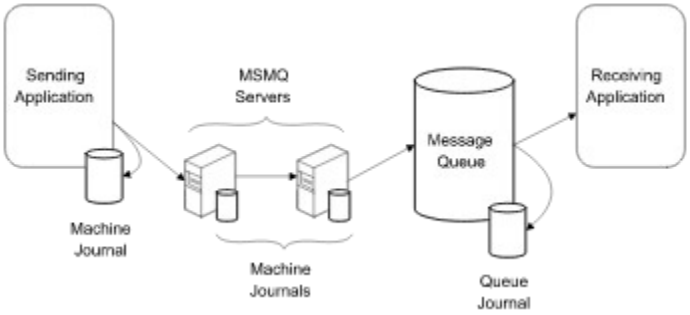
Journal Queues

Journal queues are automatically created by MSMQ whenever a computer is added to the MSMQ enterprise or a queue is created.

When a computer is created, MSMQ creates, on the computer, a machine journal queue that is used to track the messages sent from the computer and MSMQ-generated report messages.

When a queue is added, MSMQ creates a queue journal where the queue is located. The queue journal is used to track the messages removed from a queue.

The following illustration shows a message queue with its queue journal, plus several computers with machine journals. When the sending application sends a message, a copy of the message is optionally stored in the source machine journal. When the receiving application removes a message from the message queue, a copy of the message is optionally sent to the queue journal.



MSMQ never removes messages from a queue or machine journal. It is up to the application using the queue to clear the queue's messages by retrieving them, or by purging the queue using the MSMQ Explorer.

For information on opening one of these queues, see:

- Opening a Queue Journal
- Opening a Machine Journal

Opening a Queue Journal

To read messages in a queue journal (you cannot send messages to a queue journal), you must specify the queue that the journal is associated with. This is done by appending ";JOURNAL" to the format name of the queue as shown below:

PUBLIC=QueueGUID;JOURNAL
PRIVATE=MachineGUID\QueueNumber;JOURNAL

Note You cannot open a queue journal using a direct format name.

For information on reading messages, see:

- Reading Messages In a Queue (using API functions)
- Reading Messages In a Queue (using ActiveX components)

Opening a Machine Journal

To read messages in a machine journal (you cannot send messages to a machine journal), you must specify the computer where the journal resides. When opening the machine journal, the format name of the queue should look like the following:

MACHINE=MachineGUID;JOURNAL

Note You cannot open a machine journal using a direct format name.

For information on reading messages see:

- Reading Messages In a Queue (using API functions)
- Reading Messages In a Queue (using ActiveX components)

Dead Letter Queues

There are two kinds of dead letter queues, one for non-transaction messages and the other for transaction messages. Both types of queues are created by MSMQ whenever a computer is added to the enterprise.

Dead letter queues hold messages that could not be delivered. For example, a message is placed in the dead letter queue when it is not delivered in time or when a wrong destination queue is specified. When a message is placed in a dead letter queue, MSMQ sets the message's class property to the appropriate negative acknowledgment.

For non-transaction messages, MSMQ sends messages to the dead letter queue of the computer that could not send the message. This could be the source computer, the destination computer, or any MSMQ routing server in between.

Note Only one copy of a message is stored on a computer at one time. If the message reached the next computer successfully, a copy of the message is stored in that computer's journal queue. If the message could not be delivered to the next computer, it is stored in the current computer's dead letter queue.

Transaction messages are treated differently from non-transaction messages. When the sending application does not receive an acknowledgment that the transaction message reached the target queue, or the receiving application does not commit to the transaction, MSMQ automatically sends the transaction message to the transaction dead letter queue on the source computer. Undeliverable transaction messages are never stored on an MSMQ routing server.

However, MSMQ does not automatically send non-transaction messages to a dead letter queue. The sending application must specify that it wants MSMQ to send undeliverable messages to a dead letter queue at the time a message is sent.

Opening a Dead Letter Queue

To read the messages in the dead letter queue (an application cannot send messages to a dead letter queue), the format name used to open the queue should look like one of the following:

MACHINE=MachineGUID;DEADLETTER (for non-transaction messages)
-or-
MACHINE=MachineGUID;DEADXACT (for transaction messages)

Note You cannot open these queues using a direct format name.

For an example of reading messages in a dead letter see Reading Messages in a Dead Letter Queue (using API functions).

Report Queues

Report queues contain MSMQ-generated report messages that track the route of your application messages as they move toward the queue. A report message is optionally generated each time a message passes through an MSMQ routing server.

Report queues are created by the MSMQ Administrator. After the MSMQ Administrator has created the report queue, you can indicate that you want MSMQ to generate report messages by setting the message trace property.

Referencing a Queue

MSMQ Queues are referenced in three ways: By their MSMQ pathname, their format name, and a queue handle. The method that is used depends on what is happening with the queue.

First, the queue's MSMQ pathname is used to create a queue. After the queue exists, its format name is used to open the queue for sending or reading messages, resetting the queue's properties (including its security), and deleting the queue. After the queue is open, while messages are sent to the queue or read from the queue, a queue's handle is used.

For information on the three ways to reference the queue, see:

- MSMQ Pathname
- Format Name
- Queue Handle

MSMQ Pathname

The MSMQ pathname is used when creating the queue. It tells MSMQ where to store the queue's messages, where to register the queue, and provides a name for the queue. A unique MSMQ pathname must be provided by the application when the queue is created.

For example, here is the MSMQ pathname for a private queue:

MachineName\PRIVATE$\QueueName

Note To indicate the local machine, you can substitute the string "." for the name of the local computer. For example, the following MSMQ pathname tells MSMQ to store the queue's messages on the local machine:

.\PRIVATE$\QueueName

For information on format names and queue handles, see:

- Format Name
- Queue Handle

Format Name

Format Names are used to specify a queue when making calls to several API functions.

Note For ActiveX applications, the **MSMQQueueInfo** object used to represent the created queue includes a **FormatName** property that is returned when the queue is created. The object's property can be used as it is, or it can be modified by the application.

The API function calls that require a format name include the following:

- **MQDeleteQueue**
- **MQGetQueueProperties**
- **MQGetQueueSecurity**
- **MQOpenQueue**
- **MQSetQueueProperties**
- **MQSetQueueSecurity**

Unlike most of the characteristics of a queue, the format name is not an MSMQ queue property. It is simply a unique name for the queue that is generated by MSMQ when the queue is created, or generated later by the application. MSMQ never stores the format name of a queue for later reference.

Format names can be obtained by any of the following methods.

Method	Details
When creating a queue.	MSMQ returns a format name when it creates a queue. ActiveX applications can obtain the queue's format name from the **MSMQQueueInfo** object used to create the queue.
When locating a queue.	The application can retrieve the queue's MSMQ pathname or queue identifier (GUID) and translate it into a format name for the queue. See **MQPathNameToFormatName** and **MQInstanceToFormatName** respectivley. ActiveX applications can obtain the queue's format name from the collection of **MSMQQueueInfo** objects referenced by the **MSMQQueueInfos** object returned by a query.
When the queue's handle is returned.	The application can translate the queue handle returned by **MQOpenQueue** to a format name using **MQHandleToFormatName**. ActiveX applications can obtain the queue's format name from the **MSMQQueueInfo** object's **FormatName** property.
When reading application-generated messages from a message queue.	The receiving application can retrieve the format name of the response queue and the format name of administration queue by retrieving the following message properties. The format name of the response queue is found in PROPID_M_RESP_QUEUE.

	The format name of the administration queue is found in PROPID_M_ADMIN_QUEUE.
When reading MSMQ-generated acknowledgment messages from an administration queue.	When reading the acknowledgment messages in the administration queue, the format name of the original message's destination queue can also be retrieved if PROPID_M_DEST_QUEUE was set by the sending application.

The data type holding the format name is a null-terminated Unicode string, with one of the following general formats:

```
PUBLIC=QueueGUID                                *Public queues.
PUBLIC=QueueGUID;JOURNAL                        *Public queue journals.
PRIVATE=MachineGUID\QueueNumber                 *Private queues.
PRIVATE=MachineGUID\QueueNumber;JOURNAL         *Private queue journals.
DIRECT=AddressSpecification\QueueName           *Direct format for public
                                                queues.
DIRECT=AddressSpecification\PRIVATE$\QueueName  *Direct format for
                                                private queue.
MACHINE=MachineGUID;JOURNAL                     *Machine journal.
MACHINE=MachineGUID;DEADLETTER                  *Dead letter queue.
MACHINE=MachineGUID;DEADXACT                    *Transaction dead letter
                                                queue.
CONNECTOR=ForeignCNGUID                         *Foreign queues.
CONNECTOR=ForeignCNGUID:XACTONLY                *Transaction foreign
                                                queues.
```

For information on public, private, and direct format names, see:

- Public Format Names
- Private Format Names
- Direct Format Names

For information on MSMQ pathnames and queue handles, see:

- MSMQ Pathname
- Queue Handle

For information on journals and dead letter queues, see:

- Journal Queues
- Dead Letter Queues

Public Format Names

Public format names are used to specify queues registered in the MSMQ information store. A queue's public format name contains the string "Public=", followed by the queue identifier generated by MSMQ when the queue was created.

The following is the general format for public format names:

"PUBLIC=QueueGUID"

Example:

```
Lpwstr Sz = L"PUBLIC=308FB580-1EB2-11CA-923B-08002B1075A7";
```

When this format is used, MSMQ looks in MQIS to determine what computer is currently hosting the queue, what protocol the host computer uses, and any other information it needs to get to the queue.

Note A public format name is strictly equivalent to the queue's identifier except that it is formatted as a string rather than as binary data. See PROPID_Q_INSTANCE for API calls or **QueueGuid** for ActiveX component calls.

The queue's location (the specific computer where the queue's messages are stored) is not part of the format name. This allows message operations to succeed regardless of the queue location; a public queue can be relocated to another machine and the format name remains valid. This is not true of a direct format name where the queue is bound to a specific location.

For information on other types of format names, see:

- Private Format Names
- Direct Format Names.

Private Format Names

Private format names are used to specify queues that are created and managed locally by the Queue Manager on the local computer. Unlike public queues, they are not registered in the MQIS and their scope is restricted to the local computer.

The private format name of the queue includes the string "Private=", followed by the machine identifier of the computer where the queue is located and a hexadecimal number that identifies the queue.

The following is the general format of a private format name:

"PRIVATE=MachineGUID\QueueNumber"

Example:

```
Lpwstr Sz = L"PRIVATE=ae0c5671-f190-12ce-ae10-00dd0114290f\0000000d";
```

When MSMQ detects this type of format name it does not refer to MQIS for information about the queue. However, it does use MQIS to look up information on the computer for routing purposes.

For information on other types of format names, see:

- Public Format Names
- Direct Format Names

Direct Format Names

Direct format names are used to open a queue that is not in your enterprise, or when you want to make sure MSMQ sends messages to the queue in one step. Direct format names have two parts: the address specification of the computer where the queue is located, followed by the local name of the queue (the name specified in the queue's MSMQ Pathname when the queue was created).

The address specification of the computer can be specified using two forms:

- As the network address of the target machine (including the network protocol). MSMQ supports two network protocols: TCP and SPX.
- As any string that is supported natively by the underlying operating system to identify the target machine (OS is used as the protocol to indicate that the computer's native protocol should be used).

Protocol	Description	Network Address
TCP	Connection-oriented TCP over IP.	Internet address notation (IP address).
SPX	Connection-oriented SPX	Network number and host number (separated by the ":"

	over IPX.	character).
OS	Connection using native machine-naming convention.	Any machine name supported by the underlying operating system. For Windows NT version 4.0, it is either UNC or DNS (see the following examples).

The following is the general format of a direct format name (public and private queues can be accessed directly):

DIRECT=AddressSpecification\QueueName (For public queues.)
DIRECT=AddressSpecification\PRIVATE$\QueueName (For private queues.)

Public Queue Examples:

```
Lpwstr Sz = L"DIRECT=SPX: 00000012:00a0234f7500\MyQueue";
Lpwstr Sz = L"DIRECT=TCP:157.18.3.1\MyQueue";
Lpwstr Sz = L"DIRECT=OS:elvisp.ms.com\MyQueue";
Lpwstr Sz = L"DIRECT=OS:elvisp\MqQueue";
```

Private Queue Examples:

```
Lpwstr Sz = L"DIRECT=SPX: 00000012:00a0234f7500\PRIVATE$\MyQueue";
Lpwstr Sz = L"DIRECT=TCP:157.18.3.1\PRIVATE$\MyQueue";
Lpwstr Sz = L"DIRECT=OS:elvisp.ms.com\PRIVATE$\MyQueue";
Lpwstr Sz = L"DIRECT=OS:elvisp\PRIVATE$\MqQueue";
```

When MSMQ sees a direct format name, it uses the information provided in the format name to locate the queue, not information in the MQIS (MSMQ information store).

In addition, you cannot use a direct format name to access a journal queue or dead letter queue.

For information on other types of format names, see:

- Public Format Names
- Private Format Names.

Queue Handle

A queue handle is returned to an application when a queue is opened. While the queue is opened, the application uses the queue's handle to perform the following functions:

- Send messages to the queue (**MQSendMessage**).
- Retrieve and peek at messages in the queue (**MQReceiveMessage**).
- Retrieve the format name for the queue (**MQHandletoFormatName**).
- Create a cursor for navigating through the queue (**MQCreateCursor**).
- Close the queue (**MQCloseQueue**).

Note When using ActiveX Components, the **MSMQQueue** object exposes the queue handle (**Handle**) so that applications can call MSMQ API functions directly. For example, when using Microsoft® Visual Basic®, MSMQ functions can be called directly using the Declare Function facility.

For information on MSMQ pathnames and format names, see:

- MSMQ Pathname
- Format Name

Queue Properties

Queue properties define the queue. They can be set by MSMQ or by the application. Queue properties include ways to identify the queue, specify who has access to the queue, if the queue's journal queue is to be used, and many more features.

The following table lists all the queue properties provided by MSMQ.

Property Name	Description
PROPID_Q_AUTHENTICATE	Optional. Specifies whether or not the queue only accepts authenticated messages.
PROPID_Q_BASEPRIORITY	Optional for public queues only. Specifies a single base priority for all messages sent to the queue.
PROPID_Q_CREATE_TIME	Read-only. Indicates the time and date when the queue was created. This property is set by MSMQ when **MQCreateQueue** is called.
PROPID_Q_INSTANCE	Read-only (public queues only). Identifies the created queue (not an open instance of the queue).This property is set by MSMQ when **MQCreateQueue** is called.
PROPID_Q_JOURNAL	Optional. Specifies if a target journal is maintained for a specified queue.
PROPID_Q_JOURNAL_QUOTA	Optional. Specifies the maximum size of the target journal (in Kbytes).
PROPID_Q_LABEL	Optional. Specifies a description of the queue.
PROPID_Q_MODIFY_TIME	Optional. Indicates the last time the properties of a queue were modified. This property is set by MSMQ when **MQCreateQueue** is called, then reset by MSMQ each time the queue properties are modified by calls to **MQSetQueueProperties**.
PROPID_Q_PATHNAME	Required to create the queue. Specifies the MSMQ pathname of the queue. The MSMQ pathname includes the name of the machine where the queue's messages are stored, whether the queue is public or private, and the name of the queue.
PROPID_Q_PRIV_LEVEL	Optional. Specifies the privacy level required by the queue. The privacy level determines how the queue handles encrypted messages.
PROPID_Q_QUOTA	Optional. Specifies the maximum size of the queue (in Kbytes).
PROPID_Q_TRANSACTION	Optional. Specifies whether the queue is a transaction queue or a

	non-transaction queue.
PROPID_Q_TYPE	Optional. Specifies the type of service provided by the queue.

In addition to describing the queue, queue properties can be used as filters to select a unique set of public queues. The MSMQ locate functions allow you to select the queue properties that you want to use as filters, returning only those queues that match the properties you specified. All public queue properties can be used as filters.

For information on how to locate a queue, see one of the following:

- Locating a Public Queue (using API functions)
- Locating a Public Queue (using ActiveX components)

MSMQ Messages

For the most part, *MSMQ messages* refer to the messages your application uses to send data. However, there are also several MSMQ-generated messages (for example, acknowledgment and report messages) that provide additional functionality.

Your application can use any message property to identify a message, including a message's MSMQ-generated identifier, application-defined identifiers, label, class, body, and so forth.

Each message, whether it is sent by an application or generated by MSMQ, receives an MSMQ-generated message identifier when it is sent.

The two application-defined identifiers are the correlation identifier and application-specific identifier. By using these identifiers, you can create your own method for identifying messages.

For information on how to set the correlation identifier and application-specific identifier, see PROPID_M_CORRELATIONID and PROPID_M_APPSPECIFIC (for applications using ActiveX components, see **CorrelationId** and **AppSpecific**).

Message Properties

The following is a list of all the message properties used by MSMQ.

Property	Description
PROPID_M_ACKNOWLEDGE	Specifies the types of acknowledgment messages MSMQ sends.
PROPID_M_ADMIN_QUEUE	Specifies the queue used for acknowledgment messages.
PROPID_M_ADMIN_QUEUE_LEN	Specifies the length of the administration queue.
PROPID_M_APPSPECIFIC	Specifies application-generated information such as single integer values or application defined message classes.
PROPID_M_ARRIVEDTIME	Specifies the maximum amount of time the message has to reach the queue.
PROPID_M_AUTH_LEVEL	Specifies if the message needs to be authenticated when read.
PROPID_M_AUTHENTICATED	Specifies if the message was authenticated by MSMQ.
PROPID_M_BODY	Specifies the body of the message.
PROPID_M_BODY_SIZE	Specifies the size of the message.
PROPID_M_BODY_TYPE	Specifies the type of message body (string, array of bytes, or object).
PROPID_M_CLASS	Specifies the type of message (for example, a message can be a normal, acknowledgment, or report message).
PROPID_M_CONNECTOR_TYPE	Specifies that some message properties typically generated by MSMQ are generated externally from MSMQ.
PROPID_M_CORRELATIONID	Specifies the correlation identifier of the message.
PROPID_M_DELIVERY	Specifies how the message is delivered (optimize throughput or recoverability).
PROPID_M_DEST_QUEUE	Specifies the queue where the message resides (the queue where the message is sent).
PROPID_M_DEST_QUEUE_LEN	Specifies the length of the destination queue.
PROPID_M_DEST_SYMM_KEY	Specifies the symmetric key required when sending encrypted messages to foreign queues.
PROPID_M_DEST_SYMM_KEY_LEN	Specifies the length of the symmetric key.

PROPID_M_ENCRYPTION_ALG	Specifies the algorithm used to encrypt the message body.
PROPID_M_EXTENSION	Specifies additional unformatted information.
PROPID_M_EXTENSION_LEN	Specifies the length of the unformatted information.
PROPID_M_HASH_ALG	Specifies the hash algorithm used when authenticating messages.
PROPID_M_JOURNAL	Specifies if a copy of the message is stored in the machine journal.
PROPID_M_LABEL	Specifies an application-defined label for the message.
PROPID_M_LABEL_LEN	Specifies the length of the message label.
PROPID_M_MSGID	Specifies the MSMQ-generated identifier for the message.
PROPID_M_PRIORITY	Specifies the priority of the message (where it is placed in the queue).
PROPID_M_PRIV_LEVEL	Specifies if the message is private (encrypted).
PROPID_M_PROV_NAME	Specifies the name of the cryptographic provider. Required when sending authenticated messages to foreign queues.
PROPID_M_PROV_NAME_LEN	Specifies the length of the cryptographic provider name.
PROPID_M_PROV_TYPE	Specifies the type of cryptographic provider. Required when sending authenticated messages to foreign queues.
PROPID_M_RESP_QUEUE	Specifies the queue for sending responses to the message.
PROPID_M_RESP_QUEUE_LEN	Specifies the length of the response queue.
PROPID_M_SECURITY_CONTEXT	Specifies security information for authenticating the message.
PROPID_M_SENDER_CERT	Specifies the security certificate used to authenticate the message.
PROPID_M_SENDER_CERT_LEN	Specifies the length of the sender certificate buffer.
PROPID_M_SENDERID	Identifies the user who sent the message.
PROPID_M_SENDERID_LEN	Indicates the length of the sender identifier.
PROPID_M_SENDERID_TYPE	Specifies the type of sender identifier found in PROPID_M_SENDERID (currently, the only type of sender identifier available to MSMQ is an SID, or

	security identifier).
PROPID_M_SENTTIME	Indicates the date and time that the message was sent by the source queue.
PROPID_M_SIGNATURE	Specifies a digital signature for authenticating the message. Used to determine who sent the message.
PROPID_M_SIGNATURE_LEN	Specifies the length of the digital signature used for authenticating the message.
PROPID_M_SRC_MACHINE_ID	Specifies the computer where the message originated.
PROPID_M_TIME_TO_BE_RECEIVED	Specifies how long the receiving application has to remove the message from the queue.
PROPID_M_TIME_TO_REACH_QUEUE	Specifies how long the message has to reach the queue.
PROPID_M_TRACE	Specifies if the route of the message is traced.
PROPID_M_VERSION	Specifies the version of MSMQ used to send the message.
PROPID_M_XACT_STATUS_QUEUE	Specifies the format name of the transaction status queue.
PROPID_M_XACT_STATUS_QUEUE_LEN	Indicates the length (in Unicode characters) of the transaction status queue's format name buffer.

Sending Messages

Sending messages in MSMQ is always an asynchronous operation. When you are sure the queue is open, you can continue to send messages, never stopping to wait for a reply.

In addition to the basic asynchronous operation, you can add functionality to your send operation by using:

- Message timers to control how long your messages stay in the system
- Machine journals to store a copy of the messages you send
- Administration queues to receive MSMQ-generated negative and positive acknowledgment messages
- Response queues for application-defined response messages
- Report queues for tracking

For information on all the message properties that can be used when sending messages with MSMQ API functions, see:

- Sending Messages To a Queue (using API functions).
- Sending Messages To a Queue (using ActiveX components).

Reading Messages

MSMQ messages can be read from a queue either synchronously or asynchronously. In addition, they can be read within a transaction (for information on reading messages in transactions, see MSMQ Transactions).

When an application synchronously reads messages in a queue, all calls are blocked until the next message is available or a specific amount of time expires. The amount of time the application waits can be 0, a specific amount of time (in milliseconds), or the maximum time allowed by your MSMQ enterprise. When the time expires for a synchronous read, MSMQ returns a NULL message (for applications using ActiveX components) or an MQ_ERROR_IO_TIMEOUT error (for applications using API functions).

There are four ways to receive messages asynchronously:

- Use a callback function.
- Use a Windows Event mechanism.
- Use a Windows NT completion port.
- Use the MSMQ ActiveX components, defining an event handler that is notified when a message arrives or a time-out occurs. ActiveX components return MQ_ERROR_IO_TIMEOUT when a time-out occurs.

When using a callback function, MSMQ reads the message by calling the callback function that is currently registered. The function is called if a message is immediately available, when the message arrives, or when the time expires.

When using an event mechanism, an **OVERLAPPED** structure provides a valid handle (*hEvent* field) to an event object. When a suitable message arrives, or a timeout occurs, the event object is set to the signaled state. For more information on **OVERLAPPED** structures, see the Platform SDK.

When using a Windows NT® completion port, a queue handle can be associated with the port to receive messages asynchronously. For more information, see **CreateIOCompletionPort** in the Microsoft Platform SDK.

When reading messages in a queue, MSMQ can peek at the messages (leaving them in the queue) or retrieve them (removing the messages from the queue).

For examples of reading messages asynchronously, see:

- Reading Messages Asynchronously (using API functions)
- Reading Messages Asynchronously (using ActiveX)

Peeking or Retrieving Messages

MSMQ provides two methods for reading messages in a queue. The application can peek at the messages in the queue or retrieve the messages in the queue.

Peeking allows you to check a message without removing it from the queue. For example, if you are searching for a specific message in a queue, you may want to peek at each message's label or identifier to locate the message, and then retrieve the complete message once it is found.

Retrieving a message in a queue removes it from the queue. For example, if you want to purge a queue, you could retrieve each message in the queue.

Reading Messages with Cursors

Cursors are used to read messages that are not necessarily at the front of the queue. Cursors can be application-defined or implied, depending on the development platform you are using.

Using any number of application-defined cursors, C applications can navigate through a queue. Each cursor is independent of all other cursors, including those generated by other applications. Moving one cursor has no effect on where another cursor points.

ActiveX applications cannot use multiple, application-defined cursors. They have a single, implied cursor to navigate through the queue. Each application's implied cursor is, however, independent of other application's cursors.

Regardless of the development platform, there is no limit to the number of applications that can use cursors to navigate through a queue.

Message and Cursor Behavior

Although cursors are independent, how they are used may affect a message that another cursor is pointing to. For example, if two cursors are pointing at the same message and one cursor is used to remove the message, the other cursor will no longer point to a message. An error will be returned if an attempt is made to peek at or retrieve the message that was removed.

Note A cursor pointing to a message does not guarantee that the message will always exist. The message can be removed by another cursor, another application, by MSMQ Explorer, or the queue could be deleted.

The relationship between message position and cursor position can be simple or complex depending on how the messages in the queue are read. The following illustrations show several scenarios, starting with the simple case of retrieving the first message in the queue (using no cursor), and ending with an example of multiple cursors. Each example shows message and cursor position before and after the call was made.

Note Cursors work the same way for synchronous and asynchronous operations. However, you should not use the same cursor when firing receives in overlapping operations. Firing a second receive (using the same cursor) before the first one is completed will lead to unexpected behavior.

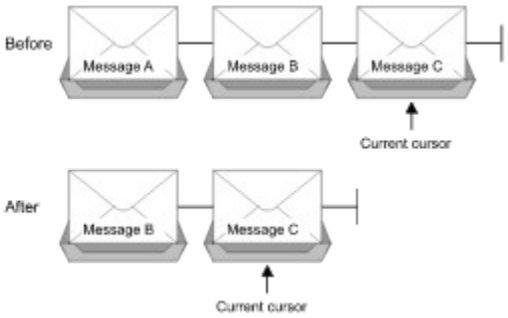
Retrieving the First Message

Here no cursor is used to remove Message A from the queue. The cursor shown in the illustration could be another application's cursor, or the implied cursor used by ActiveX applications.

Platform

C: **MQReceiveMessage**
dwAction==MQ_ACTION_RECEIVE
hCursor==NULL

ActiveX **MSMQQueue.Receive**



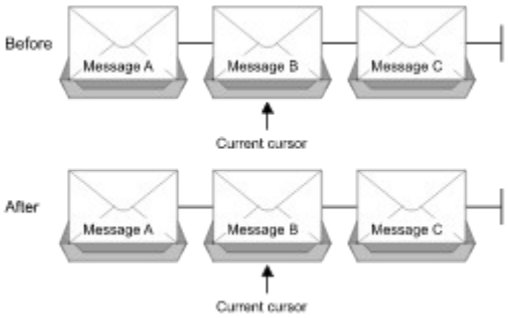
Peeking at a Message in a Queue

Here a cursor is used to look at the second message in the queue. The message is peeked at, but not removed from the queue. Cursor position and message position remain the same.

Platform

C: **MQReceiveMessage**
dwAction==MQ_ACTION_PEEK_CURRENT
hCursor==<CurrentCursorHandle>

ActiveX **MSMQQueue.PeekCurrent**



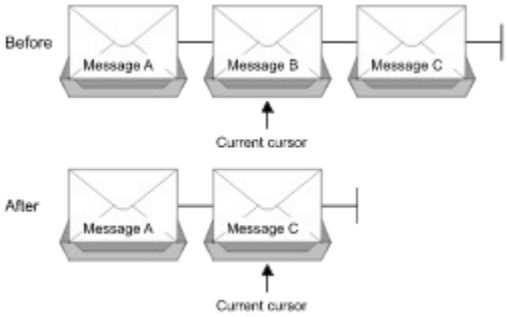
Retrieving a Message in a Queue

Here a cursor is used to retrieve the second message in the queue. When the call is made, the message is removed from the queue and the cursor now points at a new message. When the last message in a queue is retrieved, MSMQ points the cursor to the end of the queue and waits for a new message or a time-out to occur.

Platform

C: **MQReceiveMessage**
dwAction==MQ_ACTION_RECEIVE_CURRENT
hCursor==<CurrentCursorHandle>

ActiveX **MSMQQueue.ReceiveCurrent**



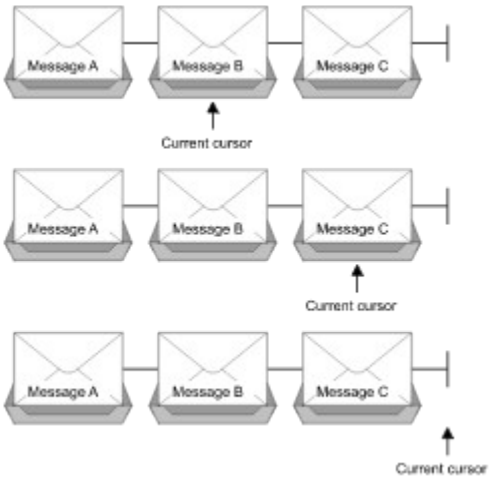
Peeking at the next Message in a Queue

Here a cursor is used to peek at the next two message in the queue. When the first call is made, MSMQ moves the cursor and then looks at the new message. When the second call is made (the cursor is pointing to the last message in the queue), MSMQ moves the cursor to the end of the queue and waits for a new message or a time-out to occur.

Platform

C: **MQReceiveMessage**
dwAction==MQ_ACTION_PEEK_NEXT
hCursor==<CurrentCursorHandle>

ActiveX **MSMQQueue.PeekNext**

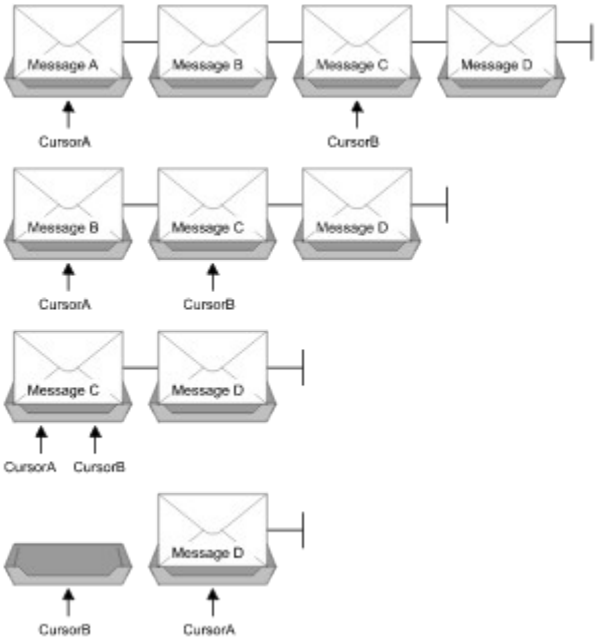


Using Multiple Cursors

Here one cursor is pointing at a message in the queue and another cursor is removing messages from the queue. CursorA removes messages from the queue with several receive-current calls while CursorB remains pointing to message “C.” After CursorA removes message “C,” CursorB no longer points to a message. CursorB will be pointing to a message place holder, and an error will be returned if the application tries to read a message at the current CursorB location. However, CursorB can still be used if peek-next is called.

Platform

C: **MQReceiveMessage**
dwAction==MQ_ACTION_RECEIVE_CURRENT
hCursor==<CurrentCursor1Handle>



Message Timers

MSMQ provides two timers to help you maintain better control of your messages: a *time-to-be-received* and a *time-to-reach-queue* timer.

The time-to-be-received timer determines how long a message remains in the system, starting from the time the message is sent to the time it is removed from the target queue.

The time-to-reach-queue timer determines how long a message has until it reaches the target Queue Manager of the target queue. Typically, this timer is set to a value less than the time-to-be-received setting.

When both timers are used, if the time-to-be-received timer is set to a shorter time interval than the time-to-reach-queue timer, it takes precedence over the time-to-reach-queue timer. MSMQ does not allow messages to remain in the system longer than the time allowed by their time-to-be-received timer.

When either timer expires, MSMQ discards the message. However, MSMQ can also send a copy of the message to a dead letter queue or an acknowledgment message to an administration queue. If the message's acknowledgment property specifies full or negative acknowledgments, MSMQ sends the appropriate negative acknowledgment message to the administration queue specified by the message. If the message's journal property specifies a dead letter queue, a copy of the message is sent to one of two places. The copies of non-transactional messages are sent to the dead letter queue on the computer where the timer expired. Copies of transactional messages are copied to the transactional dead letter queue on the source machine.

For more information see:

- Sending Messages To a Queue (using API functions)
- Sending Messages To a Queue (using ActiveX components)
- PROPID_M_TIME_TO_BE_RECEIVED
- PROPID_M_TIME_TO_REACH_QUEUE
- **MaxTimeToReceive**
- **MaxTimeToReachQueue**

Acknowledgment and Report Messages

MSMQ generates acknowledgment and report messages, which are sent to administration queues and report queues, respectively.

Note Acknowledgment and report messages can also be created by MSMQ Connector applications.

Acknowledgment messages are sent to the administration queue specified by the sending application. For more information, see Administration Queues.

Report messages are sent to the report queue created by the MSMQ Administrator. For more information, see Report Queues.

For a description of acknowledgment and report messages, see Acknowledgment Messages and Report Messages.

Acknowledgment Messages

An acknowledgment message indicates a positive or negative acknowledgment. The following table lists the message properties that have special meaning when attached to an acknowledgment message.

Note Connector applications must set these properties when sending positive (including read receipt messages) and negative acknowledgment messages back to the sending application.

Message Property (ActiveX property)	Setting
PROPID_M_ACKNOWLEDGE (**Ack**)	None.
PROPID_M_ADMIN_QUEUE (**AdminQueueInfo**)	Null.
PROPID_M_APPSPECIFIC (**AppSpecific**)	Same as original message.
PROPID_M_BODY (**Body**)	For positive acknowledgment messages, the body of the acknowledgment message is empty. You must refer to the original message to see the message body. For negative acknowledgment messages (with the exception of private, encrypted messages) the body of the acknowledgment message contains the message body of the original message.
PROPID_M_CLASS (**Class**)	Specifies the appropriate positive or negative acknowledgment class.
PROPID_M_CORRELATIONID (**CorrelationId**)	Specifies the original application message identifier (the PROPID_M_MSGID of the original message). If the correlation identifier of the original message contained an application-defined identifier, that information is not included in the acknowledgment message.
PROPID_M_DELIVERY (**Delivery**)	Same as original message.
PROPID_M_EXTENSION	Same as original message.
PROPID_M_JOURNAL (**Journal**)	Same as original message.
PROPID_M_LABEL (**Label**)	Same as original message.
PROPID_M_MSGID (**id**)	Message identifier for the acknowledgment message.
PROPID_M_PRIORITY (**Priority**)	Same as original message.
PROPID_M_RESP_QUEUE (**ResponseQueueInfo**)	Target queue of the original message.
PROPID_M_TIME_TO_BE_RECEIVED	Set to maximum time allowed by MSMQ.

(**MaxTimeToReceive**)

PROPID_M_TIME_TO_REACH_QUEUE Set to maximum time allowed by MSMQ. (**MaxTimeToReachQueue**)

Positive Acknowledgment Classes

MSMQ generates the following classes of positive acknowledgment messages:

- MQMSG_CLASS_ACK_REACH_QUEUE. This class indicates the original message reached its destination queue.
- MQMSG_CLASS_ACK_RECEIVE. This class indicates the original message was retrieved by the receiving application.

Negative Arrival Acknowledgment Classes

MSMQ generates the following classes of negative acknowledgment messages if the message fails to arrive at the queue:

- MQMSG_CLASS_NACK_ACCESS_DENIED. This class indicates the sender does not have send access rights to the destination queue.
- MQMSG_CLASS_NACK_BAD_DST_Q. This class indicates the destination queue is not available to the sender.
- MQMSG_CLASS_NACK_BAD_ENCRYPTION. This class indicates the destination Queue manager could not decrypt a private message.
- MQMSG_CLASS_NACK_BAD_SIGNATURE. This class indicates that MSMQ could not authenticate the original message. The original message's digital signature is not valid.
- MQMSG_CLASS_NACK_COULD_NOT_ENCRYPT. This class indicates the source Queue manager could not encrypt a private message.
- MQMSG_CLASS_NACK_HOP_COUNT_EXCEEDED. This class indicates the message hop count was exceeded. The maximum hop count is set by MSMQ (hop count = 15) and cannot be changed.
- MQMSG_CLASS_NACK_Q_EXCEED_QUOTA. This class indicates the message was not delivered because the destination queue is full.
- MQMSG_CLASS_NACK_REACH_QUEUE_TIMEOUT. This class indicates that the message did not reach the destination queue. It can be generated by either the time-to-reach-queue or time-to-be-received timer.
- MQMSG_CLASS_NACK_PURGED. This class indicates the message was purged before reaching the destination queue.
- MQMSG_CLASS_NACK_NOT_TRANSACTIONAL_Q. This class indicates a transaction message was sent to a non-transaction queue.
- MQMSG_CLASS_NACK_NOT_TRANSACTIONAL_MSG. This class indicates a non-transaction message was sent to a transaction queue.

Negative Read Acknowledgment Classes

MSMQ generates the following classes of negative acknowledgment messages if the original message in the queue could not be read:

- MQMSG_CLASS_NACK_Q_DELETED. This class indicates the queue was deleted before the message could be read from the queue.
- MQMSG_CLASS_NACK_Q_PURGED. This class indicates that the queue was purged and the message no longer exists.
- MQMSG_CLASS_NACK_RECEIVE_TIMEOUT. This class indicates that the message was not retrieved from the queue before its time-to-be-received timer expired.

For an example of returning acknowledgment messages, see:

- Sending Messages that Request Acknowledgments (using API functions)
- Sending Messages that Request Acknowledgments (using ActiveX components)

Report Messages

Report messages are used to trace the path of the message to its target queue. Each report message has its own message identifier, a specific report message class, a message label, and a message body.

MSMQ generates the following report message class:

MQMSG_CLASS_REPORT, used for typical report messages.These report messages are sent to the source Queue Manager as soon as your message enters or exits a source, target, or intermediate queue along its route. MSMQ generates one report message when the message leaves the source Queue Manager, two report messages for each MSMQ server it enters and leaves, and one report message when your message reaches its destination Queue Manager.

The report message label contains a simplified version of the information in the body of the message. Although it can be used by any MSMQ application, it is primarily used for display purposes by the MSMQ Explorer. The format of the message label is one of the following:

gggg:dddd:hh sent from <computer> to <address> at <time>
gggg:dddd:hh received by <computer> at <time>

where "gggg" is the first four hexadecimal digits of the source queue identifier, "dddd" is an internal message identifier, and "hh" is the hop count. The internal message identifier is a 20-byte identifier, consisting of the first 16 bytes of the original message's identifier plus a 4-byte sequence number.

The body of a report message contains detailed information about the original message. The format of the message body is:

<MESSAGE ID>{message ID}</MESSAGE ID>
<TARGET QUEUE>{target queue format name}</TARGET QUEUE>
<NEXT HOP>{IP or IPX address of the next hop}</NEXT HOP>
<HOP COUNT>integer</HOP COUNT>

For information on report queues, see Report Queues.

MSMQ Computers

MSMQ computers are defined as machine objects within the MSMQ enterprise. Machine objects are created and maintained by the MSMQ administrator via the MSMQ Explorer and cannot be created, modified, or deleted using the MSMQ SDK.

Applications can retrieve the following properties of existing machine objects:

Property	Description
PROPID_QM_CONNECTION	Identifies the Connected Network (CN) list of the computer.
PROPID_QM_ENCRYPTION_PK	Identifies the public encryption key of the computer.
PROPID_QM_MACHINE_ID	Identifies the computer.
PROPID_QM_PATHNAME	Identifies the MSMQ pathname of the computer.
PROPID_QM_SITE_ID	Identifies the MSMQ site where the computer is defined.

These properties are retrieved by calling **MQGetMachineProperties**. For example, if the computer's identifier is needed for the format name of a queue, specify PROPID_QM_MACHINE_ID in the MQGetMachineProperties call to retrieve the computer's identifier. Once the call returns the identifier, it can be translated to a string and included in the format name of the queue.

MSMQ Object Properties

MSMQ objects, including computers, public and private queues, and messages, are all defined by their properties.

Properties can be stored in MQIS, on a local computer, or passed along with their object. Properties for computers (machine properties) are stored in MQIS. Properties for queues are either stored in MQIS (for public queues), or on the local computer where the queue resides (for private queues). Properties for messages are passed along with the message.

For information on the property structures, see Property Structures.

Property Structures

MSMQ uses three property structures: **MQQUEUEPROPS** for queue properties, **MQMSGPROPS** for message properties, and **MQQMPROPS** for Queue Manager properties. All three structures have the following four members:

- A count (**cProp**), indicating how many properties are supplied. This is a double word member field (DWORD) included in all three property structures.
- An array of **PROPID** values (**aPropID**) identifying which properties are specified for the call. MSMQ uses three different property identifiers: **QUEUEPROPID**, **MSGPROPID**, and **QMPROPID**. These identifiers are used for MSMQ queue properties, message properties, and Queue Manager properties, respectively. These identifiers are all of type **PROPID**.
- An array of **PROPVARIANT** structures (**aPropVar**) containing the values of the properties. Position **i** in this array is the value of the property whose identifier (**PROPID** value) is in position **i** in its respective **aPropID** array.
- An array of **HRESULT** values (**aStatus**) returned by MSMQ. Position **i** in this array is a reported status code of the property whose identifier and value are in position **i** in the arrays discussed earlier. This array is optional.

For information on the specific property structures, see **MQQUEUEPROPS**, **MQMSGPROPS**, and **MQQMPROPS**.

Setting Properties

To set the properties of a queue or message, the corresponding elements of the **aPropID** array and the **aPropVar** array must be set. Element **i** of the arrays must contain the property's identifier and value, respectively.

When setting properties, if a property is set to the same value more than once, MSMQ uses the first entry in the **aPropVar** array and discards all the subsequent entries. In the **aStatus** array, an information error value of MQ_INFORMATION_DUPLICATE_PROPERTY is returned to each discarded entry.

For information on setting queue properties, see Setting Queue Properties.

For information on setting message properties, see Setting Message Properties.

Unlike queue and message properties, you cannot set Queue Manager properties; they are properties of the Queue Manager machine. The MSMQ API can only retrieve Queue Manager properties (see **MQGetMachineProperties**).

Setting Queue Properties

The following list demonstrates what must be done to set a queue's properties.

- Define a queue property structure.
- For each property:
  - Set element **i** of **aPropID** to the property's identifier.
  - Set element **i** of **aPropVar** to the property's value. Each element of **aPropVar** is an instance of **PROPVARIANT**. Set the **vt** field of **PROPVARIANT** to the property's type indicator and the value field (for example, **bVal**) to the appropriate property value.
- If a status code for this property is needed, include an **aStatus** array.
- Set **cProp** to the number of properties specified by the queue property structure.
- Call the appropriate function with the queue property structure.

Setting Message Properties

The following list demonstrates what must be done to set a message property.

- Define a message property structure.
- For each property:
  - Set element **i** of **aPropID** to the property's identifier.
  - Set element **i** of **aPropVar** to the property's value. Each element of **aPropVar** is an instance of **PROPVARIANT**. Set the **vt** field of **PROPVARIANT** to the property's type indicator and the value field (for example, **bVal**) to the appropriate property value.
- If a status code for this property is needed, include an **aStatus** array.
- Set **cProp** to the number of properties specified by the queue property structure.
- Call the appropriate function with the queue property structure.

Property Values

A property structure often contains three types of properties: IN properties, OUT properties, and IN/OUT properties.

- IN properties use values set by the application. They are passed to MSMQ as IN parameters and can either identify or set queue or message properties. For example, the application must supply the MSMQ pathname for the queue when the queue is created.
- OUT properties use values set by MSMQ. For example, when a queue is created, MSMQ sets the time when it was created.
- IN/OUT properties initially use values passed to MSMQ, then MSMQ resets the value when the property is passed back to the application. For example, when the receiving application asks for the destination queue of a message, it must first tell MSMQ the size of the buffer allocated for the destination queue's format name. On return, MSMQ passes back the actual size of the format name.

Properties appear in no particular order in the **aPropVar** array. To determine the type of a property, MSMQ examines the property identifier and the function called. For example, for API functions, the queue quota property (PROPID_Q_QUOTA) is an OUT property when **MQGetQueueProperties** is called, and an IN property when **MQSetQueueProperties** is called.

IN properties can be set to any valid setting for the specific property. However, VT_NULL cannot be used as an IN property **VARTYPE** value.

OUT properties returned by MSMQ require an **PROPVARIANT** entry where the returned value can be stored. For example, space must be allocated before the label of a queue or the body of a message can be specified. There are two ways for the application to specify such an **PROPVARIANT** entry:

- Prepare a **PROPVARIANT** entry with the appropriate **VARTYPE** value (**vt** field) for the expected value, allocating the corresponding buffer when appropriate. For example, if a message body is specified, allocate the array of bytes to be pointed to by the caui1.pElems field.
- Let MSMQ fill in the **PROPVARIANT** with the appropriate **VARTYPE** value. To do this, set the **vt** field to VT_NULL. If memory is required, MSMQ will allocate the buffer for the field as long as its value is stored in the PROPVARIANT structure (for example, VT_LPWSTR).

  For example, if myQueueProps is defined as an **MQQUEUEPROPS** structure to be passed to **MQGetQueueProperties**: Assign myQueueProps.aPropID[j]=PROPID_Q_LABEL and myQueueProps.aPropVar[j].vt=VT_NULL. When the call returns, myQueueProps.aPropVar[j]. pwszVal will point to the buffer that was allocated by MSMQ that contains the queue label.

For queue and queue manager properties whose field type is VT_LPWSTR, their **VARTYPE** must be set to VT_NULL.

In all cases where MSMQ allocates a buffer for the MSMQ application, it is the application's responsibility to free the memory with the **MQFreeMemory** function.

MSMQ Transactions

The following illustration shows how transactions are used by the sending and receiving application. In this model, MSMQ uses two transactions, one to send messages to the queue and the other to retrieve messages from the queue.

In this model, the sending transaction can commit to sending the messages to the queue and the receiving application can commit to retrieving the messages; MSMQ provides its own confirmation process to notify the sending application that either the messages were retrieved from the queue or why the receiving application failed to retrieve them.

Note A single transaction can contain both transaction send and receive calls.

MSMQ provides several ways to send and receive messages through transactions. Transactions can be called either explicitly by providing pointers to a transaction object, or implicitly, directing MSMQ to find the appropriate transaction object.

Transactions that can be explicitly called include MSMQ internal transactions and the Microsoft® Distributed Transaction Coordinator (MS DTC) external transactions. For information on these transactions, see:

- MSMQ Internal Transactions
- MS DTC External Transactions

Transactions that can be implicitly called include the current Microsoft® Transaction Server™ (MTS) and current XA transactions. The current MTS or XA transaction is only used if it is available. In these cases, MSMQ decides (with the help of MTS or MS DTC) whether the call will truly be part of a transaction. For information on these transactions, see:

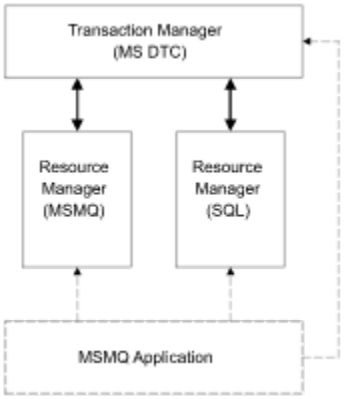
- MTS Transactions
- XA-Compliant Transactions

MS DTC External Transactions

Microsoft® Distributed Transaction Coordinator (MS DTC) external transactions are used when the transaction includes more actions than simply sending or retrieving MSMQ messages (more than one resource manager is used). In this case, the application must ask MS DTC for a transaction object and explicitly reference that object each time it sends a message, retrieves a message, or executes an action of another resource manager.

When an application is performing an MS DTC transaction, MSMQ is acting as part of a transaction processing system that includes a transaction manager and any number of resource managers.

The following illustration shows the basic model for the transaction processing system. This model demonstrates a typical system with MS DTC as the transaction manager and MSMQ and SQL as resource managers. The application must work with the transaction manager, and with each relevant resource manager.



For an example of an MS DTC external transaction, see:

- Sending Messages Using an MS DTC External Transaction (using API functions)
- Sending Messages Using an MS DTC External Transaction (using ActiveX components)

MSMQ Internal Transactions

MSMQ internal transactions provide better performance for transactions that only send or receive MSMQ messages.

Unlike MS DTC external transactions, MSMQ internal transactions cannot be passed to another resource manager. It is the cost of coordinating between several resource managers that make MSMQ internal transaction less expensive in terms of memory than MS DTC external transactions.

For information on the SDK components used to create an MSMQ internal transaction, see:

- **MQBeginTransaction** (using API functions)
- **MSMQTransactionDispenser** (using ActiveX components)

Note When sending a single message, MSMQ provides a single-message send operation that uses an MSMQ internal transaction. This mode of sending message provides the best performance of all transaction types. When using this mode, **MQBeginTransaction** and **Commit** are implied.

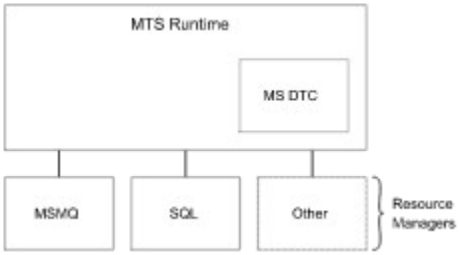
For an example of an internal transaction, see:

- Sending Messages Using an Internal Transaction (using API functions)
- Sending Messages Using an Internal Transaction (using ActiveX components)

MTS Transactions

When the application is running in the Microsoft® Transaction Server (MTS) environment, MSMQ can use the current MTS transaction if one is available. For information on MTS see the Microsoft® Platform SDK.

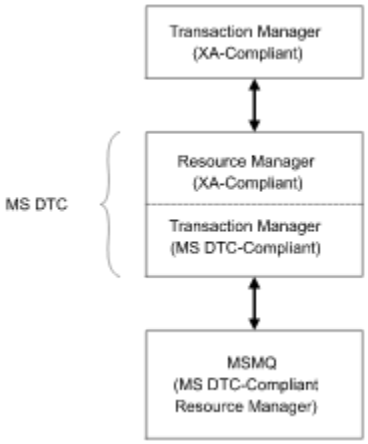
MSMQ links directly to the MTS runtime environment. Within the runtime environment, MTS uses the services of Microsoft® Distributed Transaction Coordinator (MS DTC) for transaction coordination.



When this type of transaction is used, the transaction object is provided by MTS.

XA-Compliant Transactions

MSMQ can work in transaction systems that have an XA-compliant transaction manager. Because of this MS DTC is used as a transaction process monitor and MSMQ is a resource manager.



In the preceding transaction system model, MS DTC acts as a resource manager and a transaction manager. It acts as an XA-compliant resource manager with the transaction manager above it, and as an MS DTC-compliant transaction manager to the MSMQ resource manager below it.

Error Reporting

MSMQ can report errors for the properties an application passes to a call, as well as an error for the call itself. When an error occurs, MSMQ generates an error value that is returned to the application.

For properties, error values are returned to the application in the optional **aStatus** array of **MQQUEUEPROPS**, **MQMSGPROPS**, and **MQQMPROPS**. The **aStatus** array is always optional: if the application specifies a NULL array in the property structure, MSMQ will not report errors to the application.

If the application does supply a status array when an error condition is encountered at index **n** of the property array (such as an illegal property tag or insufficient access rights to set a property), MSMQ stores the appropriate error value in entry **n** of the status array (if no error was encountered with this property, MSMQ stores the value MQ_OK). This means that if the application passes a status array, it must be the same size as the list of properties.

The returned value of any call corresponds to the highest-severity error it encounters. For example, if the error is caused by a property, MSMQ can either return an error code or an information code. To determine which property caused the error (and why), the application must examine the property status array.

For a list of error and information codes, see MSMQ Error and Information Codes. For information on the error codes returned by a specific call, refer to the appropriate API function or ActiveX method reference page.

MSMQ Offline Support

MSMQ supports offline operations for applications running on independent client computers. While the client computer is offline, the application can still send and read messages from local queues. Messages sent while a computer is offline are stored locally, then sent to their final destination when the client computer is reconnected to the network.

Note Dependent client computers cannot be used for offline operations.

When operating offline, the application must avoid using any call that would need to access the MQIS. Any call to MQIS would attempt to generate network activity, causing an error or timeout.

The table below lists all the functions and methods that access the MQIS and cannot be called when operating offline.

API Functions	ActiveX Methods
MQCreateQueue (for public queues)	**Create** (for public queues)
MQDeleteQueue (for public queues)	**Delete** (for public queues)
MQGetMachineProperties	**MachineIdOfMachineName**
MQGetQueueProperties	**Refresh**
MQGetQueueSecurity	**IsWorldReadable**
MQGetSecurityContext	**AttachSecurityContext**
MQLocateBegin	**LookupQueue**
	Reset
MQLocateNext	**Next**
MQOpenQueue (for public queues opened to receive messages)	**Open** (for public queues opened to receive messages)
MQPathNameToFormatName	
MQSetQueueProperties	**Update**
MQSetQueueSecurity	

Opening Queues Offline

Queues can be opened offline using public, private, or direct format names. Once the queue is open, all messages sent to the queue are stored locally by the client computer's Queue Manager, then passed on to the destination queue when the client computer is reconnected to the network.

Caution Messages must be sent in recoverable mode if the offline client computer is to be turned off. Messages sent in express mode are held in RAM and will be lost when the computer is turned off. To send recoverable messages, set the delivery property (PROPID_M_DELIVERY or **MSMQMessage.Delivery**) of the message to MQMSG_DELIVERY_RECOVERABLE.

Public queues cannot be opened to retrieve or peek at messages while off line.

Public Format Names

To use a public format name, the identifier of the queue must be known by the application before the computer is disconnected from the network.

Here is the syntax of a public format name:

"PUBLIC=QueueGUID"

In this format, the client computer accesses MQIS after the computer is reconnected to the network. MQIS resolves the queue identifier, then passes the messages to the appropriate queue.

Private Format Names

To use a private format name, the identifier of the client computer and the queue's name must be known.

Here is the general format of a private format name:

"PRIVATE=MachineGUID\QueueNumber"

Direct Format Names

To use a direct format name, the target computer's network address and the queue's name must be known.

Here is the general format of a direct format name (public and private queues can be accessed directly):

DIRECT=AddressSpecification\QueueName (For public queues.)
DIRECT=AddressSpecification\PRIVATE$\QueueName (For private queues.)

When using a direct format name, the messages are sent directly to the target computer as soon as the client computer is brought back online. (For more information on this format see Direct Format Names.

Opening a Queue

For information on opening a queue, see **MQOpenQueue** or **MSMQQueueInfo**.**Open**

MSMQ Security Services

The MSMQ security services provide MSMQ applications with the following security features:

- Digital signatures for authenticating messages (see Message Authentication)
- Security descriptors for access control to queues (see Access Control)
- Encryption for private messages (see Private Messages)
- Audit messages for tracking the operations performed by MSMQ (see Auditing)

Message Authentication

Message authentication allows the receiving application to verify the source of a message and that the message was not modified on its way to the queue. This is done by attaching a digital signature to the message when it is sent, then verifying the digital signature when the message reaches the queue. The receiving MSMQ Queue Manager uses the digital signature to verify the sender and that the message was not modified.

To digitally sign a message, the sending application uses a public and private signing key pair to create the digital signature. MSMQ provides the key pair when an internal security certificate is used or when an external security certificate is used. External certificates are obtained from a certificate authority (CA).

When an internal security certificate is used, the private signing key is registered the first time that the MSMQ Control Panel application is run. The public signing key is provided within the internal certificate.

Internal certificates are used when the receiving application needs to validate the sender identifier attached to a message. When using an internal certificate, only the sender identifier is guaranteed correct.

External certificates are used when you want to use the information in the certificate (not just the sender identifier sent with the message) to verify the source of a message. The information in the external certificate is guaranteed by the certificate authority that created the certificate.

MSMQ does not validate an external certificate. The receiving application must validate the certificate before using an authenticated message. MSMQ generates the digital signature of a message when it is sent and verifies the digital signature when the message is received, but does not validate the certificate itself.

Note External certificates are required when communicating with operating environments other than Windows NT® where the sender identifier is meaningless.

For information on using an internal certificate, see Authenticating Messages Using an Internal Certificate.

For information on using an external certificate, see Authenticating Messages Using an External Certificate.

How MSMQ Authenticates Messages

MSMQ authenticates message at the request of the sending application. When the sending application indicates it wants a message authenticated, the MSMQ runtime code performs the following tasks:

- Retrieves the internal certificate, external certificate certificate, or the security context information.

  Note For applications using API functions, external certificates are provided in PROPID_M_SENDER_CERT and security context information is provided in PROPID_M_SECURITY_CONTEXT

  For applications using ActiveX components, external certificates are provided in SenderCertificate and security context information is retrieved by AttachCurrentSecurityContext.

- Extracts the public signing key from the internal certificate, external certificate, or from the security context information taken from the external certificate.
- Extracts the matching private signing key. For internal certificates, MSMQ locates the private signing key internally. For external certificates, MSMQ searches for the private signing key in the Internet Explorer certificate store (this is why the external certificate must be registered in the certificate store).
- Computes a hash value of the message using the algorithm specified by the sending application.

  Note For applications using API functions, the Hash algorithm is specified by PROPID_M_HASH_ALG (the default algorithm is CALG_MD5)

  For applications using ActiveX components, the Hash algorithm is specified by the **MSMQMessage** object's **HashAlgorithm** property (the default algorithm is CALG_MD5)

- The following message properties (in the order shown here) are used to create the hash value:

API Functions	ActiveX Components
PROPID_M_CORRELATIONID	**CorrelationId**
PROPID_M_APPSPECIFIC	**AppSpecific**
PROPID_M_BODY	**Body**
PROPID_M_LABEL	**Label**
PROPID_M_RESP_QUEUE	**ResponseQueueInfo**
PROPID_M_ADMIN_QUEUE	**AdminQueueInfo**

- Encrypts the hash value using your private signing key. This is the digital signature that will be attached to the message.
- Attaches the certificate and digital signature to the message, then sends the message on to the target Queue Manager.

When the target Queue Manager receives the message, it performs the following tasks:

- Computes the hash value of the message using the algorithm specified in PROPID_M_HASH_ALG or **HashAlgorithm**.
- Extracts the public key from the certificate.
- Decrypts the digital signature using the public key, obtaining the hash value sent with the message.
- Compares the hash value computed by the Queue Manager to the hash value decoded from the digital signature.
- If the hash values are the same, the queue then verifies the sender identifier, setting PROPID_M_AUTHENTICATED or **IsAuthenticated** to 1 if the sender identifier is valid.

  The sender identifier stored with the certificate is retrieved from the MQIS and compared with the

message's PROPID_M_SENDERID or **SenderId** property. (This is why the certificate should be registered with MSMQ.)

- If the hash values are not the same, the message is discarded and a negative acknowledgment is returned to the sending application if one was requested.

Note MSMQ does not validate the external certificate. The receiving application performs any validation requirements on the certificate before using an authenticated message. MSMQ generates the digital signature of a message when it is sent and verifies the digital signature when the message is received, but does not validate the certificate itself.

Access Control

Operations on queues can be restricted to a specific user or group of users. When a queue is created, a security descriptor is included in the call to specify who has access rights to the queue's operations.

Queue operations that can be set include creating, deleting, and opening a queue (for sending messages to and reading messages from the queue). Operations also include getting and setting a queue's properties and security descriptor.

For applications using API functions, the security descriptor is specified by the *lpSecurityDescriptor* parameter of **MQCreateQueue**. For applications using ActiveX components, the default security descriptor is automatically attached to the queue when it is created and can only be changed using API functions.

Before MSMQ performs any operation on a queue, it checks the queue's security descriptor to determine if the user has sufficient access rights to perform the requested operation. To do this, MSMQ checks whether the operation is restricted. If the operation is restricted, MSMQ then checks the identity of the user to see if the restriction applies to that user. If it does, the operation is not allowed to continue.

With the exception of putting a new message in a queue, MSMQ can verify the identity of the user by the access token attached to the process. Access tokens are produced by the system. When a user logs on, the system verifies the user's password by comparing it with information stored in the system's security database. If the password is valid, the system produces an access token and attaches it to each process started by the user.

However, MSMQ cannot use this access to put a new message in the queue. Instead, it uses a security identifier (SID) that MSMQ attached to the message when it was sent. Similar to the access token, the user's SID is created by the application. For a description of access tokens and SIDs, see the Security section in the Microsoft Platform SDK.

Applications can retrieve or modify the security descriptor of a queue if they have sufficient access rights. See **MQGetQueueSecurity** and **MQSetQueueSecurity**.

Private Messages

MSMQ provides a secured channel for sending private, encrypted messages throughout your MSMQ enterprise. MSMQ ensures that the body of private messages are kept encrypted from the moment they leave the source Queue Manager to the moment they reach their target Queue Manager.

Note Private messages can also be sent to foreign queues via an MSMQ connector server. For information on sending private message to foreign queues, see Passing Private Messages.

With encryption and decryption provided by MSMQ Queue Managers, applications do not have to encrypt messages when they are sent or decrypt messages when they are received. When a private message is sent, the source Queue Manager encrypts the body of the message, then sends the message on to the target Queue Manager. When the target Queue Manager receives the message, it decodes the body of the message and passes the clear message on to the queue. The receiving application can then read the message from the queue without ever knowing it was encrypted.

Note Even though the receiving application sees the message as clear text, it can look at the message's privacy level to determine whether the message was sent encrypted, or look at the encryption algorithm used when the message was sent.

To send a private message, the sending application sets the privacy level of the message and the encryption algorithm. The default encryption algorithm is RC2 (message encryption is based on public-key encryption using the Microsoft® Cryptographic API with an underlying RSA provider).

Note In addition to setting the privacy level of a message, the privacy level of a queue can also be set so that the queue only accepts private messages.

For a complete example of sending a private message (including setting the privacy level of a queue) see:

- Sending Private Messages (using API functions)
- Sending Private Messages(using ActiveX components)

For a description of the properties used to set the privacy level of a message, see:

- PROPID_M_PRIV_LEVEL
- PROPID_M_ENCRYPTION_ALG
- **PrivLevel**
- **EncryptAlgorithm**

Auditing

MSMQ allows you to audit access operations for your MSMQ enterprise, sites, connected networks (CNs), computers, and queues.

For the most part, auditing is set up and maintained by the MSMQ Explorer (for a complete description of auditing, see the *Microsoft Message Queue Server Administrator's Guide*). However, it is possible to audit queue operations by modifying the system access control list (SACL) of the queue's security descriptor.

The following queue operations can be audited:

- MQSEC_DELETE_MESSAGE. When combined with MQSEC_PEEK_MESSAGE, the user can retrieve messages from the queue. MSMQ does not explicitly delete messages that are in queues. When a receive operation is requested, MSMQ peeks at the message then deletes it from the queue.
- MQSEC_DELETE_JOURNAL_MESSAGE. When combined with MQSEC_PEEK_MESSAGE, the user can retrieve messages from a journal queue. MSMQ does not explicitly delete messages that are in queues. When a receive operation is requested, MSMQ peeks at the message then deletes it from the queue.
- MQSEC_PEEK_MESSAGE. The user can look at (peek at) messages from a queue. Messages cannot be removed.
- MQSEC_GET_QUEUE_PROPERTIES. The user can retrieve the queue's properties.
- MQSEC_SET_QUEUE_PROPERTIES. The user can set the queue's properties.
- MQSEC_DELETE_QUEUE. The user can delete the queue (equivalent to DELETE: as defined in the Win32 header files).
- MQSEC_GET_QUEUE_PERMISSIONS. The user can retrieve the queue's security descriptor (equivalent to READ_CONTROL: as defined by the Win32 header files).
- MQSEC_CHANGE_QUEUE_PERMISSIONS. The user can modify the discretionary access control list (DACL) of the queue's security descriptor (equivalent to WRITE_DAC: as defined by the Win32 header files).
- MQSEC_TAKE_QUEUE_OWNERSHIP: The user can change the queue's owner in the queue's security descriptor (equivalent to WRITE_OWNER: as defined by the Win32 header files).

Audit log messages are written in the event log on the server that performs the actual operation, not necessarily the server that owns the object. For example, audits for opening a queue are logged on the computer where the queue resides. However, other operations (such as setting queue properties) are logged on the machine that performed the operation. As a result, the audit messages for a queue can be logged on servers throughout your MSMQ enterprise.

Note The send operation cannot be audited.

For applications using MSMQ API functions, call **MQSetQueueSecurity** to modify the queue's security descriptor.

MSMQ Connector Server

The MSMQ connector server is not available with the Windows NT® 4.0 Option Pack.

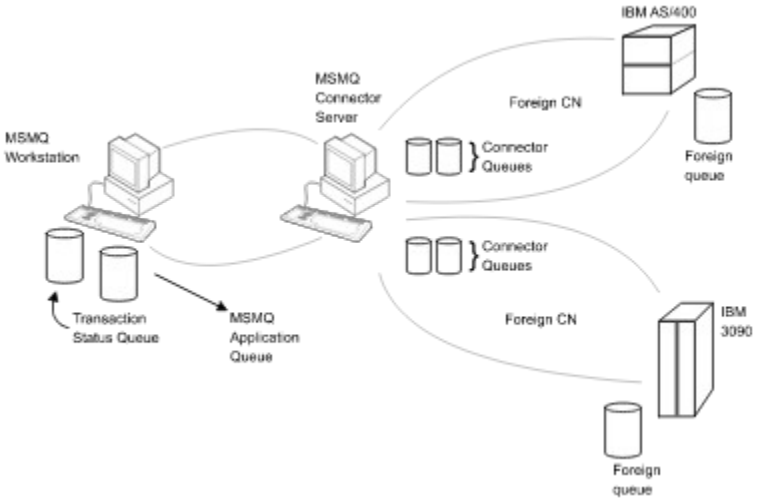
The MSMQ connector server provides a mechanism for MSMQ applications to send and receive message from computers using other messaging systems. To do this, MSMQ connector servers use internal connector queues to receive the messages from MSMQ and a connector application to translate messages between MSMQ and the other message system.

By using an MSMQ connector server, MSMQ applications can perform the same operations on foreign queues that they would typically perform on the queues within their enterprise.

Note Foreign queues, foreign computers, and foreign Connected-Networks (*CNs*) are all created and used the same way their MSMQ counterparts are. For information on creating foreign computers and CNs, see the *MSMQ Administrator's Guide*.

The MSMQ connector server uses a pair of internal connector queues for each foreign CN attached to the connector. One is used for transactional messages and the other for non-transactional messages. Connector queues are implicitly created by MSMQ. They are not registered in the MQIS and they are transparent to the MSMQ application sending messages to the foreign queue. For applications, sending a message to a foreign queue is identical to sending a message to an MSMQ queue.

The following illustration shows a single MSMQ server connected to two foreign CNs. In this case, there are two pairs of connector queues, one pair for each foreign CN.



MSMQ Connector Applications

The MSMQ connector server is not available with the Windows NT® 4.0 Option Pack.

The connector application must be able to translate between MSMQ and the message queue system used by the foreign computer. It must be able to read the message properties in one system, translate their values into the format of the other system, then send the message on (with its new properties) to the appropriate destination.

When an MSMQ application sends a message to a foreign queue, the message is routed to the connector queue that represents the foreign CN where the queue's computer resides. The connector application must read the messages in the connector queue, translate them, then forward the message on to the foreign queue.

When a message is sent to MSMQ it must be routed to the connector application, translated into an MSMQ message, then forward the message on to the appropriate MSMQ queue.

Connector Application Responsibilities

The MSMQ connector server should be transparent to the applications running on MSMQ computers. The connector application running on the server, must be able to:

- Handle the time-to-be-received timer. The time-to-reach-queue timer stops when the message reaches the connector queues.
- Handle the acknowledgment level of messages. The connector must return the appropriate acknowledgment messages to the appropriate administration queue according to the message properties.
- When the connector application creates an acknowledgment message, it must set the PROPID_M_CLASS and PROPID_M_CONNECTOR_TYPE properties when it sends the acknowledgment back to the sending application.
- Complete any pending transaction status for transacted messages. For information on transactions with foreign queues, see Using the MSMQ Connector in a Transaction.
- Handle security issues appropriately. See Connector Application Security for information on handling encrypted messages and authenticating messages.

Moving Messages from MSMQ to another Message Queue System

To move messages to another message queue system, the connector application must read all the messages that arrive in the connector queues. All transactional messages sent to foreign queues are stored in the transactional connector queue. All non-transaction messages are stored in the non-transactional connector queue. Non-transactional and transactional messages must be read from the appropriate connector queue.

To open these queues, the connector application must use a specific format name.

The following example shows two **MQOpenQueue** calls: the first one opens the connector's non-transaction queue and the second opens the transaction queue.

```
MQOpenQueue("CONNECTOR=ForeignCNGUID",       //Special format name.
    MQ_RECEIVE,                               //Receive access.
    TRUE,                                    //Read Exclusive.
    &qh                                      //Returned queue handle.
    )


MQOpenQueue("CONNECTOR=ForeignCNGUID;XACTONLY",   //Format name.
    MQ_RECEIVE,                                    //Receive access.
    TRUE,                                         //Read Exclusive.
    &qh                                           //Returned handle.
    )
```

An application can find the ForeignCNGUID from the Foreign CN name by calling **MQGetMachineProperties** and retrieve PROPID_QM_CONNECTION.

Moving Messages from another Message Queue System to MSMQ

To move messages from another message queue system to MSMQ, the connector application must use the standard MSMQ API functions.

To return acknowledgment messages to MSMQ, the connector application must set PROPID_M_CONNECTOR_TYPE. By setting this property, the application receiving the message knows that it was not sent by MSMQ.

Message Property Values

There are several message properties that have a specific meaning when they are sent to a foreign queue. The following table lists these properties and provides a brief description of any special circumstances.

Property	Description
PROPID_M_ADMIN_QUEUE	Needed for acknowledgments and computing the digital signature for the message.
PROPID_M_APPSPECIFIC	Needed for computing the digital signature for the message.
PROPID_M_BODY	Specifies message. Needed for decrypting private (encrypted) messages and computing the digital signature for the message.
PROPID_M_CONNECTOR_TYPE	Specifies the connector that was used to send message to MSMQ. Not used when sending messages to foreign queues.
PROPID_M_CORRELATIONID	For acknowledgment messages, specifies the message identifier of the original message. Also, needed for computing the digital signature for the message.
PROPID_M_DEST_SYMM_KEY	Needed for decrypting private (encrypted) messages. When messages are sent to MSMQ queues, the symmetric key is ignored if a connector is not specified.
PROPID_M_DEST_SYMM_KEY_LEN	Needed for security. Specifies the length of the symmetric key.
PROPID_M_ENCRYPTION_ALG	Needed for security. Specifies the encryption algorithm used to encrypt the symmetric key.
PROPID_M_EXTENSION	Specifies any foreign message properties that have no counterpart in MSMQ. MSMQ includes this property when returning acknowledgment messages.
PROPID_M_EXTENSION_LEN	Specifies the length of the foreign message properties.
PROPID M HASH ALG	Needed for security. Specifies the hash algorithm used when authenticating messages.
PROPID_M_LABEL	Needed for computing the digital signature for the message.
PROPID_M_PROV_NAME	Cryptographic provider needed to verify signature.
PROPID_M_PROV_TYPE	Cryptographic provider needed to

	verify signature.
PROPID_M_RESP_QUEUE	Needed for computing the digital signature for the message.
PROPID_M_SENDER_CERT	Includes public key.
PROPID_M_SIGNATURE	Signature of sender.

Connector Application Security

The MSMQ connector server is not available with the Windows NT® 4.0 Option Pack.

Security operations can be performed by the connector application or they can be passed on to the foreign computer where the destination queue is located.

Applications that pass security operations to the foreign queue are referred to as transparent applications. As a transparent application, the connector application translates the message properties so they can be understood by the foreign computer, then passes the translated message properties on to their destination. In this case, the foreign computer must interpret the message properties and perform any required actions.

Applications that perform security operations are referred to as non-transparent applications. As a non-transparent application, an application receives messages from either MSMQ or the other message queue system, interprets the message's properties and performs any required actions. Then the application sends the message on with the appropriate message properties. In this case, the foreign queue must be able to trust the connector application to perform the correct actions.

Note Tasks for non-transparent applications are also relevant to message systems that use a transparent connector applications. Even though the transparent application does not perform them at the server, these security tasks still must be done when the message reaches the foreign queue.

When translating message properties, transparent and non-transparent connector applications must use the following message properties as described below.

Property	Description
PROPID_M_ADMIN_QUEUE	Part of signature.
PROPID_M_APPSPECIFIC	Part of signature.
PROPID_M_BODY	Part of signature.
PROPID_M_CORRELATIONID	Part of signature.
PROPID_M_HASH ALG	Algorithm used to create signature.
PROPID_M_LABEL	Part of signature.
PROPID_M_PROV_NAME	Cryptographic provider needed to verify signature.
PROPID_M_PROV_TYPE	
PROPID_M_RESP_QUEUE	Part of signature.
PROPID_M_SENDER_CERT	Includes public key.
PROPID_M_SIGNATURE	Signature of sender.

In the preceding list, several properties are used when creating the signature of the sender. When a transparent connector application translates these properties (in particular the administration and response queue properties) to their new values, it must include both the translated and original values when it passes the message on to its destination. The foreign application will need the original values to authenticate the signature when the message arrives.

Providing the necessary information is much more difficult for messages being sent to an MSMQ queue than messages sent to a foreign queue. In this case, the foreign application must retrieve an MSMQ-representation of the signature properties before it creates the signature.

Passing Authenticated Messages

To pass authenticated messages between MSMQ and another message queue system, the connector application (both transparent and non-transparent applications) must retrieve all the properties needed for authentication.

Transparent connector applications must perform different operations depending on the direction of the messages. If the message is going from the foreign application to MSMQ, the connector application needs to translate new property values and pass the new values on to MSMQ. If the messages are being sent from MSMQ to the other message queue system, the connector application must translate new property values, then pass both them and the original values on to the foreign application. The original values are needed to generate the hash value used to authenticate the signature.

Non-transparent applications perform the security operations (verify signature) so there is no need to pass on security properties. However, the applications do need to indicate that the message was verified when it passes the message on.

The code needed to perform security operations varies for each application. However, the pseudo-code provided in the following sample shows the basic elements needed to authenticate a message.

When messages are being sent from another message queue system to MSMQ, non-transparent applications must have access to the private signing keys of all the users on the foreign side. The application must compute the message's hash value, encrypt the hash value by applying the user's private signing key, then pass the message on to MSMQ.

Signature Verification Pseudo Code

The following code describes the basic elements needed to authenticate a message.

Retrieve cryptographic provider information needed to perform the cryptographic operation required for signature verification.

```
CryptProvName = GetMessageProperty(Message, PROPID_M_PROV_NAME)
CryptProvType = GetMessageProperty(Message, PROPID_M_PROV_TYPE)
```

Initialize the crpytographic provider.

```
CryptProvider = AcquireCrpytoraphicConext(
                                CryptProvName,
                                CryptProvType)
```

Get the hash algorithm identifier and initialize a hash object. This object is used to perform the hashing and signature-verification operations.

```
HashAlogorithm = GetMessageProperty(Message, PROPID_M_HASH_ALG)
HashObject = GetHashObject(CryptProvider, HashAlogorithm)
```

Get the six message properties that are required for calculating the hash value for the message.

```
CorrelationId = GetMessageProperty(Message, PROPID_M_CORRELATIONID)
AppSpecific = GetMessageProperty(Message, PROPID_M_APPSPECIFIC)
MessageBody = GetMessageProperty(Message, PROPID_M_BODY)
MessageLabel = GetMessageProperty(Message,   PROPID_M_LABEL)
RespQueueFormat = GetMessageProperty(Message, PROPID_M_RESP_QUEUE)
AdminQueueFormat = GetMessageProperty(Message,   PROPID_M_ADMIN_QUEUE)
```

Compute the hash value for the message by adding (in order) each message property to the hash value. The order in which the properties are added is important. Changing the calculation order of the message properties will cause signature verification to fail.

```
HashData(HashObject, CorrelationId)
HashData(HashObject, AppSpecific)
```

```
if NotEmpty(MessageBody)
        HashData(HashObject, MessageBody)
if NotEmpty(MessageLabel)
        HashData(HashObject, MessageLabel)
if NotEmpty(RespQueueFormat)
        HashData(HashObject, RespQueueFormat)
if NotEmpty(AdminQueueFormat)
        HashData(HashObject, AdminQueueFormat)
```

Get the message signature.

```
MessageSignature = GetMessageProperty(Message, PROPID_M_SIGNATURE)
```

Get the sender's certificate.

```
SenderCert = GetMessageProperty(Message, PROPID_M_SENDER_CERT)
```

Get the sender's public key out from the sender's certificate.

```
SenderPublicKey = GetPublicKeyFromCertificate(CryptProvider,
                  SenderCert)
```

Verify the signature of the message according to the message hash value and the sender's public key.

```
VerifySignature(HashObject, SenderPublicKey, MessageSignature)
```

The result of the verify signature function indicates whether or not the signature is valid.

Passing Private Messages

Passing private, encrypted messages between MSMQ and another message queue system is very similar to passing private messages between MSMQ applications. The only difference is a symmetric key that is exposed when sending a private message to the other message queue system.

The implementation of the connector application plays a significant role when passing private messages to their destination.

Transparent connector applications pass the symmetric key, encryption algorithm, and the encrypted message body on to the queue manager of the destination queue. This is the same regardless of the direction the message is going. When using a transparent connector application, the public keys of all destination queue managers must be registered in MQIS.

Non-transparent connector applications decrypt the message at the server and can pass on a clear-text message body or, encrypt the message again (using a the receiving systems encryption operations) and pass on a newly encrypted message body. When using a non-transparent connector application, all destination queues must use the public key of the connector application.

In addition to the type of connector application, there are two design considerations that should be taken into account:

- When multiple servers are used (two or more servers are connected to the same foreign Connector Networks), only transparent connector applications can be used.
- When sending messages from another message queue system to the MSMQ, the source application must get the public key of the destination queue before it can encrypt the message body. The source application must tell the connector application to call **MQGetMachineProperties** and pass back the public key found in PROPID_QM_ENCRYPTION_PK.

  This means that there must be a level of trust between the connector application and the foreign queue manager. There is no way for the foreign queue manager to independently verify that the key it receives is the public key it requested.

Caching the Symmetric Key

Non-transparent connector applications can cache the symmetric key to save time when decrypting private messages. If several messages arrive from the same source, there is no need to decrypt a symmetric key for each message.

To do this, the application must cache the identifier of the source computer and the decrypted and encrypted values of the symmetric key. The connector application can call PROPID_M_SRC_MACHINE_ID, to retrieve the identifier of an MSMQ source machine.

When the next message arrives, the connector application can first check if a message has already arrived from that source machine. If the machine identifier is not found, the connector application will add it to the cached values.

If a previous message has arrived, the application can next test to see if the new encrypted key matches the cached encrypted key. If they match, the cached decrypted value can be used immediately. If they do not match, it will decrypt the new key (storing both the encrypted and decrypted values in the cache), using the new value to decrypt the message body.

Using the MSMQ Connector in a Transaction

The MSMQ connector server is not available with the Windows NT® 4.0 Option Pack.

When an MSMQ application sends messages in a transaction, MSMQ routes the message to the transactional connector queue.

When the message is sent, MSMQ keeps a private copy of the message in the source queue manager. This private copy is either discarded when the source queue manager receives a read receipt acknowledgment message, or moved to the computer's DEADXACT queue if a negative acknowledgment is received or a time-out occurs.

To know where to return the read receipt or negative acknowledgment, the connector application must retrieve the format name of the queue by looking at the message's PROPID_M_XACT_STATUS_QUEUE property. The read receipt must be returned even if the sending application did not request acknowledgments (PROPID_M_ACKNOWLEDGE is set to MQMSG_ACKNOWLEDGMENT_NONE).

MSMQ Mail Services

MSMQ mail services provide a way to send MSMQ mail messages to applications that use e-mail based messaging. These applications include Microsoft® Exchange, as well as individual MAPI client applications. Application developers can use these mail services to combine the ease of development and use of e-mail-based forms with the computing power and interoperability of MSMQ.

MSMQ provides two mail services: the MSMQ MAPI Transport Provider and the MSMQ Exchange Connector. The differences between the two services are listed below.

MSMQ Exchange Connector	MSMQ MAPI Transport Provider
Runs on a single computer, serving all Microsoft Exchange users.	Runs on a each MAPI client application computer, serving only one MAPI client.
Requires MSMQ installed on one computer.	Requires MSMQ installed on each MAPI client computer.
Uses a single queue to send e-mail to all users.	Uses a different user input queue for each MAPI user.
Stores the addresses of the MSMQ applications in the Microsoft Exchange address book, which is common to all Exchange users.	Stores addresses in personal address books, which cannot be shared among MAPI clients.
Requires a connection to a Microsoft Exchange server.	Does not require Microsoft Exchange.
When sending mail messages, the MSMQ application must specify the address of the Microsoft Exchange user.	When sending mail messages, only the label of the MAPI client's queue is needed.

The MSMQ mail services currently support sending/receiving of the following:

- Text Messages, without attachments of any kind.
- EFD (Exchange Form Designer) forms, without attachments of any kind.
- MAPI TNEF messages.
- Delivery and non-delivery reports.

The MSMQ mail services do not support sending/receiving email with:

- Any type of attachment (word docs/pictures/etc...)
- Outlook custom forms

MSMQ Mail SDK

The MSMQ Mail SDK provides the API functions and ActiveX components needed to compose and parse mail messages.

The MSMQ Mail API functions include:

- **MQMailComposeBody**
- **MQMailFreeMemory**
- **MQMailParseBody**

The MSMQ Mail ActiveX objects include:

- **MSMQMailEMail**
- **MSMQMailFormData**
- **MSMQMailFormField**
- **MSMQMailFormFieldList**
- **MSMQMailRecipient**
- **MSMQMailRecipientList**
- **MSMQMailTextMessageData**
- **MSMQMailTnefData**
- **MSMQMailDeliveryReportData**
- **MSMQMailNonDeliveryReportData**

MSMQ Exchange Connector

The MSMQ Exchange Connector provides a way for Microsoft® Exchange users to communicate with MSMQ applications. Using the Exchange Connector, e-mail can be sent to MSMQ applications and MSMQ mail messages can be sent to Exchange users.

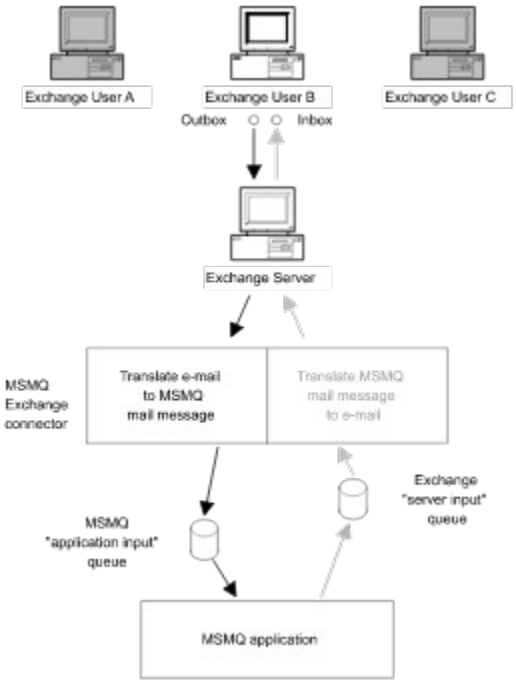
The Exchange Connector is the recommended solution for connecting Microsoft Exchange users to MSMQ applications.

Additional Exchange Connector information can be found in the following:

- To install and use the Exchange Connector, see the *Microsoft Message Queue Administrator's Guide*.
- To see how mail is sent between Exchange users and MSMQ applications, see Sending E-mail to an MSMQ Application or Sending MSMQ Mail Messages to an Exchange User.
- To install the MSMQ Mail SDK, see MSMQ Mail SDK.
- To use the API functions and ActiveX components provided by the MSMQ Mail SDK, see the reference page for each item. They can be located by their name, or under MSMQ Mail Functions and MSMQ Mail ActiveX Components.
- For an example of an MSMQ Mail sample application, see MSMQ Exchange Connector: Book Server Application.

Sending E-mail to an MSMQ Application

E-mail sent by Exchange users is routed through their Exchange server to the MSMQ Exchange Connector, then sent on to the input queue of the MSMQ application (see illustration below). The MSMQ application can then read, process, and return a mail message to the Exchange user who sent the mail, or forward the mail message (or another mail message) on to the next processing destination.



To send e-mail to an MSMQ application, the Exchange Connector performs the following functions:

- Converts the e-mail to a mail message whose body is in MSMQ mail format.
- Sends the converted mail message to the application input queue of the MSMQ application.

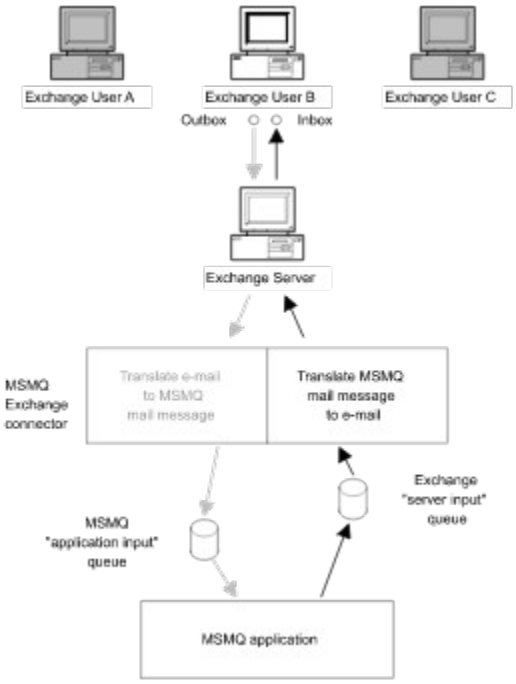
The MSMQ application creates the application input queue that will receive the mail message. The application must specify a queue type when it creates a queue for the Exchange Connector to find the queue.

Additional information can be found in the following:

- To see how MSMQ messages are sent to Exchange users, see Sending MSMQ Mail Messages to an Exchange User.
- To set up the Exchange Connector, see the *MSMQ Administrator's Guide*.
- To install the MSMQ Mail SDK, see MSMQ Mail SDK.
- To use the API functions and ActiveX components provided by the MSMQ Mail SDK, see the individual reference pages for each item. They can be located by their name, or under MSMQ Mail Functions and MSMQ Mail ActiveX Components.

Sending MSMQ Mail Messages to an Exchange User

MSMQ mail messages sent to Exchange users are sent to the server input queue of the Exchange Connector. MSMQ mail messages are then routed through the Exchange Connector where they are translated, then sent on to the Exchange Server where they are distributed to the appropriate Exchange user (see illustration below).



When an MSMQ message is sent to an Exchange user, the Exchange Connector performs the following functions:

- Creates a server input queue. This queue is created only once by the Exchange Connector. The Exchange Server input queue is not used when sending information in the other direction (when an Exchange user sends e-mail to an MSMQ application).
- Translates the mail message into e-mail.
- Sends the e-mail on to the Exchange Server.

The MSMQ application must know the address of the Exchange user it is sending messages to.

Additional information can be found in the following:

- To see how mail is sent to MSMQ applications, see Sending E-mail to an MSMQ Application.
- To set up the Exchange Connector, see the Administrator's Guide.
- To install the MSMQ Mail SDK, see MSMQ Mail SDK.
- To use the API functions and ActiveX components provided by the MSMQ Mail SDK, see the reference page for the item. They can be located by their name, or by using MSMQ Mail Functions and MSMQ Mail ActiveX Components.

MSMQ Exchange Connector: Book Server Application

The Book Server sample application uses the MSMQ Exchange Connector to pass messages between a Microsoft® Exchange user and an MSMQ application. In this application the Exchange user requests a query, the MSMQ application performs the query, then the results of the query are sent back to the Exchange user.

How Book Server Works

Here is the step-by-step process that Book Server uses to perform its query.

- Using Microsoft Exchange, the Microsoft Exchange user enters the search criteria and submits the query mail, which is pre-addressed to a recipient called Book Server. The mail's MSMQ-typed address is set to the label of the application input queue of the sample MSMQ application *booksrvr*.
- The Exchange connector receives the form from Exchange (because the recipient is an MSMQ-typed recipient), translates it into an MSMQ Mail message, and sends it to the application input queue of the sample MSMQ application.
- The sample MSMQ application (*booksrvr*) retrieves the message from the application input queue, calls **MQMailParseBody** to parse the body of the message into a mail structure, then finds the values of the query fields from the structure and performs the query.
- Once the query is complete, a reply form is sent to the Exchange connector. The reply form is created by including the results of the query, who is sending the results (in this case the *booksrvr* application), and who will receive the results (the sender of the original mail) in a new mail structure. After the structure is filled in, a call to **MQMailComposeBody** creates a message body that is then sent to the Exchange connector's server input queue.
- The Exchange connector retrieves the message from the queue, translates it into mail, then submits it to the Exchange user who requested the query.
- The Exchange user can then retrieve the mail from the inbox and view the results of the query.

To Run Book Server

1. Build the sample MSMQ application.
   - Make sure that the environment (PATH, INCLUDE, and LIB variables) is set correctly to compile windows applications.
   - Edit the file mk.bat that is in the *booksrvr* directory. Make the following changes to point to the correct locations of the MSMQ SDK directory, and the MSMQ Mail SDK directory.

     ```
     set MSMQ_SDK=c:\msmq\sdk
     set MQMAIL_SDK=c:\msmqmail
     ```

   - Run mk.bat. This should build booksrvr.exe in the *booksrvr* directory.
2. Ask your Exchange administrator to install the MSMQ Exchange Connector from the MSMQ CD.
3. From the booksrvr directory, the administrator must install the forms bookfrm.efp and bookres.efp into your Exchange system.
4. Your Exchange administrator must then add a custom recipient named "Book Server" with the address-type MSMQ and whose address is "booksrvrq." It must NOT be a rich-text recipient.
5. Run booksrvr.exe (make sure MSMQ is running).Type "booksrvr booksrvrq" in the command line. The argument booksrvrq is the label of the queue that the booksrvr application will use as its input queue. This label should be identical to the address of the "Book Server" custom recipient that your Exchange administrator entered.
6. Send the query. In your mail client, open the form named "Book Search". Fill in the appropriate fields, and submit the form.
7. Verify the query results. After a short time, you should receive a result form sent by the sample *booksrvr* application with the results of the query.

MSMQ MAPI Transport Provider

The Microsoft® Message Queue Server (MSMQ) MAPI Transport Provider contains a single MAPI transport provider that allows you to connect MAPI client applications (such as Microsoft® Exchange clients) to MSMQ applications. As a transport provider, the MSMQ MAPI Transport Provider translates between e-mail and MSMQ mail messages.

Additional MAPI Transport Provider is outlined in the following:

- To set up the MAPI Transport Provider, see Setting up the MSMQ MAPI Transport Provider.
- To see how e-mail forms are sent to MSMQ applications, see Sending E-mail to an MSMQ Application.
- To see how mail messages are sent to MAPI applications, see Sending MSMQ Mail Message to a MAPI Client
- To install the MSMQ Mail SDK, see MSMQ Mail SDK.
- To use the API functions and ActiveX components provided by the MSMQ Mail SDK, see the reference page for each item. They can be located by their name, or by using MSMQ Mail Functions and MSMQ Mail ActiveX Components.

Setting up the MSMQ MAPI Transport Provider

Setting up the MSMQ MAPI Transport Provider is a two-part process that includes:

- Installing the MAPI Transport Provider.
- Creating a MAPI address for your MSMQ application.

Installing the MSMQ MAPI Transport Provider

The MSMQ MAPI Transport Provider can be installed using its own Setup program or from any other MAPI profile configuration tool (such as Microsoft Exchange).

The MAPI Transport Provider's Setup program can be found in:

msmq\MQMail\mapixp\setup

Note To run Setup from another MAPI profile configuration tool, refer to the documentation for that tool. Either process adds the Microsoft® Message Queue service to the list of MAPI Transport Providers available to your MAPI applications.

After MAPI Transport Provider is installed, add Microsoft Message Queue to your other MAPI profiles (for more information on how to add services to your profiles, refer to the documentation for your MAPI client). After adding Microsoft Message Queue, restart your other MAPI applications to make sure your profile changes have taken effect.

Creating MAPI Addresses

Each MSMQ application that sends mail messages or receives translated e-mail messages must have a MAPI address. The MSMQ MAPI Transport Provider uses this information to locate the application's input queue.

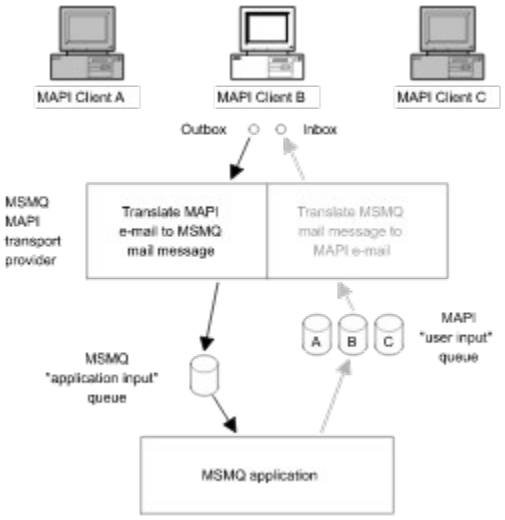
The addresses for your MSMQ applications should be created and saved in the personal address book used by your MAPI applications. To create an address, open the personal address book and add a new address of the type **Other Address**. The following table describes the necessary content for each field.

Field	Description
Display name	Specify a name for the MSMQ application. This name is an alias and is only used for display purposes.
E-mail address	Specify the label of the MSMQ application's input queue (case sensitive). The MAPI transport provider uses this information to locate the input queue.
E-mail type	Set to **MSMQ**. This instructs the MAPI Transport Provider to handle this type of forms.

Clear the **Always send to this recipient in Microsoft Exchange rich text** check box. Rich text information should never be sent to an MSMQ application.

Sending E-mail to an MSMQ Application

The e-mail submitted by MAPI client applications are routed through the MSMQ MAPI Transport Provider to the MSMQ application (see the following illustration). The MSMQ application can then pick up, process, and return the mail to the MAPI user who sent the mail, or forward it (or another mail) on to the next processing destination.



To send forms to an MSMQ application, the MSMQ MAPI Transport Provider performs the following tasks:

- Converts the e-mail form to an MSMQ mail message whose body is in MSMQ mail format.
- Sends the converted mail message to the MSMQ application input queue used by the MSMQ application.

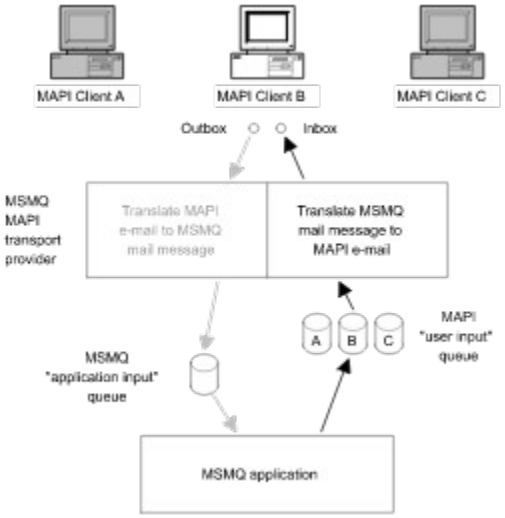
The MSMQ application must create the application input queue. When creating the queue, the application must specify a queue type so that the MAPI Transport Provider can find the queue.

Additional information can be found in the following:

- To see how MSMQ messages are sent to MAPI clients, see Sending MSMQ Mail Message to a MAPI Client.
- To set up the MAPI Transport Provider, see Setting up the MSMQ MAPI Transport Provider.
- To install the MSMQ Mail SDK, see MSMQ Mail SDK.
- To use the API functions and ActiveX components provided by the MSMQ Mail SDK, see the reference page for each item. They can be located by name, or by using MSMQ Mail Functions and MSMQ Mail ActiveX Components.

Sending MSMQ Mail Message to a MAPI Client

MSMQ mail messages sent to a MAPI client are sent to the client's MAPI user input queue and routed through the MAPI Transport Provider where they are translated. Then they are sent to the inbox of the appropriate MAPI client. See the following illustration.



When an MSMQ message is sent to a MAPI client, the MAPI Transport Provider performs the following functions:

- Creates a MAPI user input queue for the MAPI user. This queue is created just once by the MSMQ MAPI Transport Provider. However, the MAPI user input queue is not used when sending information in the other direction (when MAPI clients send e-mail to an MSMQ application).
- Translates the MSMQ mail message into an e-mail form.
- Sends the e-mail form on to the inbox for the MAPI application.

The MSMQ application must know the MAPI address of the client where it sends the messages.

Additional information about MAPI tasks can be found in the following:

- To create an address for the MAPI client, see Creating MAPI Addresses.
- To see how e-mail forms are sent to MSMQ applications, see Sending E-mail to an MSMQ Application.
- To set up the MAPI Transport Provider, see Setting up the MSMQ MAPI Transport Provider.
- To install the MSMQ Mail SDK, see MSMQ Mail SDK.
- To use the API functions and ActiveX components provided by the MSMQ Mail SDK, see the reference page for each item. They can be located by name, or by using MSMQ Mail Functions and MSMQ Mail ActiveX Components.

Logging Information

The MSMQ MAPI Transport Provider uses the Application Event Log to record informational messages and error messages. When logging messages, it inserts the name MSMQMAPI in the source column of the event viewer.

For example, when Microsoft Exchange starts the MAPI Transport Provider, it performs a basic initialization, logging the following informational messages in the event viewer.

If Initialization...	**Then...**
Succeeds	"Transport provider logon succeeded"
Fails	"Transport provider logon failed"

Error entries that are logged by the MAPI Transport Provider include a details section that provides a stack trace of error descriptions. Save this information for a reference when troubleshooting problems.

Designing Mail

For mail processing to work correctly, the mail designer and the MSMQ application developer must agree on the field names used in the mail. The MSMQ application must know the name of the fields on the form so that it can compose new mail messages and parse received mail messages.

For example, when creating a form that has several fields, the mail designer can use the Microsoft® Exchange Form Designer to specify a name for each field (the name of each field is entered in the **Reference Name** field on the **General** page in the **Field Properties** dialog box). The MSMQ application developer needs to know these names to write an application that can parse the mail messages it receives and compose its own mail messages.

Note For more information on designing forms, refer to your Exchange Form Designer documentation.

MSMQ Mail Format

The MSMQ mail format is used to represent e-mail information within the body of an MSMQ mail message. It is used by the MSMQ Exchange Connector, MSMQ MASPI Transport provider, and the MSMQ Mail SDK.

Note The MSMQ mail format is a subset of the MIME (Multipurpose Internet Mail Extensions) format.

The body of a mail message is composed of:

- A header section (see Header Section)
- An empty line (CR/LF pair)
- A body section (see Body Section)

For samples of the MSMQ mail format, see Sample Form Message and Sample Text Message.

Header Section

The header section is composed of several headers, with each header on a separate line (each line is terminated by a CR/LF pair). Each header is composed of a header name, followed by a colon, and a header value (for example, *Subject*: *message subject*).

The following table describes the headers that are used.

Header Name	Description
Mime-Version	Describes the MIME version used. It should contain the value 1.0. Example: *Mime-Version: 1.0*
Date	Indicates the time the message was submitted by the sender. It is formatted as {*sender-local-date*} {*sender-local-time*} {*sender-local-time-zone*}. Example: *Date: 21 Jul 1996 20:12:06-0300* Month names: *Jan*, *Feb*, *Mar*, *Apr*, *May*, *Jun*, *Jul*, *Aug*, *Sep*, *Oct*, *Nov*, *Sep*, and *Dec* Time zones should be numeric and prefixed with a "-" (minus sign) or a "+" (plus sign); that is, no named time zones.
From	Indicates the sender of the message. For applications using the Exchange Connector, this header is formatted as {sender-friendly-name<user-email-alias@server-input-queue-label>}. Where the user-email-alias is the user's alias in the Microsoft Exchange system (corresponds to the PR_ACCOUNT property of the user) and server-input-queue-label is the label of the Exchange Connector's input queue. Example: *From: Jane Doe <janedoe@Exchangeserverinputqueue>* For applications using the MAPI Transport Provider, it is formatted as {*sender-friendly-name* <*sender-queue-label*>}. Where the *sender-queue-label* is the MSMQ queue label associated with the sender. Example: *From: John Doe <johndoe>*
Subject	Indicates the subject of the message. The subject text is not formatted. Example: *Subject: message subject*
To	Lists each message recipient, separated by a comma. Each recipient is formatted as *{recipient-friendly-name <recipient-address>}*. For the MSMQ MAPI Transport Provider, the recipient address is the label of the MAPI user input queue associated with the recipient. Example: *To: John Doe<johndoe>* For the MSMQ Exchange Connector, the recipient name includes the user name plus the

	label of the Exchange Connector's server input queue. Example: *To: Jane Doe<janedoe@ServerInputQueueLabel>*
Cc	Optional. Lists the copied message recipients, separated by a comma. Each recipient is formatted as *{recipient-friendly-name <recipient-address>}*. For the MSMQ MAPI Transport Provider, the recipient address is the label of the MAPI user input queue associated with the recipient. Example: *To: John Doe<johndoe>* For the MSMQ Exchange Connector, the recipient name includes the user name plus the label of the Exchange Connector's server input queue. Example: *To: Jane Doe<janedoe@ServerInputQueueLabel>*
Bcc	Optional. Lists the hidden copied message recipients, separated by a comma. Each recipient is formatted as *{recipient-friendly-name <recipient-address>}*. For the MSMQ MAPI Transport Provider, the recipient address can be the MSMQ queue label associated with the recipient. Example: *Bcc: John Doe<johndoe>* For MSMQ Exchange Connector, the recipient name includes the user name plus the MSMQ queue label of the connector's queue. Example: *Bcc: Jane Doe<janedoe@ExchangeConnectorQueueLabel>*
Content-Type	Optional. Describes the content of the body section of the message. It can have one of the following values (default is "text/plain; charset=us-ascii"): *text/plain; charset=us-ascii* Means that this is a regular message, and the body section contains the message text. Example: *Content-Type: text/plain; charset=us-ascii* *application/x-ms-tnef* Means that this is a TNEF message and that the body section contains the TNEF data of the message. Example: *Content-Type: application/x-ms-tnef* *multipart/form-data; boundary=boundary-string* Means that the form and body section have

	multiple parts, and each part describes a single form field (name, value, and so on). The boundary-string separates multiple parts in the body section (see Body Section). Example: *Content-Type: multipart/form-data; boundary=xyz12sssdeeggg*
Content-Transfer-Encoding	Describes the encoding of the body section (if it contains binary data). The only legal header value is binary (that is, no encoding of binary data). Example: *Content-Transfer-Encoding: binary*
X-Form-Name	Name of the form whose fields are described in the body section. This header is not necessary on text messages, only on form messages. The name here corresponds to the name defined by the Exchange Form Designer (see the **Item Type** field in the **General** tab of the **Form Properties** dialog box) and the PR_MESSAGE_CLASS property of the form. Example: *X-Form: IPM.ORGANIZATION.SAMPLE*
X-Delivery-Report-Requested	Specifies whether the receiving application should return a delivery report when the message is received. Valid values are True or False. Example: *X-Delivery-Report-Requested: True*
X-Non-Delivery-Report-Requested	Specifies whether a non-delivery report should be sent back. Valid values are True or False. Example: *X-Non-Delivery-Report-Requested: True*
X-Report-Date	Indicates the time the report was submitted. This header is only set for delivery or non-delivery report messages. It is formatted as the *Date* header mentioned previously. Example: *X-Report-Date: 22 Jul 1996 21:12:06 -0300*
X-Report-To	Specifies the recipient of the delivery or non-delivery report. This header is set only on delivery or non-delivery report messages. It is formatted as the *To* header above, but contains only one recipient. Example: *X-Report-To: John Doe<johndoe>*
X-Report-Delivered-To X-Report-Delivered-Cc	Specifies recipients who received a previously sent mail. These headers are only set on delivery report

X-Report-Delivered-Bcc	messages. X-Report-Delivered-To specifies the delivered recipients from the To: list of the original e-mail. X-Report-Delivered-Cc specifies the delivered recipients from the Cc: list of the original e-mail. X-Report-Delivered-Bcc specifies the delivered recipients from the Bcc: list of the original e-mail. It is formatted as the *To* header above, but contains the delivery time as well. The delivery time is formatted as the *Date* header above, and appears between round brackets after the recipient's address. Example: *X-Report-Delivered-To: John Doe<johndoe>(22 Jul 1996 21:11:06 -0300), Jane Doe*<mailto:janedoe@ExchangeConnectorQueueLabel *>(21 Jul 1996 20:06:06 -0300)*
X-Report-Not-Delivered-To	Specifies recipients who did not receive a previously sent mail.
X-Report- Not-Delivered-Cc	These headers are set only on non-delivery report messages.
X-Report- Not-Delivered-Bcc	X-Report-Not-Delivered-To specifies the non-delivered recipients from the To: list of the original e-mail. X-Report-Not-Delivered-Cc specifies the non-delivered recipients from the Cc: list of the original e-mail. X-Report-Not-Delivered-Bcc specifies the non-delivered recipients from the Bcc: list of the original e-mail. It is formatted as the X-Report-Delivered-To header above, but contains the non-delivery reason instead of the delivery time. The non-delivery reason is not formatted, and appears between round brackets after the recipient's address. Example: *X-Report-Not-Delivered-To: John Doe<johndoe>(Communication failure), Jane Doe*<mailto:janedoe@ExchangeConnectorQueueLabel *>(Recipient is not known at this address)*

For samples of the MSMQ mail format format, see Sample Form Message and Sample Text Message.

Body Section

The body section of the MSMQ mail format begins at the line after the separating empty line and ends at end of the file.

The body section can be formatted in several ways. If the Content-Type header value is *text/plain*, the body section only contains the text of the message. If the Content-Type header value is *multipart/form-data*, the body section contains a collection of field sections, each separated by a boundary string (the boundary string is defined in the header section).If the Content-Type header value is *application/x-ms-tnef*, the body section contains the TNEF data of the message. For information on the Content-Type header, see Header Section.

The format of a multiple, form-data body section is as follows:

```
--boundary string
field section 1
--boundary string
field section 2
--boundary string--
```

Each field section describes one of the form's fields and is formatted very much like the message itself. Each section starts with a Content-Disposition header line, followed by an empty line, followed by a body section line. The Content-Disposition line describes the field and the body section line specifies the value of the field. The boundary string following the last field section always ends with two minus characters (--).

The following example shows a field section that specifies the customer name *John Doe* (*7fs9dfsdfs9sdf* is the boundary string):

```
--7fs9dfsdfs9sdf
Content-Disposition: form-data; name=Customer

John Doe
--7fs9dfsdfs9sdf                    'If last field, add --
```

If the field is a Boolean field such as a text box, then the Content-Disposition line should also contain the parameter *x-type=boolean* (prefixed by a semicolon).

For example:

```
blank line(cr/lf)
--hj57ujkdfg4535
Content-Disposition: form-data; name=SaveSettingsOnExit; x-type=boolean

true
--hj57ujkdfg4535                    'If last field, add --
```

Note When creating a multiple, form-data body section, the following apply:

- The actual separator starts with two minus characters (--), followed by the supplied boundary string. The last boundary also ends with two minus characters (--). This is part of the MIME multipart format.
- The blank line (CR/LF pair) before the boundary string belongs to the boundary, not to the field section before it. If you'd like the field section to end with a CR/LF, leave an empty line before the boundary.
- Any text in the body section that comes before the first boundary string, or after the last boundary string (the boundary string followed by two minus characters) is ignored.

For samples of the MSMQ MAIL format, see Sample Form Message and Sample Text Message.

Sample Text Message

In this sample, there are eight headers in the header section and two lines in the message body. Notice that the Content-Type header (see Body Section) indicates that the body of the message contains only text.

```
Mime-Version: 1.0
Date: 21 Jul 1996 20:12:06 -0300
From: User name <useraddress>
Subject: Message representation
Content-Transfer-Encoding: binary
To: John Doe <johndoe>, Good Guy <ggqueue>
Cc: Another Good Guy <aggqueue>
Content-Type: text/plain; charset=us-ascii

This is the first line of the message body. Notice the previous
empty line that separates this body section from the eight headers
in the header section.

This is the second line of the message body.
```

For information on the header and body sections, see Header Section and Body Section.

For a sample of a form message, see Sample Form Message.

Sample Form Message

In this sample, there are eight headers in the header section and five section fields in the message body. Notice that the Content-Type header (see Body Section) indicates that the body of the message contains fields, and that the boundary string is *7pKviP84rI4ZzGBds*.

Mime-Version: 1.0
Date: 21 Jul 1996 20:12:06 -0300
From: Jane Doe (MSMQ) <janedoe>
Subject: Form representation
X-Form-Name: IPM.MSMQMAPI.SAMPLE
Content-Transfer-Encoding: binary
To: John Doe <johndoe>
Content-Type: multipart/form-data; boundary=7pKviP84rI4ZzGBds

--7pKviP84rI4ZzGBds
Content-Disposition: form-data; name=TextBox_Name

This is a text box control
--7pKviP84rI4ZzGBds
Content-Disposition: form-data; name=CheckBox_Name; x-type=boolean

true
--7pKviP84rI4ZzGBds
Content-Disposition: form-data; name=RadioButton_Name

RadioButton_choice1
--7pKviP84rI4ZzGBds
Content-Disposition: form-data; name=ComboBox_Name

This is a combo box control
--7pKviP84rI4ZzGBds
Content-Disposition: form-data; name=ListBox_Name

2222
--7pKviP84rI4ZzGBds--

For information on the header and body sections, see Header Section and Body Section.

For a sample of a text message, see Sample Text Message.

MSMQ ActiveX Support

MSMQ provides a set of ActiveX objects for developing MSMQ applications using any ActiveX development tool, including Microsoft® Visual Basic® (version 4.0 or later), or C.

These objects provide the most common MSMQ API functionality needed for developing MSMQ applications. This includes queue lookup, queue management, message management, and queue administration. The guiding principle in designing these ActiveX objects is object model simplicity.

The MSMQ Object Model

The MSMQ object model supports queue lookup, queue management, message management, and queue administration, allowing for a wide variety of MSMQ application development.

This object model includes:

- **MSMQQuery** This object locates a collection of queues. It provides a lookup method based on the queue's properties. The lookup method used by **MSMQQuery** returns an **MSMQQueueInfos** object that references the selected queues.

  For an example, see:

  Locating a Public Queue.
- **MSMQQueueInfos** This object represents a collection of **MSMQQueueInfo** objects, each corresponding to one of the queues found by the **MSMQQuery** object. It provides the methods for selecting one of the queues found in the query.

  For an example, see:

  Locating a Public Queue.
- **MSMQQueueInfo** This object contains the information needed to create or access a single queue. It provides methods for creating, opening, and deleting queues. Each time a queue is opened, an **MSMQQueue** object is returned.

  For examples see:

  Creating a Queue

  Opening a Queue
- **MSMQQueue** This object references a single instance of a queue. It provides methods for working with the messages in the queue, as well as enabling receive notification and closing the queue instance.
- **MSMQMessage** This object corresponds to a message in the queue. Each **MSMQQueue** object (described earlier) has an implicit collection of **MSMQMessage** objects.
- **MSMQEvent** This object provides the events needed to implement a single event handler that can support multiple queues. Only two events are provided: message arrival and error notification.
- **MSMQCoordinatedTransactionDispenser** This object provides a single method for creating a transaction object for external transactions.

  For an example see:

  Sending Messages Using an MS DTC External Transaction
- **MSMQTransactionDispenser** This object provides a single method for creating a transaction object for internal transactions.

  For an example see:

  Sending Messages Using an Internal Transaction
- **MSMQTransaction** This object provides methods for committing and aborting transactions.

  For examples see:

  Sending Messages Using an MS DTC External Transaction

  Sending Messages Using an Internal Transaction
- **MSMQApplication** This object provides a single method for obtaining the machine identifier of a computer.

Invoking ActiveX Objects

ActiveX objects can be invoked using Microsoft® Visual Basic®, C/C++, and VC5 with #import.

The following example shows how to create a queue using Visual Basic, VC5 with #import, and C.

Using Visual Basic

```
dim qinfo as New MSMQQueueInfo

on error goto ErrHandler
qinfo.PathName = ".\queuename"
qinfo.Create
Exit Function
ErrHandler:
    ' handle Create error
```

Using VC5 and #import

Using VC5 with #import provides an easy-to-use syntax that is similar to the syntax provided by Visual Basic. This syntax provides:

- HRESULT to exception mapping. A VC5/ActiveX application can use **try** and **catch** to handle errors instead of testing for return values.
- Support for optional parameters (not available in C/C++ implementations).
- Reference counting and Query Interface supportso there is no need for explicit AddRef/QueryInterface.

```
#import "mqoa.dll;

    try {
        IMSMQQueueInfoPtr pqinfo ("MSMQ.MSMQQueueInfo");
        pqinfo->PutPathName (L".\queuename");
        //
        // Create non-transactional, non-world-readable queue.
        //
        pqinfo->Create();
        catch (_com_error &e) {
              // UNDONE: handle error.
              }
        }
```

Using C/C++

```
    IMSMQQueueInfo *pqinfo;

    HRESULT hresult;
    VARIANT varIsTransactional;
    VARIANT varIsWorldReadable;
    //
    // Create MSMQQueueInfo object
    //
    hresult = CoCreateInstance(
                      CLSID_MSMQQueueInfo,
                      NULL,                         // punkOuter
                      CLSCTX_SERVER,
```

```
                    IID_IMSMQQueueInfo,
                    (LPVOID *)&pqinfo
                    );
if (SUCCEEDED(hresult)) {
  // Set the PathName.
  pqinfo->put_PathName(L".\queuename");
  //
  // specify if transactional
  //
  VariantInit(&varIsTransactional);
  varIsTransactional.vt = VT_BOOL;
  varIsTransactional.boolVal = MQ_TRANSACTIONAL_NONE;
  VariantInit(&varIsWorldReadable);
  varIsWorldReadable.vt = VT_BOOL;
  varIsWorldReadable.boolVal = FALSE;
  //
  // create the queue
  //
  hresult = pqinfo->Create(&varIsTransactional,
                           &varIsWorldReadable);
  //
  // UNDONE: need to handle failure...
  //
}
```

Setup and Installation

This topic describes the system requirements and installation procedures for developing your MSMQ application.

System Requirements

- Any development tool that supports ActiveX objects, including Microsoft® Visual Basic® version 4.0 (32 bit), Microsoft® Visual Basic® version 5.0, Microsoft® Office (95 and 97), Delphi, Microsoft® Internet Explorer, Microsoft® Active Server Pages, as well as others.

To Use MSMQ ActiveX Components

1. Run MSMQ Queue Manager.
2. Run your development tool (Visual Basic or any other development tool that supports ActiveX objects).
3. Select the MSMQ Object Library.
   - For Visual Basic 4.0: On the **Tools** menu, click **References**, and then choose **MSMQ Object Library**.
   - For Visual Basic 5.0: On the **Project** menu, click **References**, and then choose **MSMQ Object Library**.
   - For all other tools, refer to the development tools documentation for selecting type libraries
4. Select any other object library needed by your application.

Using Microsoft Visual Basic

The following programming tips are provided for those new to writing MSMQ or Microsoft® Visual Basic® applications. They highlight several issues that may make writing your MSMQ application a bit easier.

- When declaring object variables, the **New** keyword can be added to the **Dim** statement to enable implicit creation of the object. What this means is that each time the variable is referenced, a new instance of the object is implicitly created if the current value of the variable is NULL.
- In contrast, when the **New** keyword is not used an instance of the object is only created when the **Set** command is called or some other mechanism (such as a call to a method that returns an object reference) is used to obtain an object instance. The **Set** command can be used as well for variables that were declared with the **New** keyword.
- The example below shows when the **New** keyword should be used and when it should not be.

```
Dim qDest As MSMQQueue                  'Set command needed.
Dim msgSent As New MSMQMessage
Dim msgDest As MSMQMessage              'Peek method returns
                                        'MSMQMessage instance.

Set qDest = qinfoDest.Open(MQ_SEND_ACCESS, MQ_DENY_NONE)
msgSent.Send qDest
Set qDest = qinfoDest.Open(MQ_PEEK_ACCESS, MQ_DENY_NONE)
Set msgDest = qDest.Peek(ReceiveTimeout:=100)
```

When functions and subroutines are called, parentheses are sometimes required and sometimes not.

When functions or subroutines are called explicitly using the **Call** keyword, parentheses are required whenever there is one or more arguments. (The return value of a function can be ignored.)

For example:

```
Call Foo(x)
Call Foo(1, 2)
```

When a function is called and the return value is used, parentheses are always required whenever there is one or more arguments.

For example:

```
y = Foo(x, z)           ' Result of Foo used to assign to y.
Call Bar(Foo(1))        ' Result of Foo used as argument to Bar.
```

However, when a function or subroutine is called without using the **Call** keyword and its return value is ignored:

- Parentheses cannot be used for functions or subroutines that take more than one argument.
- If parentheses are used for functions or subroutines that take a single argument, then that argument is effectively passed 'by value' since the argument is in effect an expression whose result is returned in a temporary variable.

For example:

```
Foo x       'Parentheses cannot be used: x is passed by reference.
Foo (x)     'x is effectively passed by value.
```

- Use named arguments to make your code easier to read. Using non-named arguments forces the reader to remember the argument's name and the order of the arguments. For example, the following to lines of code are functionally identical, yet the first is much easier to understand:

```
Create IsWorldReadable:=True, IsTransactional:=False
Create False, True
```

- A Variant containing an array can be used like an array, e.g. ubound(msg.Id) or msg.CorrelationId(10).
- When declaring object's, specify the object class in the **Dim** statement (early-binding). Using early-binding whenever possible will make your application run faster. For example, the following examples are both functionally equal, yet the first example executes faster due to early-binding of the object.

```
dim qinfo as MSMQQueueInfo
set qinfo = New MSMQQueueInfo
qinfo.PathName = ".\PRIVATE$\CreateTest"
qinfo.Create


dim qinfo as Object
set qinfo = New MSMQQueueInfo
qinfo.PathName = ".\PRIVATE$\CreateTest"
qinfo.Create
```

- The Microsoft® Visual Basic® 5.0 debugger can be used on MSMQ application executable and DLL files generated by Visual Basic 5.0. You can set breakpoints as well as disassemble and see the generated VBA code and look at local variables.
- The **MSMQApplication** object does not have to be referenced. For example, the following three calls to **MachineIdOfMachineName** all return the same computer identifier.

```
Dim strId As String
Dim myapp As New MSMQApplication
strId = MachineIdOfMachineName("machinename")
Debug.Print strId
strId = MSMQApplication.MachineIdOfMachineName("machinename")
Debug.Print strId
strId = myapp.MachineIdOfMachineName("machinename")
Debug.Print strId
```

About MSMQ Reference

This reference describes the functions, properties, structures, error and warning codes, and ActiveX components provided by the Microsoft® Message Queue Server (MSMQ) SDK and the Microsoft® Message Queue Server Mail SDK.

Note For code examples of basic MSMQ functions such as creating a queue, sending message, receiving messages, and so on, refer to "Using MSMQ." For background information on MSMQ concepts, refer to the "MSMQ Guide."

The following topics are in the "MSMQ Reference:"

MSMQ Functions

The MSMQ API functions provide the means to manage queues and messages within your MSMQ application. The MSMQ API includes functions for creating, opening, and deleting queues; for locating existing queues and messages in queues; for sending messages and reading them in queues; as well as functions for setting and retrieving properties.

MSMQ API functions include:

- **MQBeginTransaction**
- **MQCloseCursor**
- **MQCloseQueue**
- **MQCreateCursor**
- **MQCreateQueue**
- **MQDeleteQueue**
- **MQFreeMemory**
- **MQFreeSecurityContext**
- **MQGetMachineProperties**
- **MQGetQueueProperties**
- **MQGetQueueSecurity**
- **MQGetSecurityContext**
- **MQHandleToFormatName**
- **MQInstanceToFormatName**
- **MQLocateBegin**
- **MQLocateEnd**
- **MQLocateNext**
- **MQOpenQueue**
- **MQPathNameToFormatName**
- **MQReceiveMessage**
- **MQSendMessage**
- **MQSetQueueProperties**
- **MQSetQueueSecurity**

Note All strings passed to or returned by MSMQ functions are in Unicode format (two-byte characters).

MQBeginTransaction

The **MQBeginTransaction** function creates an internal MSMQ transaction object that can be used to send messages to a queue or read messages from a queue.

```
HRESULT APIENTRY MQBeginTransaction(
  Transaction **ppTransaction
);
```

Parameters

ppTransaction
[out] Pointer to *Transaction* variable that points to the new transaction object.

Return Values

MQ_OK
Indicates success.

MQ_ERROR_INSUFFICIENT_RESOURCES
There are no resources to create a new transaction.

Remarks

The pointer returned by MQBeginTransaction can be used to set the *pTransaction* parameter of MQSendMessage or MQReceiveMessage.

For a description of internal transactions, see MSMQ Internal Transactions.

For an example of an internal transaction, see Sending Messages Using an Internal Transaction.

See Also

MQReceiveMessage, **MQSendMessage**

MQCloseCursor

The **MQCloseCursor** function closes a given cursor, allowing MSMQ to release the associated resources.

```
HRESULT APIENTRY MQCloseCursor(
  HANDLE hCursor
);
```

Parameters

hCursor
[in] Handle to the cursor you want to close.

Return Values

MQ_OK
Indicates success.

MQ_ERROR_INVALID_HANDLE
The cursor handle specified in *hCursor* is not valid.

Remarks

Typically, cursors are closed by calling **MQCloseCursor**. However, MSMQ automatically closes any cursor created for a given queue when the queue is closed. **MQCloseCursor** returns MQ_ERROR_INVALID_HANDLE when an attempt is made to close a cursor that is already closed by MSMQ.

To create a cursor, call **MQCreateCursor**.

Examples

For an example of how cursors are used when reading messages, see Reading Messages in a Queue.

For examples of using **MQCreateClose**, see Reading Messages Using a Cursor.

See Also

MQCreateCursor

MQCloseQueue

The **MQCloseQueue** function closes a given queue.

```
HRESULT APIENTRY MQCloseQueue(
  QUEUEHANDLE hQueue
);
```

Parameters

hQueue
[in] Handle to the queue you want to close.

Return Values

MQ_OK
Indicates success.

MQ_ERROR_INVALID_HANDLE
The queue handle specified in *hQueue* is not valid.

Remarks

When an application closes a queue, the queue handle becomes invalid, but the messages waiting in the queue remain in the queue. These includes any messages sent to the queue by the application closing the queue.

When **MQCloseQueue** is called, any cursors created for the queue are also closed.

Examples

For an example of using **MQCloseQueue**, see:

- Closing a Queue
- Sending Private Messages

See Also

MQOpenQueue

MQCreateCursor

The **MQCreateCursor** function creates a cursor for a specific queue and returns its handle. The cursor is used to maintain a specific location in a queue when reading the queue's messages.

```
HRESULT APIENTRY MQCreateCursor(
 QUEUEHANDLE hQueue,
   PHANDLE phCursor
);
```

Parameters

hQueue
[in] Handle to the queue where you want to create a cursor.

phCursor
[out] Pointer to a variable that receives the resulting cursor handle.

Return Values

MQ_OK
Indicates success.

MQ_ERROR_INVALID_HANDLE
The queue handle specified in *hQueue* is not valid.

MQ_ERROR_STALE_HANDLE
The specified queue handle was obtained in a previous session of the Queue Manager service. Close the queue and open it again to obtain a fresh handle.

Remarks

The **MQCreateCursor** function is used with **MQReceiveMessage** when you need to read messages that are not at the front of the queue. You do not need to create a cursor if you only want to read the first message in a queue.

For an example of how cursors are used when reading messages, see Reading Messages in a Queue.

For a description of how MSMQ uses cursors to navigate a queue, see Peeking at the next Message in a Queue or Retrieving a Message in a Queue

To close the cursor, call **MQCloseCursor**.

Examples

For examples of using **MQCreateCursor**, see Reading Messages Using a Cursor.

See Also

MQCloseCursor, **MQReceiveMessage**

MQCreateQueue

The **MQCreateQueue** function creates a queue and registers it in MQIS (for public queues) or on the local computer (for private queues). It also attaches the specified queue properties (default values are used for all properties that are not specified) and security attributes to the queue, and returns a format name that can be used to open the queue.

HRESULT APIENTRY MQCreateQueue(
  PSECURITY_DESCRIPTOR *pSecurityDescriptor*,
  MQQUEUEPROPS * *pQueueProps*,
  LPWSTR *lpwcsFormatName*,
  LPDWORD *lpdwFormatNameLength*
);

Parameters

pSecurityDescriptor

[in] Pointer to a **SECURITY_DESCRIPTOR** structure that specifies the security information associated with the queue (for information on the **SECURITY_DESCRIPTOR** structure, see the Microsoft Platform SDK). Following are the default values (NULL pointer indicates all default values are used):

Owner
: The process user.

Group
: The process group.

DACL
: Full control for the creator of the queue. All other processes can get queue properties, get queue security, and send messages to the queue.

SACL
: None.

pQueueProps

[in, out] Pointer to the **MQQUEUEPROPS** structure that specifies the created queue's properties.

On input, the **cProps** member of **MQQUEUEPROPS** specifies the number of queue properties supplied, the **aPropID** array specifies their property identifiers, and **aPropVar** array specifies their values.

On output, the optional **aStatus** array, if it was included in **MQQUEUPROPS**, indicates the status of the properties.

lpwcsFormatName

[out] Pointer to buffer to receive the format name for the queue (NULL pointer allowed). This buffer is called the format name buffer.

lpdwFormatNameLength

[in, out] On input, specifies the length of the *lpwcsFormatName* buffer (in Unicode characters). Public queues require at least 44 unicode characters; private queues require at least 54. NULL pointer is not allowed.

On output, indicates the length of the returned format name string, including the null-terminating character. If the output value is greater than the initial input value, the supplied buffer is not large enough to contain the complete format name string and MQ_INFORMATION_FORMATNAME_BUFFER_TOO_SMALL is returned. In this case, the queue is created and only a portion of the format name is returned.

Return Values

MQ_OK

Indicates success.

MQ_ERROR_ACCESS_DENIED
The process does not have the rights to create a queue on this computer.

MQ_ERROR_ILLEGAL_PROPERTY_VALUE
An illegal property value is specified.

MQ_ERROR_ILLEGAL_QUEUE_PATHNAME
PROPID_Q_PATHNAME contains an illegal MSMQ pathname string.

MQ_ERROR_ILLEGAL_SECURITY_DESCRIPTOR
The passed security descriptor has an invalid structure.

MQ_ERROR_INSUFFICIENT_PROPERTIES
No MSMQ pathname was specified (PROPID_Q_PATHNAME).

MQ_ERROR_INVALID_OWNER
The specified MSMQ pathname (PROPID_Q_PATHNAME) contains the name of an unrecognized machine.

MQ_ERROR_NO_DS
No connection with the Site Controller server. Cannot access the MQIS.

MQ_ERROR_PROPERTY
One or more properties resulted in an error.

MQ_ERROR_PROPERTY_NOTALLOWED
A specified property is not valid when creating the queue.

MQ_ERROR_QUEUE_EXISTS
Queue with an identical MSMQ pathname or instance already exists.

MQ_ERROR_SERVICE_NOT_AVAILABLE
Cannot connect to the Queue Manager.

MQ_ERROR_WRITE_NOT_ALLOWED
Cannot add a queue to MQIS while an MSMQ information store server is being installed.

MQ_INFORMATION_FORMATNAME_BUFFER_TOO_SMALL
The queue was created successfully, but the size of the buffer for receiving the format name of the created queue is too small.

MQ_INFORMATION_PROPERTY
The function completed, but one or more properties resulted in a warning.

Remarks

You must always specify the queue's MSMQ pathname (PROPID_Q_PATHNAME) before calling **MQCreateQueue**. The PROPID_Q_PATHNAME property tells MSMQ where to store the queue's messages, if the queue is public or private, and the name of the queue.

Public queues are registered in MQIS, and private queues are registered on the local computer. All queues exist until explicitly deleted.

Private queues can only be created on the local computer. When a private queue is created, MSMQ stores a description of the queue in the LQS directory on the local computer (by default, \program files\msmq\storage\lqs). Applications must ensure that no other private queues with the same name exist on a local computer. If a queue with the same name already exists, MSMQ will return an MQ_ERROR_QUEUE_EXISTS error when **MQCreateQueue** is called.

Setting other queue properties is optional. If a property is not specified in the *pQueueProps* parameter, MSMQ uses its default value when creating the queue. After the queue is created, its properties can be changed by calling **MQSetQueueProperties**.

To use a public queue's queue journal, pass PROPID_Q_JOURNAL and PROPID_Q_JOURNAL_QUOTA to **MQCreateQueue**. The journal keeps a copy of all messages retrieved from the public queue. For conceptual information about queue and machine journals, see

Journal Queues.

To create a transaction queue, pass PROPID_Q_TRANSACTION to **MQCreateQueue**. This property cannot be changed once the queue is created.

The queue's returned format name is invalid if **MQCreateQueue** fails for any reason, including returning the error MQ_ERROR_QUEUE_EXISTS. If the call returns MQ_INFORMATION_FORMATNAME_BUFFER_TOO_SMALL, this means that the queue was created, but the format name could not fit in the format name buffer.

After the queue is created, its properties can be retrieved using **MQGetQueueProperties** and reset using **MQSetQueueProperties**.

Access control for the queue is based on the MSMQ default security. For information on access control, see Access Control.

Foreign queues must be public queues. Their PROPID_Q_PATHNAME property must specify the name of the foreign computer as it is defined in MQIS. For information on connecting to foreign queues, see MSMQ Connector Server.

When creating public queues, some clients may not be able to detect the new queue registered in the MSMQ information even though the queue was registered. Changes to MQIS (such as creating a public queue) must be propagated from site to site, which can cause delays in the availability of current information. Consequently, clients at some sites may not be able to open the queue, even though it exists. Propagation delays, including communication network delays such as down links, are controlled by the MSMQ Administrator.

Public queues cannot be created by independent client computers running offline. For information on offline operations, see MSMQ Offline Support.

Examples

For an example of using **MQCreateQueue**, see Creating a Queue.

See Also

MQCloseQueue, **MQDeleteQueue**, **MQGetQueueProperties**, **MQLocateBegin**, **MQOpenQueue**, **MQSetQueueProperties**, PROPID_Q_INSTANCE, PROPID_Q_JOURNAL, PROPID_Q_PATHNAME, PROPID_Q_TRANSACTION

MQDeleteQueue

The **MQDeleteQueue** function deletes a queue from MQIS (in the case of public queues), or from the local computer (in the case of private queues).

HRESULT APIENTRY MQDeleteQueue(
  LPCWSTR *lpwcsFormatName*
);

Parameters

lpwcsFormatName
  [in] Pointer to the queue's format name buffer. This buffer contains the format name string of the queue to be deleted.

Return Values

MQ_OK
  Indicates success.

MQ_ERROR_ACCESS_DENIED
  The process does not have the access rights to delete this queue. To change access rights, call **MQSetQueueSecurity**.

MQ_ERROR_ILLEGAL_FORMATNAME
  The specified format name in *lpwcsFormatName* is illegal.

MQ_ERROR_NO_DS
  No connection with the Site Controller server. Cannot access the MQIS.

MQ_ERROR_SERVICE_NOT_AVAILABLE
  Cannot connect to the Queue Manager.

MQ_ERROR_UNSUPPORTED_FORMATNAME_OPERATION
  Cannot delete a queue using a direct format name.

MQ_ERROR_WRITE_NOT_ALLOWED
  Cannot delete a queue from MQIS while an MSMQ information store server is being installed.

Remarks

The format name of the queue (specified by *lpwcsFormatName)* cannot be a direct format name.

When deleting public queues, some clients may still see the queue registered in MQIS after the queue was deleted. Changes to MQIS (such as deleting a public queue) are propagated from site to site, which can cause delays in the availability of current information. Consequently, clients in some sites may still try to send messages to the queue, even though it was deleted. Propagation delays, including communication network delays such as down links, are controlled by the MSMQ Administrator.

Public queues cannot be deleted by independent client computers running offline. For information on offline operations, see MSMQ Offline Support.

Examples

For an example of using **MQDeleteQueue**, see Deleting a Queue.

See Also

MQCloseQueue, **MQCreateQueue**, **MQOpenQueue**, **MQSetQueueSecurity**

MQFreeMemory

The **MQFreeMemory** function frees memory allocated by MSMQ.

VOID MQFreeMemory(
  PVOID *pvMemory*
);

Parameters

pvMemory
  [in] Pointer to the memory to be freed.

Remarks

Whenever an application passes VT_NULL in an **aPropVar** array and MSMQ allocates memory for the returned property value (for example, *puuid* and *pwszVal*), **MQFreeMemory** must be called.

For **MQLocateNext**, all properties whose values returned by MSMQ are stored outside **aPropVar** and must also be freed using **MQFreeMemory**.

See Also

MQLocateNext

MQFreeSecurityContext

The **MQFreeSecurityContext** function frees the memory allocated by **MQGetSecurityContext**.

VOID APIENTRY MQFreeSecurityContext(
  HANDLE *hSecurityContext*
);

Parameters

hSecurityContext
  [in] Handle to the security context buffer.

Return Values

None.

Remarks

The security context buffer is created by **MQGetSecurityContext**. It contains information MSMQ needs to authenticate messages.

See Also

MQGetSecurityContext

MQGetMachineProperties

The **MQGetMachineProperties** function retrieves information about a Queue Manager computer.

HRESULT APIENTRY MQGetMachineProperties(
  LPCWSTR *lpwcsMachineName***,**
  GUID *pguidMachineID***,**
  MQQMPROPS *pQMProps*
);

Parameters

lpwcsMachineName
  [in] The name of the Queue Manager computer you want to access. If this parameter is used, set *pguidMachineID* to NULL.

pguidMachineID
  [in] The identifier of the Queue Manager computer you want to access. If this parameter is used, set *lpwcsMachineName* to NULL.

pQMProps
  [in, out] Pointer to a Queue Manager properties structure (**MQQMPROPS**), specifying which properties to retrieve.

Return Values

MQ_OK
  Indicates success.

MQ_ERROR_ACCESS_DENIED
  Access to the specified computer is denied. Verify the access rights for the operation.

MQ_ERROR_INVALID_PARAMETER
  Both name (*lpwcsMachineName)* and computer (*pguidMachineID*) identifiers were specified.

MQ_ERROR_ILLEGAL_MQQMPROPS
  Either *pQMprops* was NULL or no properties were specified.

MQ_ERROR_ILLEGAL_PROPERTY_VT
  An invalid type indicator was supplied for one of the property values in *pQMProps*.

MQ_ERROR_MACHINE_NOT_FOUND
  The specified computer could not be found in MQIS.

MQ_ERROR_NO_DS
  No connection with the Site Controller server. Cannot access the MQIS.

MQ_INFORMATION_UNSUPPORTED_PROPERTY
  An unsupported property identifier was specified in *pQMProps*.

MQ_INFORMATION_DUPLICATE_PROPERTY
  A duplicate property identifier was specified in *pQMProps*. The second entry is ignored.

Remarks

If *lpwcsMachineName* and *pguidMachineID* are set to NULL, the properties of the local computer are retreived.

Valid Queue Manager properties are:

- PROPID_QM_CONNECTION
- PROPID_QM_ENCRYPTION_PK
- PROPID_QM_MACHINE_ID
- PROPID_QM_PATHNAME

- PROPID_QM_SITE_ID

If a property specified by *pQMProps* is set to VT_NULL, MSMQ allocates the memory needed to store the property value when **MQGetMachineProperties** is called. However, when the returned property type replaces the VT_NULL value, the application must still free the memory allocated for the property value by calling **MQFreeMemory**.

MQGetMachineProperties is not supported for offline operations. For information on offline operations, see MSMQ Offline Support.

Examples

For an example of using **MQGetMachineProperties**, see:

- Reading Messages in a Dead Letter Queue
- Reading Messages in a Machine Journal

See Also

PROPID_QM_CONNECTION, PROPID_QM_ENCRYPTION_PK, PROPID_QM_MACHINE_ID, PROPID_QM_PATHNAME, PROPID_QM_SITE_ID, **MQFreeMemory**

MQGetQueueProperties

The **MQGetQueueProperties** function retrieves the specified properties for a specific queue.

```
HRESULT APIENTRY MQGetQueueProperties(
  LPCWSTR lpwcsFormatName,
  MQQUEUEPROPS *pQueueProps
);
```

Parameters

lpwcsFormatName

[in] Pointer to the format name string of the queue whose properties will be retrieved. Use a public or private format name to specify the queue. You cannot specify a queue using a direct format name.

pQueueProps

[in, out] Pointer to the **MQQUEUEPROPS** structure that specifies which properties to retrieve.

On input, the **cProps** member of **MQQUEUEPROPS** specifies the number of properties to be retrieved, and the **aPropID** array specifies the specific properties.

On output, the **aPropVar** array indicates the current values of the requested properties, and the optional **aStatus** array, if it was included in **MQQUEUPROPS**, indicates the status of the properties.

Return Values

MQ_OK

Indicates success.

MQ_ERROR_ACCESS_DENIED

The process does not have the correct access rights to get the queue's properties. For a complete list of queue access rights, see Access Control.

To change access rights, call **MQSetQueueSecurity**.

MQ_ERROR_ILLEGAL_FORMATNAME

The *lpwcsFormatName* parameter specified an illegal format name.

MQ_ERROR_ILLEGAL_PROPERTY_VT

The variant type of a property does not match the expected variant type. For example, for PROPID_Q_TYPE, the expected variant types are VT_NULL or VT_CLSID. For PROPID_Q_PATHNAME and PROPID_Q_LABEL, the expected variant type is VT_NULL.

MQ_ERROR_NO_DS

No connection with the Site Controller server. Cannot access the MQIS.

MQ_ERROR_PROPERTY

One or more properties resulted in an error.

MQ_ERROR_SERVICE_NOT_AVAILABLE

Cannot connect to the Queue Manager.

MQ_ERROR_UNSUPPORTED_FORMATNAME_OPERATION

The *lpwcsFormatName* parameter specified a direct format name. You cannot get properties from a queue by accessing it directly.

MQ_INFORMATION_DUPLICATE_PROPERTY

The same property appears more than once in the **aPropID** array, this information is returned on the second appearance of the property.

MQ_INFORMATION_PROPERTY

One or more of the properties resulted in a warning code even though the function was completed.

MQ_INFORMATION_UNSUPPORTED_PROPERTY

Not a valid property identifier. The property is ignored.

Remarks

All queue properties can be retrieved; however, you can only retrieve the properties of private queues if they are located on your local computer.

If the format name of the queue is unknown, see Format Name to find ways to obtain a new format name.

To retrieve the queue's MSMQ pathname or label, the corresponding property's variant type (PROPID_Q_PATHNAME or PROPID_Q_LABEL) must be initially set to VT_NULL. If it is not set to VT_NULL, the operation fails and MQ_ERROR_ILLEGAL_PROPERTY_VT is returned.

If a property value specified by *pQueueProps* is set to VT_NULL, MSMQ allocates the memory needed to store the returned value when **MQGetQueueProperties** is called. When this happens, the application must free the memory allocated for the returned property value by calling **MQFreeMemory**.

For a complete discussion on retrieving a queue's properties, see Retrieving a Queue's Properties Using API Functions.

A public queue's properties cannot be retrieved by independent client computers running offline. For information on offline operations, see MSMQ Offline Support.

See Also

MQFreeMemory, PROPID_Q_LABEL, PROPID_Q_PATHNAME

MQGetQueueSecurity

The **MQGetQueueSecurity** function retrieves the access control security descriptor for the specified queue.

```
HRESULT APIENTRY MQGetQueueSecurity(
  LPCWSTR lpwcsFormatName,
  SECURITY_INFORMATION *SecurityInformation,
  PSECURITY_DESCRIPTOR *pSecurityDescriptor,
  DWORD nLength,
  LPDWORD lpnLengthNeeded
);
```

Parameters

lpwcsFormatName

[in] Pointer to the format name string of the queue whose security information will be retrieved. Use a public or private format name to specify the queue. You cannot specify a direct format name.

SecurityInformation

[in] Specifies the **SECURITY_INFORMATION** structure that identifies the access control information being requested. (For information on the **SECURITY_INFORMATION** structure, see the Microsoft Platform SDK.)

pSecurityDescriptor

[out] Pointer to the security descriptor buffer that receives the queue's security descriptor. The calling process must have the rights to view the specified aspects of the queue's security status. The **SECURITY_DESCRIPTOR** structure is returned in self-relative format. (For information on the **SECURITY_DESCRIPTOR** structure, see the Microsoft Platform SDK.)

nLength

[in] Specifies the size, in bytes, of the security descriptor buffer (see *pSecurityDescriptor*).

lpnLengthNeeded

[out] Pointer to a variable that indicates if any additional length is needed for the security descriptor. If the security descriptor fits in the buffer, this variable indicates the actual size of the security descriptor.

If the security descriptor buffer is too small for the security descriptor (the value of *lpnLengthNeeded* is greater than the value of *nLength*), this variable indicates the size of the buffer needed to hold the security descriptor. When this happens, the security descriptor is not copied to the buffer and MQ_ERROR_SECURITY_DESCRIPTOR_BUFFER_TOO_SMALL is returned.

Return Values

MQ_OK

Indicates success.

MQ_ERROR_ACCESS_DENIED

The process does not have the correct access rights to get the queue's security descriptor. For a complete list of queue access rights, see Access Control.

To change access rights, call **MQSetQueueSecurity**.

MQ_ERROR_FUNCTION_NOT_SUPPORTED

MQGetQueueSecurity is not supported in Windows 95.

MQ_ERROR_ILLEGAL_FORMATNAME

The *lpwcsFormatName* parameter specified an illegal format name.

MQ_ERROR_NO_DS

No connection with the Site Controller server. Cannot access the MQIS.

MQ_ERROR_PRIVILEGE_NOT_HELD

The process does not have the proper privilege to read the queue's system access control list.

MQ_ERROR_SECURITY_DESCRIPTOR_BUFFER_TOO_SMALL
The buffer pointed by *pSecurityDescriptor* is too small to hold the security descriptor; the returned value of *lpnLengthNeeded* is greater than the supplied value of *nLength.*

MQ_ERROR_UNSUPPORTED_FORMATNAME_OPERATION
The *lpwcsFormatName* parameter contains a public or private queue using a direct format name, or a journal, dead letter, or connector queue .

Remarks

The queue's security descriptor is initially set when the queue is created (see **MQCreateQueue**). Access to the following queue operations can be controlled: creating, deleting, and opening the queue for sending messages to and reading messages from the queue; getting and setting the queue's properties; and getting and setting the queue's security descriptor.

The format name of the queue (specified by *lpwcsFormatName*) must be a public or private format name. MSMQ must be able to access MQIS (for public queues) or the local computer (for private queues) to get the queue's security descriptor.

If the format name of the queue is unknown, see Format Name to find ways to obtain a new format name.

To read the security descriptor of a queue, the calling process must have READ_CONTROL access or be the owner of the queue. Access rights such as READ_CONTROL are set when the queue is created and can be modified by calling **MQSetQueueSecurity**.

To read the queue's system access control list, the caller must have SE_SECURITY_NAME privileges on the MQIS server (for public queues) or on the local computer (for private queues).

MQGetQueueSecurity cannot retrieve the security descriptor of a journal, dead letter, connector, or foreign queues.

A public queue's security descriptor cannot be retrieved by independent client computers running offline. For information on offline operations, see MSMQ Offline Support.

See Also

MQSetQueueSecurity

MQGetSecurityContext

The **MQGetSecurityContext** function retrieves security information needed to authenticate messages.

```
VOID APIENTRY MQGetSecurityContext(
  LPVOID lpCertBuffer,
  DWORD dwCertBufferLength,
  HANDLE* hSecurityContext
);
```

Parameters

lpCertBuffer
[In] Pointer to the security certificate buffer. External certificates must be in ASN.1 DER encoded format. If this parameter is NULL, the internal security certificate provided by MSMQ is used.

dwCertBufferLength
[In] Length of the security certificate buffer pointed to by *lpCertBuffer*. For internal certificates, set to 0.

hSecurityContext
[Out] Handle to the security context buffer allocated by MSMQ.

Return Values

MQ_OK
Indicates success.

MQ_ERROR_COULD_NOT_GET_USER_SID
MSMQ could not get the specified sender identifier.

MQ_ERROR_NO_DS
Only Windows 95. No connection with the Site Controller server. Cannot access the MQIS.

MQ_ERROR_INVALID_PARAMETER
One of the IN parameters supplied by the operation is not valid.

MQ_ERROR_INSUFFICIENT_RESOURCES
Insufficient resources to complete operation (for example, not enough memory). Operation failed

MQ_ERROR_INVALID_CERTIFICATE
Security certificate specified by PROPID_M_SENDER_CERT is invalid, or the certificate is not correctly placed in the Microsoft® Internet Explorer personal certificate store.

MQ_ERROR_CORRUPTED_INTERNAL_CERTIFICATE
MSMQ-supplied internal certificate is corrupted.

MQ_ERROR_CORRUPTED_SECURITY_DATA
Cryptographic function (CryptoAPI) has failed.

Remarks

The **MQGetSecurityContext** function retrieves the information MSMQ needs to authenticate messages from the supplied certificate. It provides a way to send messages that require authentication in a more efficient way. **MQGetSecurityContext** should be used when the same certificate is used to send multiple messages and when impersonating another user.

Although the security information in the certificate can be retrieved directly by the sending application, this function retrieves and caches the needed information using a single function call. When **MQGetSecurityContext** is used, the sending application is only responsible for passing the security context buffer (PROPID_M_SECURITY_CONTEXT) to **MQSendMessage**.

When authenticating messages, MSMQ must track which sender certificate is associated with which message. Consequently, calling **MQSendMessage** must be done in the same user context as the call

to **MQGetSecurityContext**. If **MQGetSecurityContext** is not called before the message is sent (or PROPID_M_SECURITY_CONTEXT is not passed to **MQSendMessage**) the security context of the user who originally ran the process is used.

When more than one certificate is used, **MQGetSecurityContext** must be called for each certificate the sending application wants to use.

When impersonating another user, **MQGetSecurityContext** must be called before a message is sent. Once the security information for the impersonated user is retrieved, the sending application can revert to the original user and later use the impersonated security context to send the message, without the need to impersonate the user again.

To retrieve the security information of an impersonated user, HKEY_CURRENT_USER must point to the registry of the impersonated user. To do this, call the Win32 API function RegLoadKey() to load the impersonated user's registry hive. Call ReqCloseKey(HKEY_CURRENT_USER) to close the current user registry, then call ImpersonateLoggedOnUser() and **MQGetSecurityContext** to access the impersonated user registry to retrieve information about the impersonated user. The calls to RegCloseKey(), ImpersonateLoggedOnUser, MQGetSecurityContext, plus any other calls that may access the registry under HKEY_CURRENT_USER must be protected by the same critical section object.

Note For information on RegLoadKey(), RegCloseKey(), ImpersonateLoggedOnUser, and critical section objects, see the Platform SDK.

After the security certificate is no longer needed, free the memory allocated for the security context buffer by calling **MQFreeSecurityContext**.

Windows 95 applications cannot retrieve the security context of the certificate when operating on an independent client computer running offline. For information on offline operations, see MSMQ Offline Support.

See Also

MQFreeSecurityContext, **MQSendMessage**, PROPID_M_SECURITY_CONTEXT

MQHandleToFormatName

The **MQHandleToFormatName** function returns a format name for the queue based on its handle.

```
HRESULT APIENTRY MQHandleToFormatName(
  QUEUEHANDLE hQueue,
  LPWSTR lpwcsFormatName,
  LPDWORD lpdwCount
);
```

Parameters

hQueue
[in] Handle to the queue.

lpwcsFormatName
[out] Buffer to receive the format name for the queue.

lpdwCount
[in, out] On input, specifies the length of the *lpwcsFormatName* buffer (in Unicode characters). Public queues require at least 44 unicode characters; private queues require at least 54. NULL pointer is not allowed.

On output, indicates the length of the returned format name string, including the null-terminating character. If the output value is greater than the initial input value, the supplied buffer is not large enough to contain the complete format name string and MQ_ERROR_FORMATNAME_BUFFER_TOO_SMALL is returned. In this case, only a portion of the format name is returned.

Return Values

MQ_OK
Indicates success.

MQ_ERROR_FORMATNAME_BUFFER_TOO_SMALL
The *lpwcsFormatName* buffer is too small to contain the format name string.

MQ_ERROR_INVALID_HANDLE
The queue handle specified in *hQueue* is not valid.

MQ_ERROR_SERVICE_NOT_AVAILABLE
Cannot connect to the Queue Manager.

MQ_ERROR_STALE_HANDLE
The specified queue handle was obtained in a previous session of the Queue Manager service. Close the queue and open it again to obtain a fresh handle.

Remarks

Format names are not stored by MSMQ; the format name is created when **MQHandleToFormatName** is called.

Typically, this function is used when you need a format name to specify a queue when calling **MQGetQueueProperties**, **MQSetQueueProperties**, **MQGetQueueSecurity**, or **MQSetQueueSecurity**.

Other format name translation functions include **MQPathNameToFormatName** and **MQInstanceToFormatName**.

See Also

MQGetQueueProperties, **MQGetQueueSecurity**, **MQInstanceToFormatName**, **MQPathNameToFormatName**, **MQSetQueueProperties**

MQInstanceToFormatName

The **MQInstanceToFormatName** function returns a format name for the queue based on the identifier provided.

This function does not check to see if the identifier is valid.

```
HRESULT APIENTRY MQInstanceToFormatName(
  GUID * pGUID,
  LPWSTR lpwcsFormatName,
  LPDWORD lpdwCount
);
```

Parameters

pGUID
[in] Pointer to the queue identifier (a GUID structure).

lpwcsFormatName
[out] Pointer to a buffer to receive the format name for the queue.

lpdwCount
[in, out] On input, specifies the length of the *lpwcsFormatName* buffer (in Unicode characters). Public queues require at least 44 unicode characters; private queues require at least 54. NULL pointer is not allowed.

On output, indicates the length of the returned format name string, including the null-terminating character. If the output value is greater than the initial input value, the supplied buffer is not large enough to contain the complete format name string and MQ_ERROR_FORMATNAME_BUFFER_TOO_SMALL is returned. In this case, only a portion of the format name is returned.

Return Values

MQ_OK
Indicates success.

MQ_ERROR_SERVICE_NOT_AVAILABLE
Cannot connect to the Queue Manager.

MQ_ERROR_FORMATNAME_BUFFER_TOO_SMALL
The buffer pointed to by *lpwcsFormatName* is too small to contain the format name string.

Remarks

Format names are not stored by MSMQ; the format name is created when **MQInstanceToFormatName** is called.

This function is used when you need a format name to specify a queue when calling **MQOpenQueue**, **MQGetQueueProperties**, **MQSetQueueProperties**, **MQGetQueueSecurity**, or **MQSetQueueSecurity**, and the only available information is the queue's identifier (PROPID_Q_INSTANCE). Typically, this happens when **MQLocateBegin** or **MQLocateNext** locates a queue and stores its identifier, not the queue's format name, in MQIS.

Other format name translation functions include **MQPathNameToFormatName** and **MQHandleToFormatName**.

See Also

MQGetQueueProperties, **MQGetQueueSecurity**, **MQHandleToFormatName**, **MQLocateBegin**, **MQLocateNext**, **MQOpenQueue**, **MQPathNameToFormatName**, **MQSetQueueProperties**, PROPID_Q_INSTANCE

MQLocateBegin

The **MQLocateBegin** function starts a query to locate a single public queue (or set of public queues), returning a query handle. Use **MQLocateNext** to retrieve the query results.

This function does not return the number of matching entries.

```
HRESULT APIENTRY MQLocateBegin(
  LPCWSTR lpwcsContext,
  MQRESTRICTION *pRestriction,
  MQCOLUMNSET *pColumns,
  MQSORTSET *pSort,
  PHANDLE phEnum
);
```

Parameters

lpwcsContext
[in] Specifies the starting point of the query within MQIS. Must be NULL.

pRestriction
[in] Specifies the search criteria for the query. NULL value indicates no restrictions.

pColumns
[in] Specifies which queue properties should be returned by the query. Must not be set to NULL.

pSort
[in] Specifies the sort order for the query results. PROPID_Q_PATHNAME cannot be used as a sort key. Can be set to NULL to indicate no sort order is needed.

phEnum
[out] Pointer to a query handle to use when calling **MQLocateNext** and **MQLocateEnd**.

Return Values

MQ_OK
Indicates success.

MQ_ERROR_ILLEGAL_CONTEXT
Indicates that *lpwcsContext* is not NULL.

MQ_ERROR_ILLEGAL_MQCOLUMNS
Indicates that *pColumns* is NULL.

MQ_ERROR_ILLEGAL_PROPERTY_VALUE
Indicates that an illegal property value was specified in *pRestriction.* For example, this error is returned if PROPID_Q_LABEL is specified and the supplied queue label is longer than the maximum label length.

MQ_ERROR_ILLEGAL_PROPID
Indicates that an illegal property identifier was specified in *pColumns*.

MQ_ERROR_ILLEGAL_RELATION
Indicates that an invalid relationship value was specified in *pRestriction*.

MQ_ERROR_ILLEGAL_RESTRICTION_PROPID
Indicates that an illegal property identifier was specified in *pRestriction.* For example, PROPID_Q_PATHNAME is not valid.

MQ_ERROR_ILLEGAL_SORT_PROPID
Indicates that an illegal property identifier was specified in *pSort*. For example, if *pSort* specifies PROPID_Q_PATHNAME, this error is returned.

MQ_ERROR_NO_DS
No connection with the Site Controller server. Cannot access the MQIS.

Remarks

Queries can only locate public queues. **MQLocateBegin** cannot locate a private queue. For a complete description of running a query, see Locating a Public Queue.

The *pRestriction* parameter specifies the search criteria (which queue properties and values are used in the search). It is a pointer to a structure containing a logically **AND**ed list of property restrictions, with each restriction including a queue property identifier, a property value, and a comparison operator. Comparison operators include: less than (PRLT), less than or equal to (PRLE), equal (PREQ), not equal (PRNE), greater than or equal to (PRGE), greater than (PRGT).

To specify the search criteria, MSMQ uses two structures: **MQPROPERTYRESTRICTION** and **MQRESTRICTION**. **MQPROPERTYRESTRICTION** defines a single property restriction. **MQRESTRICTION** specifies an array of restrictions as well as a count of how many restrictions there are (see the following examples).

This example locates queues by their type of service. For example, locating all the queues whose PROPID_Q_TYPE equals *guidx*.

```
/*Set property restriction. */
MQPROPERTYRESTRICTION PropertyRestriction;
PropertyRestriction.rel           = PREQ;
PropertyRestriction.prop          = PROPID_Q_TYPE;
PropertyRestriction.prval.vt      = VT_CLSID;
PropertyRestriction.prval.puuid = &guidX;

/*Package restrictions. */
MQRESTRICTION rest;
rest.cRes = 1;
rest.paPropRes = &PropertyRestriction;
```

This example locates queues by their type of service (PROPID_Q_TYPE equals *guidx*) and by their message journal property (PROPID_Q_JOURNAL equals 1).

First, prepare the property restrictions:

```
/*Set property restrictions. */
MQPROPERTYRESTRICTION PropertyRestrictions[2];
/*Set first restriction. */
PropertyRestrictions[0].rel = PREQ;
PropertyRestrictions[0].prop = PROPID_Q_TYPE;
PropertyRestrictions[0].prval.vt   = VT_CLSID;
PropertyRestrictions[0].prval.puuid = &guidX;
/*Set second restriction. */
PropertyRestrictions[1].rel = PREQ;
PropertyRestrictions[1].prop = PROPID_Q_JOURNAL;
PropertyRestrictions[1].prval.vt   = VT_UI1;
PropertyRestrictions[1].prval.bVal = 1;

/*Package the restrictions. */
MQRESTRICTION rest;
rest.cRes = 2;
rest.paPropRes = PropertyRestrictions;
```

Setting *pRestriction* to NULL retrieves information about all the queues.

The *pColumns* parameter allows you to specify which queue properties to retrieve. You can retrieve any number of queue properties with the same call to **MQLocateBegin**.

The *pSort* parameter allows you to specify the sort order (ascending or descending) of the result according to one or more of the properties (PROPID_Q_PATHNAME cannot be used as a sort key). Two structures are used to set the sort order: **MQSORTKEY** specifies a single sort key, and **MQSORTSET** specifies an array of sort keys along with the number of keys.

For example, the following code sorts queues according to their quota:

```
/* Prepare sort key. */
MQSORTKEY SortKey;
SortKey.propColumn = PROPID_Q_QUOTA;
SortKey.dwOrder = QUERY_SORTASCEND;

/* Indicate number of sort keys. */
MQSORTSET SortSet;
SortSet.cCol = 1;
SortSet.aCol = &SortKey;
```

When running a query, MSMQ can locate queues faster when the query is based on PROPID_Q_INSTANCE, PROPID_Q_TYPE, or PROPID_Q_LABEL (PREQ only). The query runs faster because these properties are indexed in MQIS, providing a faster way for MSMQ to locate the property specified in the call.

MQLocateBegin can only return queues that are in MQIS when **MQLocateBegin** is called. Queues created after **MQLocateBegin** is called are not included.

MQLocateBegin is not supported for offline operations. For information on offline operations, see MSMQ Offline Support.

Examples

For an example of using **MQLocateBegin**, see Locating a Public Queue.

See Also

MQCOLUMNSET, **MQLocateEnd**, **MQLocateNext**, **MQPROPERTYRESTRICTION**, **MQRESTRICTION**, **MQSORTKEY**, **MQSORTSET**, PROPID_Q_JOURNAL, PROPID_Q_LABEL, PROPID_Q_PATHNAME, PROPID_Q_TYPE

MQLocateEnd

The **MQLocateEnd** function ends a query, releasing the resources associated with the query.

```
HRESULT APIENTRY MQLocateEnd(
  HANDLE hEnum
);
```

Parameters

hEnum
[in] Query handle returned by a call to **MQLocateBegin**.

Return Values

MQ_OK
Indicates success.

MQ_ERROR_INVALID_HANDLE
The query handle specified in *hEnum* is not valid.

Remarks

MQLocateEnd is not supported for offline operations. For information on offline operations, see MSMQ Offline Support.

Examples

For an example of using **MQLocateEnd**, see Locating a Public Queue.

See Also

MQLocateBegin, **MQLocateNext**

MQLocateNext

The **MQLocateNext** function retrieves the requested queue information from the query. It is called after obtaining a query handle from a previous call to **MQLocateBegin**.

```
HRESULT APIENTRY MQLocateNext(
  HANDLE hEnum,
  DWORD * pcProps,
  PROPVARIANT aPropVar[]
);
```

Parameters

hEnum

[in] Query handle returned by a previous call to **MQLocateBegin**.

pcProps

[in, out] On input, a pointer to a variable that specifies the number of elements in the *aPropVar[]* array.

On return, *pcProps* holds the number of properties returned to the query. **MQLocateNext** returns as many completed sets of properties (the number of properties returned for each queue) as possible. A returned value of 0 indicates no queues were found.

aPropVar

[out] Holds the values of the retrieved properties in an array of **PROPVARIANT**. For each property returned, **MQLocateNext** sets the **VT** field and the corresponding **union** member of the appropriate *aPropVar[]* element. **MQFreeMemory** must be called to free memory allocated by MSMQ, which happens when an **PROPVARIANT** element involves allocation of memory (for example **lpwstr**, **GUID**).

Return Values

MQ_OK

Indicates success.

MQ_ERROR_INVALID_HANDLE

The query handle specified in *hEnum* is not valid.

MQ_ERROR_NO_DS

No connection with the Site Controller server. Cannot access the MQIS.

MQ_ERROR_RESULT_BUFFER_TOO_SMALL

The supplied buffer for *aPropVar* is too small. **MQLocateNext** could not return at least one complete query result.

Remarks

The **MQLocateNext** function is called after obtaining a query handle from a previous call to **MQLocateBegin**. The call can be repeated, using the same query handle, until all the results of the query are received (until *pcProps* is 0). For a complete description of running a query, see Locating a Public Queue.

The **MQLocateNext** function returns as many completed results (the number of properties requested in **MQLocateBegin**) as possible. Consequently, you should always specify a multiple of the number of requested properties when setting *pcProps*. By using a multiple of the requested properties, allocated space for these properties is not wasted.

MSMQ only returns information for those queues that the calling application has access rights (MQSEC_GET_QUEUE_PROPERTIES) to.

MQLocateNext is not supported for offline operations. For information on offline operations, see

MSMQ Offline Support.

Examples

For an example of using **MQLocateNext**, see Locating a Public Queue.

See Also

MQFreeMemory, **MQGetQueueProperties**, **MQLocateBegin**, **MQLocateEnd**, **MQSetQueueSecurity**

MQOpenQueue

The **MQOpenQueue** function opens a queue for sending messages to the queue or for reading its messages.

HRESULT APIENTRY MQOpenQueue(
  LPCWSTR *lpwcsFormatName***,**
  DWORD *dwAccess***,**
  DWORD *dwShareMode***,**
  LPQUEUEHANDLE *phQueue*
);

Parameters

lpwcsFormatName
[in] Pointer to the format name string of the queue you want to open. The format name can be in a public, private, or direct format. See the Remarks section for details on direct format names.

dwAccess
[in] Specifies how the application accesses the queue (peek, send, or receive). This setting cannot be changed while the queue is open.

Specify one of the following access modes:

MQ_PEEK_ACCESS
Messages can only be looked at. They cannot be removed from the queue.

MQ_SEND_ACCESS
Messages can only be sent to the queue.

MQ_RECEIVE_ACCESS
Messages can be looked at and removed from of the queue. Whether a message is removed from the queue or looked at depends on the *dwAction* parameter of **MQReceiveMessage**.

See *dwShareMode* for limiting who can receive the messages.

dwShareMode
[in] How the queue will be shared. Specify one of the following:

MQ_DENY_NONE
Default. The queue is available to everyone. This setting must be used if *dwAccess* is set to MQ_SEND_ACCESS.

MQ_DENY_RECEIVE_SHARE
Limits those who can receive messages from the queue to this process. If the queue is already opened for receiving messages by another process, this call fails and returns MQ_ERROR_SHARING_VIOLATION. Applicable only when *dwAccess* is set to MQ_RECEIVE_ACCESS or MQ_PEEK_ACCESS.

phQueue
[out] Handle to the opened queue.

Return Values

MQ_OK
Indicates success.

MQ_ERROR_SERVICE_NOT_AVAILABLE
Cannot connect to the Queue Manager.

MQ_ERROR_ILLEGAL_FORMATNAME
The *lpwcsFormatName* parameter specified an illegal format name.

MQ_ERROR_NO_DS
No connection with the Site Controller server. Cannot access the MQIS.

MQ_ERROR_INVALID_PARAMETER
One of the IN parameters is not valid.

MQ_ERROR_SHARING_VIOLATION
Another process already opened this queue with *dwShareMode* set to MQ_DENY_RECEIVE_SHARE, or another process has already opened the queue for receive so you can't specify MQ_DENY_RECEIVE_SHARE.

MQ_ERROR_ACCESS_DENIED
The calling process does not have the required access rights to open the queue with the access mode specified by dwAccess.

MQ_ERROR_UNSUPPORTED_ACCESS_MODE
The access mode specified by *dwAccess* is not supported. Set *dwAccess* to MQ_PEEK_MESSAGE, MQ_SEND_MESSAGE, or MQ_RECEIVE_MESSAGE and call **MQOpenQueue** again.

MQ_ERROR_UNSUPPORTED_FORMATNAME_OPERATION
The *lpwcsFormatName* parameter specified a format name that is not supported by the access rights specified in *dwAccess*. See the following Remarks section for restrictions on using direct format names.

Remarks

If the format name of the queue is unknown, see Format Name to find ways to obtain a new format name.

Direct format names can only be used when sending messages to a queue. A direct format name instructs MSMQ not to use MQIS (for public queues) or the local computer (for private queues) to get routing information. When a direct format name is used, all routing information is derived from the format name and MSMQ sends the messages to the queue in a single hop.

Setting *dwShareMode* to MQ_DENY_RECEIVE_SHARE indicates that until the calling application calls **MQCloseQueue**, no other MSMQ applications can open a queue with receive access.

When opening a queue on a remote computer, MSMQ does not check for the existence of the queue when *dwAccess* is set to MQ_SEND_ACCESS. In addition, if *dwAccess* is set to MQ_RECEIVE_ACCESS, the computer opening the queue must support the same protocol as the remote computer where the queue is located.

Journal queues and dead letter queues can only be opened with *dwAccess* set to MQ_PEEK_ACCESS or MQ_RECEIVE_ACCESS. You cannot send messages to a journal queue.

With one exception, foreign queues are opened the same way queues located within the enterprise are opened. Applications cannot open a foreign queue using a direct format name. MSMQ needs the routing information stored in MQIS to find a MSMQ Connector Sever for the foreign queue.

If the calling application does not have sufficient access rights to a queue, the following two things can happen:

- If *dwAccess* is set to MQ_ACCESS_SEND, **MQOpenQueue** will succeed, but errors will be returned when the application tries to send a message.
- If *dwAccess* is set to MQ_PEEK_ACCESS or MQ_RECEIVE_ACCESS, **MQOpenQueue** will fail and return MQ_ERROR_ACCESS_DENIED. In this case a queue handle is not returned to *phQueue*.

To change the access rights of the queue, call **MQSetQueueSecurity**. The following table lists the access right needed to open the queue in peek, send, or receive access mode.

Queue Access Mode	Queue Access Right
MQ_PEEK_MESSAGE	MQSEC_PEEK_MESSAGE
MQ_SEND_MESSAGE	MQSEC_WRITE_MESSAGE
MQ_RECEIVE_MESSAGE	MQSEC_RECEIVE_MESSAGE

There is no provision to change the access mode of the queue when it is open. Either close and open the queue with the desired access mode, or open a second instance of the queue.

For Windows NT

For Windows NT, a queue handle is always inherited by a child process. If a child process is created by the process that opened the queue, the queue handle is inherited by the child process.

For Windows 95

A queue handle is not inherited by a child process.

Examples

For examples of using **MQOpenQueue**, see:

- Opening a Queue
- Reading Messages Using a Cursor
- Reading Messages in a Dead Letter Queue
- Reading Messages in a Machine Journal
- Sending Private Messages

See Also

MQCloseQueue, **MQReceiveMessage**, **MQSetQueueSecurity**

MQPathNameToFormatName

The **MQPathNameToFormatName** function returns a format name based on the MSMQ pathname provided.

```
HRESULT APIENTRY MQPathNameToFormatName(
  LPCWSTR lpwcsPathName,
  LPWSTR lpwcsFormatName,
  LPDWORD lpdwCount
);
```

Parameters

lpwcsPathName
[in] Pathname of the queue. Either private or public MSMQ pathnames are valid.

lpwcsFormatName
[out] Pointer to a buffer to receive the format name for the queue.

lpdwCount
[in, out] On input, specifies the length of the *lpwcsFormatName* buffer (in Unicode characters). Public queues require at least 44 unicode characters; private queues require at least 54.

On output, indicates the length of the returned format name string, including the null-terminating character. If the output value is greater than the initial input value, the supplied buffer is not large enough to contain the complete format name string and MQ_ERROR_FORMATNAME_BUFFER_TOO_SMALL is returned. In this case, only a portion of the format name is returned.

Return Values

MQ_OK
Indicates success.

MQ_ERROR_ILLEGAL_QUEUE_PATHNAME
The *lpwcsPathName* parameter contains an illegal MSMQ pathname string.

MQ_ERROR_SERVICE_NOT_AVAILABLE
Cannot connect to the Queue Manager.

MQ_ERROR_NO_DS
No connection with the Site Controller server. Cannot access the MQIS.

MQ_ERROR_FORMATNAME_BUFFER_TOO_SMALL
The buffer pointed to by *lpwcsFormatName* is too small to contain the format name string.

Remarks

Private (local only) and public MSMQ pathnames can be specified.

Typically, this function is used when you need a format name to specify a queue when calling **MQOpenQueue**, **MQGetQueueProperties**, **MQSetQueueProperties**, **MQGetQueueSecurity**, or **MQSetQueueSecurity** and the only available information is the queue's MSMQ pathname.

The format name of a public queue cannot be returned by an independent client computer running offline. For information on offline operations, see MSMQ Offline Support.

Other format name translation functions include **MQInstanceToFormatName** and **MQHandleToFormatName**.

Examples

For an example of using **MQPathNameToFormatName**, see Sending Private Messages.

See Also

MQGetQueueProperties, **MQGetQueueSecurity**, **MQHandleToFormatName**, **MQInstanceToFormatName**, **MQOpenQueue**, **MQSetQueueProperties**, **MQSetQueueSecurity**

MQReceiveMessage

The **MQReceiveMessage** function allows you to read messages in the queue. When reading messages, you can either peek at (not removing them) or retrieve the messages (removing them) in the queue.

Messages can be read either synchronously, asynchronously, or through a transaction.

```
HRESULT APIENTRY MQReceiveMessage(
  QUEUEHANDLE hSource,
  DWORD dwTimeout,
  DWORD dwAction,
  MQMSGPROPS  pMessageProps,
  LPOVERLAPPED lpOverlapped,
  PMQRECEIVECALLBACK fnReceiveCallback,
  HANDLE hCursor,
  Transaction *pTransaction
);
```

Parameters

hSource

[in] Handle to the queue that contains the message. For transactions, specify a queue on a local computer.

dwTimeout

[in] Time, in milliseconds, to wait for the message. Can be set to INFINITE.

dwAction

[in] How the message is read in the queue. Specify one of the following:

MQ_ACTION_RECEIVE

Reads the message at the current cursor location and removes it from the queue.

MQ_ACTION_PEEK_CURRENT

Reads the message at the current cursor location but does not remove it from the queue. The cursor remains pointing at the current message.

If a cursor was not created by **MQCreateCursor** (*hCursor* is NULL), the queue's cursor can only point to the first message in the queue.

MQ_ACTION_PEEK_NEXT

Reads the next message in the queue (skipping the message at the current cursor location) but does not remove it from the queue.

MQCreateCursor must be called (*hCursor* is not NULL) before MQ_ACTION_PEEK_NEXT can be used.

pMessageProps

[in, out] On input, a pointer to an **MQMSGPROPS** structure that specifies which message properties will be received. Can be set to NULL.

On output, it contains the received message property values.

lpOverlapped

[in, out] Pointer to an **OVERLAPPED** structure. Set to NULL for synchronous receive and transactions.

fnReceiveCallback

[in] Pointer to the callback function. Set to NULL for synchronous receive and transactions.

hCursor

[in] Handle to cursor for looking at messages in the queue. Can be set to NULL. See the following Remarks section

pTransaction

[in] Must be a pointer to a transaction object, a constant, or NULL.

Transaction object can be obtained internally from MSMQ (by calling **MQBeginTransaction**), or externally from Microsoft® DTC (Distributed Transaction Coordinator).

Constants include:

MQ_NO_TRANSACTION
Specifies that the call is not part of a transaction.

MQ_MTS_TRANSACTION
Specifies that the current MTS (Microsoft® Transaction Server) transaction is used to retrieve the message.

MQ_XA_TRANSACTION
Specifies that the call is part of an externally coordinated, XA-compliant transaction.

NULL indicates the message is not retrieved as part of a transaction.

Return Values

MQ_OK
Indicates success.

MQ_ERROR_ACCESS_DENIED
The action specified in *dwAction* does not agree with the access rights the queue was opened with.

MQ_ERROR_BUFFER_OVERFLOW
The supplied buffer for the message body is too small. The part of the message body that fits into the buffer is copied, but the message is not removed from the queue.

MQ_ERROR_SENDERID_BUFFER_TOO_SMALL
The supplied sender identification buffer is too small to hold the sender identification.

MQ_ERROR_SYMM_KEY_BUFFER_TOO_SMALL
The supplied symmetric key buffer is too small to hold the symmetric key.

MQ_ERROR_SENDER_CERT_BUFFER_TOO_SMALL
The supplied sender certificate buffer is too small to hold security certificate.

MQ_ERROR_SIGNATURE_BUFFER_TOO_SMALL
The supplied signature buffer is too small to hold the message's digital signature.

MQ_ERROR_PROV_NAME_BUFFER_TOO_SMALL
The supplied provider name buffer is too small to hold cryptographic service provider's name.

MQ_ERROR_LABEL_BUFFER_TOO_SMALL
The supplied message label buffer is too small to hold the message's label.

MQ_ERROR_FORMATNAME_BUFFER_TOO_SMALL
The supplied format name buffer is too small to hold the format name of the queue.

MQ_ERROR_DTC_CONNECT
MSMQ was unable to connect to the MS DTC.

MQ_ERROR_INSUFFICIENT_PROPERTIES
One of the following message properties was specified (in *pMessageProps*) without its associated length property: PROPID_M_ADMIN_QUEUE, PROPID_M_DEST_QUEUE, PROPID_M_LABEL, PROPID_M_RESP_QUEUE, PROPID_M_XACT_STATUS_QUEUE, or PROPID_M_PROV_NAME

MQ_ERROR_INVALID_HANDLE
The queue handle specified in *hSource* is not valid.

MQ_ERROR_IO_TIMEOUT
No message was received within the timeout period specified by *dwTimeout*.

MQ_ERROR_MESSAGE_ALREADY_RECEIVED
A message that is currently pointed at by the cursor has been removed from the queue. It can be removed by another process, by another call to **MQReceiveMessage** using a different cursor, or the message time-to-be-received timer has expired.

MQ_ERROR_OPERATION_CANCELLED
The operation was canceled before it could be completed. For example, the queue handle was closed by another thread while waiting for a message.

MQ_ERROR_PROPERTY
One or more message properties specified in *pMessageProps* resulted in an error.

MQ_ERROR_QUEUE_DELETED
The queue was deleted before the message could be read. The specified queue handle is no longer valid and the queue handle must be closed.

MQ_ERROR_ILLEGAL_CURSOR_ACTION
MQ_ACTION_PEEK_NEXT cannot be used with the current cursor position.

MQ_ERROR_SERVICE_NOT_AVAILABLE
Cannot connect to the Queue Manager.

MQ_ERROR_STALE_HANDLE
The specified queue handle was obtained in a previous session of the Queue Manager service. Close the queue and open it again to obtain a fresh handle.

MQ_ERROR_TRANSACTION_USAGE
Transaction error. Either reading the message is part of a transaction and its queue is a non-transaction queue, or reading the message is not part of a transaction and the queue is a transaction queue.

MQ_INFORMATION_PROPERTY
One or more of the properties specified in *pMessageProps* resulted in a warning code even though the function is completed.

Remarks

All message properties can be read. However, only those properties specified in the *pMessageProps* parameter are returned to the calling application; other properties are simply discarded when the message is read. For example, when browsing for messages some applications may choose to retrieve the size of the message without retrieving the message body itself. To do this, PROPID_M_BODY_SIZE is included in *pMessageProps* and PROPID_M_BODY is not; the size of the message is returned to the calling application and the message body is not.

The *hCursor* parameter contains the handle to a cursor created by **MQCreateCursor**. Using a cursor is optional, and is only needed when you want to read messages that are not at the front of the queue.

When using a cursor, you must peek at the first message in the queue by setting *dwAction* to MQ_ACTION_PEEK_CURRENT, followed by subsequent calls with *dwAction* set to MQ_ACTION_PEEK_NEXT.

The *dwAction* parameter specifies how MSMQ reads the message (either peek or receive) and which message is read. For a description of how MSMQ reads the messages in the queue, see:

- Peeking at a Message in a Queue
- Peeking at the next Message in a Queue
- Retrieving the First Message
- Retrieving a Message in a Queue

To retrieve the message body, PROPID_M_BODY must be specified in *pMessageProps*. The **VT** field of the corresponding element in the **aPropVar** array should be set to VT_UI1 | VT_VECTOR, allowing MSMQ to use the buffer specified in **CAUI1** to store the message. If the supplied buffer is too small to contain the entire message body, **MQReceiveMessage** fails and MQ_ERROR_BUFFER_OVERFLOW is returned. The buffer is filled to capacity and the full message remains in the queue. When this happens, the other properties specified by *pMessageProps* are still read.

To retrieve the size of the message, specify PROPID_M_BODY_SIZE in *pMessageProps*. MSMQ sets PROPID_M_BODY_SIZE to the size of the message body, even if **MQReceiveMessage** fails because

the message body exceeded the buffer allocated by PROPID_M_BODY. When retrieving the message body size, the **CAUI1**structure associated with the PROPID_M_BODY property does not indicate the size. The *cElems* field of the **CAUI1** structure merely indicates the maximum message body, which could be copied into the *pElems* buffer. The *cElems* field is never modified by MSMQ.

Not all properties require the application to specify the property type in the **VT** field of the **aPropVar** array. For these properties, the corresponding **VT** field in the **aPropVar** array can be set to VT_NULL.

When reading messages in a queue, the function's timeout timer (*dwTimeout*) can be set to 0, a specific amount of time, or INFINITE. A message can be retrieved if it is available at that period of time.

Synchronously Reading Messages

To synchronously read messages, *fnReceiveCallback* and *lpOverlapped* must be set to NULL. When this is done, the calling thread is blocked until a suitable message is available or a timeout occurs.

For an example of using **MQReceiveMessage** to read messages synchronously, see Reading Messages Synchronously.

Asynchronously Reading Messages

When asynchronously reading messages, **MQReceiveMessage** returns a SUCCESS value as soon as a suitable message is found. Otherwise, the function returns immediately with the return value MQ_INFORMATION_OPERATION_PENDING. This return value indicates that the operation is pending and will be completed as soon as a suitable message can be found. Asynchronous receive is based on standard Microsoft Win32 mechanisms.

There are three possible ways to read messages asynchronously:

- By using a callback function (*fnReceiveCallback* is not NULL). In this case, the message is not retrieved by **MQReceiveMessage**. It is retrieved by the registered callback function. Once registered, a callback function cannot be un-registered.

  When multiple asynchronous **MQReceiveMessage** calls are outstanding, several callbacks are registered; upon arrival of a message, the first registered callback is called.
- By using a Windows Event mechanism (*fnReceiveCallback* is NULL and *lpOverlapped* is not NULL). In this case, the **hEvent** member of the specified **OVERLAPPED** structure contains a valid handle to an event object. When a suitable message arrives, or a timeout occurs, the event object is set to the signaled state.

  For more information on the **OVERLAPPED** structure, see the Microsoft Platform SDK.
- By using a Windows NT completion port mechanism (*fnReceiveCallback* is NULL and *lpOverlapped* is not NULL). A queue handle can be associated with a completion port to receive messages asynchronously.

  For more information, see **CreateIOCompletionPort** in the Microsoft Platform SDK.

The output parameters to an asynchronous call to **MQReceiveMessage** should be kept intact until the operation completes, that is, you cannot free them or reuse them. Use automatic variables with caution.

For an example of using **MQReceiveMessage** to read messages asynchronously, see Reading Messages Asynchronously.

Reading message in Transactions

If **MQReceiveMessage** is called as part of a transaction (*pTransaction* is not set to MQ_NO_TRANSACTION or NULL), the following parameters must be set accordingly:

- The *lpOverlapped* and *fnReceiveCallback* parameters must be set to NULL. The operation must be synchronous receive.
- The *hSource* parameter must specify a queue on a local computer.

When the call is made, MSMQ performs the following tasks.

- In the case of a subsequent Abort, the message is returned to its original place in the queue.
- In the case of a Commit, a positive acknowledgment message is sent to the sender's administration queue. The class property of the acknowledgment message is MQMSG_CLASS_ACK_RECEIVE. For information on ITransaction::Commit, see the Platform SDK.

For more information about MSMQ transactions, see MSMQ Transactions.

Messages in Administration Queues

When reading acknowledgment messages in an administration queue, you can see if the original message failed by looking at the class property (PROPID_M_CLASS) of the acknowledgment message. The class property will contain a positive or negative acknowledgment.

If the class property is positive, the original message body is not included in the acknowledgment message. If the class property is negative, the message body is included as the message body of the acknowledgment message. For a complete description of all the properties of the acknowledgment message, see Acknowledgment Messages.

Responding to Messages

The receiving application can pass PROPID_M_RESP_QUEUE to **MQReceiveMessage** to see if the sending application expects a response to the message. The messages sent back to the response queue specified by this property, must be understood by the original sending application.

When receiving a message, always check PROPID_M_RESP_QUEUE to see if it is non-NULL. If it is not NULL, send responses to the specified response queue.

Response queue handles returned by **MQOpenQueue** can be cached to eliminate the need to call **MQOpenQueue** several times for the same response queue.

For an example of sending response messages, see Sending Messages that Request a Response.

Examples

For examples of using **MQReceiveMessage**, see Reading Messages Using a Cursor.

See Also

MQCreateCursor, **MQMSGPROPS**, **MQOpenQueue**, **PROPVARIANT**, **MQSetQueueSecurity**, PROPID_M_BODY, PROPID_M_BODY_SIZE, PROPID_M_CLASS, PROPID_M_RESP_QUEUE

MQSendMessage

The **MQSendMessage** function sends a message to the queue corresponding to the handle *hDestinationQueue*.

```
HRESULT APIENTRY MQSendMessage(
  QUEUEHANDLE hDestinationQueue,
  MQMSGPROPS * pMessageProps,
  ITransaction * pTransaction
);
```

Parameters

hDestinationQueue

[in] Handle to the queue where you want to send the message.

pMessageProps

[in] Pointer to an **MQMSGPROPS** structure describing the message to send.

pTransaction

[in] Must be a pointer to a transaction object, a constant, or NULL.

Transaction object can be obtained internally from MSMQ (by calling **MQBeginTransaction**), or externally from Microsoft® Distributed Transaction Coordinator (MS DTC).

Constants include:

MQ_NO_TRANSACTION
: Specifies that the call is not part of a transaction.

MQ_MTS_TRANSACTION
: Specifies that the current Microsoft Transaction Server (MTS) transaction is used to send the message.

MQ_SINGLE_MESSAGE
: Sends a single message as a transaction. Messages sent as a single-message transaction must be sent to a transaction queue.

MQ_XA_TRANSACTION
: Specifies that the call is part of an externally coordinated, XA-compliant transaction.

NULL indicates the message is not sent as part of a transaction.

Return Values

MQ_OK
: Indicates success.

MQ_ERROR_ACCESS_DENIED
: The queue was not opened with MQ_SEND_ACCESS rights.

MQ_ERROR_BAD_SECURITY_CONTEXT
: The security context buffer (PROPID_M_SECURITY_CONTEXT) is corrupted.

MQ_ERROR_CORRUPTED_INTERNAL_CERTIFICATE
: The internal security certificate provided by MSMQ is corrupted. Register the internal certificate again using the MSMQ Control Panel applet.

MQ_ERROR_CORRUPTED_PERSONAL_CERT_STORE
: The Microsoft® Internet Explorer personal certificate store is corrupted.

MQ_ERROR_CORRUPTED_SECURITY_DATA
: The operating system encountered an error when calling one of the cryptographic functions (CryptoAPI).

MQ_ERROR_COULD_NOT_GET_USER_SID
: **MQSendMessage** could not retrieve the user identifier specified by PROPID_M_SENDERID.

MQ_ERROR_DTC_CONNECT
MSMQ was unable to connect to MS DTC.

MQ_ERROR_ILLEGAL_FORMATNAME
Format name specified in PROPID_M_ADMIN_QUEUE or PROPID_M_RESP_QUEUE is illegal.

MQ_ERROR_INVALID_CERTIFICATE
The external security certificate passed in PROPID_M_SENDER_CERT is not valid. The certificate is corrupted, or not placed in the Microsoft Internet Explorer personal certificate store.

MQ_ERROR_INVALID_HANDLE
The queue handle specified in *hDestinationQueue* is not valid.

MQ_ERROR_MESSAGE_STORAGE_FAILED
A recoverable message (PROPID_M_DELIVERY is set to MQMSG_DELIVERY_RECOVERABLE) could not be stored on the local computer.

MQ_ERROR_NO_INTERNAL_USER_CERT
The internal security certificate provided by MSMQ is not registered. Register the internal certificate using the MSMQ Control Panel.

MQ_ERROR_PROPERTY
One or more properties resulted in an error.

MQ_ERROR_SERVICE_NOT_AVAILABLE
Cannot connect to the Queue Manager.

MQ_ERROR_STALE_HANDLE
The specified queue handle was obtained in a previous session of the Queue Manager service. Close the queue and open it again to obtain a fresh handle.

MQ_ERROR_TRANSACTION_USAGE
Transaction error. Either the message is part of a transaction and its target queue is a non-transaction queue, or the message is not part of a transaction and the target queue is a transaction queue.

MQ_INFORMATION_PROPERTY
One or more of the properties resulted in a warning even though the function completed.

Remarks

All message properties can be attached to a message. Some are attached by MSMQ and others can be attached by the sending application. For a complete list of all the message properties, Sending Messages To a Queue.

The PROPID_M_BODY property is a **CAUI1** structure that contains the message body. The caui1.cElems field of this structure represents the size of the message body.

The sending application can receive two types of messages in response to the messages it sends:

- If PROPID_M_RESP_QUEUE is passed to **MQSendMessage**, the receiving application can send application-defined response messages back to the specified queue. For information on response queues, see Response Queues.
- If PROPID_M_ADMIN_QUEUE and PROPID_M_ACKNOWLEDGE is passed to **MQSendMessage**, MSMQ can return acknowledgment messages back to the administration queue.

You can use the same queue for the response queue (PROPID_M_RESP_QUEUE) and administration queue (PROPID_M_ADMIN_QUEUE).

To save a copy of a message after it is successfully sent, set PROPID_M_JOURNAL to MQMSG_JOURNAL or MQMSG_JOURNAL | MQMSG_DEADLETTER and attach it to the message. This tells MSMQ to save a copy of message in the sending computer's machine journal after the message is successfully sent. For more information, see Journal Queues.

To save a copy of a message if it does not reach its destination, set PROPID_M_JOURNAL to

MQMSG_DEADLETTER, or MQMSG_JOURNAL | MQMSG_DEADLETTER and attach it to the message. This tells MSMQ to save a copy of the message in the dead letter queue of the computer that could not deliver the message. This could be the sending machine, or any MSMQ server used to rout the message to its destination. For information on dead letter queues, see Dead Letter Queues.

When impersonating another user, **MQGetSecurityContext** must be called. Typically, **MQGetSecurityContext** is only needed when sending authenticated messages.

PROPID_M_RESP_QUEUE can be used to send the format name of a private queue to another application. This is typically done when the sending application wants to make a private queue available to other applications.

Sending Messages within a Transactions

If the send is part of a transaction (*pTransaction* is not set to MQ_NO_TRANSACTION or NULL), the *hDestinationQueue* parameter must refer to a transaction queue.

During a transaction, MSMQ performs the following tasks:

- The message's priority property (PROPID_M_PRIORITY) is set to 0.
- The message's delivery property (PROPID_M_DELIVERY) is set to MQMSG_DELIVERY_RECOVERABLE.
- The time-to-be-received and time-to-reach-queue timers are set by the first message sent in the transaction. All subsequent messages use the first message's timer settings.
- As part of a transaction, messages are not sent until the Commit time frame. For information on ITransaction::Commit, see the Platform SDK.

Requesting a Response

The sending application can pass PROPID_M_RESP_QUEUE to **MQSendMessage** to indicate that it expects a response from the receiving application. The messages returned to the response queue specified by this property are application-defined.

Examples

For examples of using **MQSendMessage**, see:

- Sending Messages That Request Acknowledgments
- Sending Messages That Request a Response
- Sending Private Messages
- Sending Messages Using an Internal Transaction
- Sending Messages Using an MS DTC External Transaction

See Also

MQGetSecurityContext, **MQMSGPROPS**, **MQSetQueueSecurity**, PROPID_M_ACKNOWLEDGE, PROPID_M_ADMIN_QUEUE, PROPID_M_BODY, PROPID_M_BODY_SIZE, PROPID_M_CLASS, PROPID_M_DELIVERY, PROPID_M_JOURNAL, PROPID_M_MSGID, PROPID_M_PRIORITY, PROPID_M_RESP_QUEUE, PROPID_M_SENDERID, PROPID_M_SENTTIME, PROPID_M_TIME_TO_BE_RECEIVED, PROPID_M_TIME_TO_REACH_QUEUE

MQSetQueueProperties

The **MQSetQueueProperties** function sets the properties of a specific queue.

```
HRESULT APIENTRY MQSetQueueProperties(
  LPCWSTR lpwcsFormatName,
  MQQUEUEPROPS *pQueueProps
);
```

Parameters

lpwcsFormatName

[in] Pointer to the format name string of the queue whose properties will be set. Use a public or private format name to specify the queue. You cannot specify a queue using a direct format name.

pQueueProps

[in] Pointer to the **MQQUEUEPROPS** structure that specifies the properties to be set.

On input, the **cProps** member of **MQQUEUEPROPS** specifies the number of properties to be set, the **aPropID** array specifies their property identifiers, and the **aPropVar** array indicates the new values of the specified properties.

On output, the optional **aStatus** array, if it was included in **MQQUEUPROPS**, indicates the status of the properties.

Return Values

MQ_OK

Indicates success.

MQ_ERROR_ACCESS_DENIED

The process does not have the sufficient access rights to set the properties of the queue. For a complete list of queue access rights, see Access Control.

To change access rights, call **MQSetQueueSecurity**.

MQ_ERROR_ILLEGAL_FORMATNAME

The *lpwcsFormatName* parameter specified an illegal format name.

MQ_ERROR_ILLEGAL_PROPERTY_VALUE

An illegal property value is specified.

MQ_ERROR_NO_DS

No connection with the Site Controller server. Cannot access the MQIS.

MQ_ERROR_PROPERTY

One or more properties resulted in error.

MQ_ERROR_SERVICE_NOT_AVAILABLE

Cannot connect to the Queue Manager.

MQ_ERROR_UNSUPPORTED_FORMATNAME_OPERATION

The *lpwcsFormatName* parameter specified a direct format name. You cannot set properties for a queue that is being accessed directly.

MQ_ERROR_WRITE_NOT_ALLOWED

Cannot set the properties of the queue in MQIS while an MSMQ information store server is being installed.

MQ_INFORMATION_PROPERTY

One or more of the properties resulted in a warning even though the function completed.

Remarks

If the format name of the queue is unknown, see Format Name to find ways to obtain a new format name.

The following queue properties cannot be set by **MQSetQueueProperties**.

Property Name	Reason
PROPID_Q_BASEPRIORITY	For public queues only. Cannot be set for private queues.
PROPID_Q_CREATE_TIME	Set by MSMQ.
PROPID_Q_INSTANCE	Set by MSMQ.
PROPID_Q_MODIFY_TIME	Set by MSMQ.
PROPID_Q_PATHNAME	Can only be set when the queue is created.
PROPID_Q_TRANSACTION	Can only be set when the queue is created.

For a list of the queue properties you can set, see Setting a Queue's Properties Using API Functions.

When setting the properties for public queues, some clients may see the old settings registered in MQIS. Changes to MQIS (such as setting queue properties) are propagated from site to site, which can cause delays in availability of the most current information. Consequently, clients in some sites may still see old settings, even though they were changed by **MQSetQueueProperties**. Propagation delays, including communication network delays such as down links, are controlled by the MSMQ Administrator.

A public queue's properties cannot be set by an independent client computer running offline. For information on offline operations, see MSMQ Offline Support.

See Also

MQGetQueueProperties, **MQSetQueueSecurity**, PROPID_Q_BASEPRIORITY, PROPID_Q_CREATE_TIME, PROPID_Q_INSTANCE, PROPID_Q_MODIFY_TIME, PROPID_Q_PATHNAME, PROPID_Q_TRANSACTION

MQSetQueueSecurity

The **MQSetQueueSecurity** function sets the access control security for the queue.

```
HRESULT APIENTRY MQSetQueueSecurity(
  LPCWSTR lpwcsFormatName,
  SECURITY_INFORMATION *SecurityInformation,
  PSECURITY_DESCRIPTOR *pSecurityDescriptor
);
```

Parameters

lpwcsFormatName

[in] Pointer to the format name string of the queue to be secured. You cannot specify a direct format name.

securityInformation

[in] Specifies a **SECURITY_INFORMATION** structure identifying the contents of the security descriptor pointed to by the *pSecurityDescriptor* parameter. (For information on the **SECURITY_INFORMATION** structure, see the Microsoft Platform SDK.)

pSecurityDescriptor

[in] Pointer to a **SECURITY_DESCRIPTOR** structure. Can be set to NULL; see default values in the following Return Values section. (For information on the **SECURITY_DESCRIPTOR** structure, see the Microsoft Platform SDK.)

Return Values

MQ_OK

Indicates success.

MQ_ERROR_ACCESS_DENIED

The process owner does not have the sufficient access rights to set the queue security information. The following access rights may be required:

- MQSEC_CHANGE_QUEUE_PERMISSIONS
- MQSEC_TAKE_QUEUE_OWNERSHIP

If access is denied, contact someone who has rights to modify the security descriptor.

MQ_ERROR_FUNCTION_NOT_SUPPORTED

MQSetQueueSecurity is not supported in Windows 95.

MQ_ERROR_ILLEGAL_FORMATNAME

The *lpwcsFormatName* parameter specified an illegal format name.

MQ_ERROR_NO_DS

No connection with the Site Controller server. Cannot access the MQIS.

MQ_ERROR_PRIVILEGE_NOT_HELD

The process owner does not have the proper privilege to set the queue's system access control list.

MQ_ERROR_SERVICE_NOT_AVAILABLE

Cannot connect to the Queue Manager.

MQ_ERROR_UNSUPPORTED_FORMATNAME_OPERATION

The *lpwcsFormatName* parameter contains a public or private queue using a direct format name or a journal, dead letter, or connector queue.

Remarks

Following are the default values for the security descriptor.

Default Value	Meaning
Owner	The process user.

Group	The process group.
DACL	Full control for the process user. All processes of other users can get queue properties, get queue security, and send messages to the queue.
SACL	None.

The format name of the queue (specified by *lpwcsFormatName*) must be a public or private format name. MSMQ must be able to access MQIS (for public queues) or the local computer (for private queues) to get the queue's security descriptor.

If the format name of the queue is unknown, see Format Name to find ways to obtain a new format name.

The following access rights and privileges are required to change the queue's security descriptor.

Access Right/Privilege	**Required To**
MQSEC_TAKE_QUEUE_OWNERSHIP	Change the owner of the queue. This access right is equivalent to WRITE_OWNER as defined by the Win32 header files.
SE_TAKE_OWNERSHIP_NAME	Change the owner of the queue. This privilege can be used instead of having the MQSEC_TAKE_QUEUE_OWNERSHIP access on the queue. If a user has this privilege on the server, the user can change the owner of any public queue in the enterprise. If the user has this privilege on the local computer, the user can change the owner of any private queue.
MQSEC_CHANGE_QUEUE_PERMISSIONS	Change the queue's discretionary access control list (DACL) if the process is not the owner of the queue. This access right is equivalent to WRITE_DAC as defined by the Win32 header files.
SE_SECURITY_NAME	Change the queue's system ACL (SACL); this privilege must be enabled for the calling process on MQIS for public queues and on the local computer for private queues.

MQGetQueueSecurity cannot retrieve the security descriptor of a journal, dead letter, connector or foreign queues.

A public queue's security descriptor cannot be set by an independent client computer running offline. For information on offline operations, see MSMQ Offline Support.

See Also

MQGetQueueSecurity

MSMQ Mail Functions

The following topics describe the MSMQ mail functions. These functions allow you to parse and compose messages, and free the memory allocated when using these functions.

MSMQ Mail Functions include:

- **MQMailComposeBody**
- **MQMailFreeMemory**
- **MQMailParseBody**

MQMailComposeBody

The **MQMailComposeBody** function composes a message body from information provided by an e-mail data structure. The message body is formatted in MSMQ mail format.

```
STDAPI MQMailComposeBody(
  LPMQMailEMail pEMail,
  ULONG FAR *pcbBuffer,
  LPBYTE FAR  ppbBuffer
);
```

Parameters

pEMail
[in] Pointer to an **MQMailEMail** structure that contains mail information.

pcbBuffer
[out] Pointer to where the size of the newly allocated message body buffer is copied to.

ppbBuffer
[out] Pointer to where the address of the newly allocated message body buffer is copied to.

Return Values

S_OK
Indicates success.

E_INVALIDARG
Invalid arguments.

Other system errors
Use the FAILED(hr) macro to test for errors.

Remarks

The **MQMailComposeBody** function is used to create the body of an MSMQ mail message. It allocates a message body buffer and fills it with an MSMQ mail format representation of the data specified by the *pEMail* parameter.

Mail messages can be sent to MAPI clients via the MSMQ MAPI Transport Provider, Microsoft® Exchange users via the MSMQ Exchange Connector, or other MSMQ applications.

When this function succeeds, the application must free the memory allocated for the message body buffer by calling **MQMailFreeMemory** after the buffer is used.

The following table defines which members of the **MQMailEMail** structure are required to compose a message body for each type of e-mail. Required properties are marked with an X (default values can be used). When composing an e-mail object, all required members must have valid values, otherwise an error condition is raised by **MQMailComposeBody**.

MQMailEMail	Text Message	Form	Tnef	Delivery Report	Non-Delivery Report
pForm	X	X			
szSubject	X (Empty string)	(Empty string)			
fRequestDeliveryReport	X (False)	X (False)			
fRequestNonDeliveryReport	X (False)	X (False)			
pftDate	X (Current time)	X (Current time)		X (Current time)	X (Current time)

pRecips	X	X	X	X	X
iType	X	X	X	X	X
form		X			
message	X				
tnef			X		
DeliveryReport				X	
NonDeliveryReport					X

See Also

MQMailEMail, **MQMailFreeMemory**

MQMailFreeMemory

The **MQMailFreeMemory** function frees memory allocated by **MQMailComposeBody** and **MQMailParseBody**.

```
STDAPI_(void) MQMailFreeMemory(
  LPVOID lpBuffer
);
```

Parameters

lpBuffer
[in] Pointer to allocated memory.

Return Values

None.

Remarks

It is the application's responsibility to free all memory allocated by **MQMailComposeBody** and **MQMailParseBody**.

See Also

MQMailComposeBody, **MQMailParseBody**

MQMailParseBody

The **MQMailParseBody** function parses the body of an MSMQ mail message and places the information in a mail data structure.

```
STDAPI MQMailParseBody(
  ULONG cbBuffer,
  LPBYTE FAR *pbBuffer,
  LPMQMailEMail FAR *ppEMail
);
```

Parameters

cbBuffer
[in] Buffer size.

pbBuffer
[in] Pointer to the buffer containing the body of the MSMQ mail message.

ppEMail
[out] Pointer to where the address of the newly allocated **MQMailEMail** structure that contains the copied mail information.

Return Values

S_OK
Indicates success.

E_INVALIDARG
Invalid arguments.

Other system errors
Use the FAILED(hr) macro to test for errors.

Remarks

The **MQMailParseBody** function is used to convert the body of an MSMQ mail message so its information can be used by an MSMQ application.

The application must free the memory allocated for the **MQMailEMail** object once the parsed information is used. To free the memory call **MQMailFreeMemory**.

MSMQ mail messages can be received from another MSMQ application or either of the two MSMQ Mail services: MSMQ MAPI Transport Provider or the MSMQ Exchange Connector.

See Also

MQMailEMail, **MQMailFreeMemory**

MSMQ Error and Information Codes

The following error and information codes are defined in mq.h.

Code	Description
MQ_ERROR	Generic error code.
MQ_ERROR_ACCESS_DENIED	Access to the specified queue or computer is denied. Verify the access rights for the operation (for example, creating, setting properties, or deleting a queue). To change access rights for a queue, call **MQSetQueueSecurity**.
MQ_ERROR_BAD_DSSERVER_REGISTRY	Registry parameter for MQIS server has a bad value.
MQ_ERROR_BAD_SECURITY_CONTEXT	Security context specified by PROPID_M_SECURITY_CONTEXT is corrupted.
MQ_ERROR_BUFFER_OVERFLOW	Supplied message body buffer is too small. A partial copy of the message body is copied to the buffer, but the message is not removed from the queue.
MQ_ERROR_CANNOT_IMPERSONATE_CLIENT	MSMQ information store server cannot impersonate the client application. Security credentials could not be verified.
MQ_ERROR_COMPUTER_DOES_NOT_SUPPORT_ENCRYPTION	Encryption failed. Computer (source or destination) does not support encryption operations.
MQ_ERROR_CORRUPTED_INTERNAL_CERTIFICATE	MSMQ-supplied internal certificate is corrupted.
MQ_ERROR_CORRUPTED_PERSONAL_CERT_STORE	Microsoft® Internet Explorer personal certificate store is corrupted.
MQ_ERROR_CORRUPTED_SECURITY_DATA	Cryptographic function (CryptoAPI) has failed.
MQ_ERROR_COULD_NOT_GET_ACCOUNT_INFO	MSMQ could not get account information for the user.
MQ_ERROR_COULD_NOT_GET_USER_SID	MSMQ could not get the specified sender identifier.
MQ_ERROR_DELETE_CN_IN_USE	Specified connected network (CN) cannot be deleted because it is defined in at least one computer. Remove the CN from all CN lists and try again.
MQ_ERROR_DS_ERROR	Internal error with MQIS.
MQ_ERROR_DS_IS_FULL	MSMQ information store is full.
MQ_ERROR_DTC_CONNECT	MSMQ cannot connect to the

	Microsoft® Distributed Transaction Coordinator (MS DTC).
MQ_ERROR_FORMATNAME_BUFFER_TOO_SMALL	Specified format name buffer is too small to contain the queue's format name.
MQ_ERROR_FUNCTION_NOT_SUPPORTED	Function is not supported in Windows 95.
MQ_ERROR_ILLEGAL_CONTEXT	The *lpwcsContext* parameter of **MQLocateBegin** is not NULL.
MQ_ERROR_ILLEGAL_CURSOR_OPERATION	An attempt was made to peek at the next message in the queue when cursor was at the end of the queue.
MQ_ERROR_ILLEGAL_FORMATNAME	Format name specified is not valid.
MQ_ERROR_ILLEGAL_MQCOLUMNS	Indicates that *pColumns* is NULL.
MQ_ERROR_ILLEGAL_MQQMPROPS	No properties are specified by the **MQQMPROPS** structure, or it is set to NULL.
MQ_ERROR_ILLEGAL_MQQUEUEPROPS	No properties are specified by the **MQQUEUEPROPS** structure, or it is set to NULL.
MQ_ERROR_ILLEGAL_OPERATION	The operation is not supported on this specific platform.
MQ_ERROR_ILLEGAL_PROPERTY_SIZE	The specified buffer for the message identifier or correlation identifier is not the correct size.
MQ_ERROR_ILLEGAL_PROPERTY_VALUE	Property value specified in the **PROPVARIANT** array is illegal.
MQ_ERROR_ILLEGAL_PROPERTY_VT	**VARTYPE** specified in the **VT** field of the **PROPVARIANT** array is not valid.
MQ_ERROR_ILLEGAL_PROPID	Property identifier in the property identifier array is not valid.
MQ_ERROR_ILLEGAL_QUEUE_PATHNAME	MSMQ pathname specified for the queue is not valid.
MQ_ERROR_ILLEGAL_RELATION	Relationship parameter is not valid.
MQ_ERROR_ILLEGAL_RESTRICTION_PROPID	Property identifier specified in **MQRESTRICTION** is invalid.
MQ_ERROR_ILLEGAL_SECURITY_DESCRIPTOR	Specified security descriptor is not valid.
MQ_ERROR_ILLEGAL_SORT	Illegal sort specified.
MQ_ERROR_ILLEGAL_SORT_PROPID	Property identifier specified in **MQSORTSET** is not valid.
MQ_ERROR_ILLEGAL_USER	User is not legal.
MQ_ERROR_INCONSISTENT_QM_ID	Inconsistency exists between the Queue Manager identifier found in MQIS and the one found in the registry.
MQ_ERROR_INSUFFICIENT	Not all properties required for the

_PROPERTIES	operation were specified.
MQ_ERROR_INSUFFICIENT _RESOURCES	Insufficient resources to complete operation (for example, not enough memory). Operation failed.
MQ_ERROR_INVALID _CERTIFICATE	Security certificate specified by PROPID_M_SENDER_CERT is invalid, or the certificate is not correctly placed in the Microsoft® Internet Explorer personal certificate store.
MQ_ERROR_INVALID_HANDLE	Specified queue handle is not valid.
MQ_ERROR_INVALID_OWNER	Object owner is not valid. Owner was not found when trying to create object.
MQ_ERROR_INVALID_PARAMETER	One of the IN parameters supplied by the operation is not valid.
MQ_ERROR_IO_TIMEOUT	**MQReceiveMessage** I/O timeout has expired.
MQ_ERROR_LABEL_TOO_LONG	Message label is too long. It should be equal to or less than MQ_MAX_MSG_LABEL_LEN.
MQ_ERROR_LABEL_BUFFER _TOO_SMALL	Message label buffer is too small for received label.
MQ_ERROR_LOG_XACT_STATE	Cannot log transaction state.
MQ_ERROR_LOGMGR_LOAD	Cannot get log manager library or interface.
MQ_ERROR_MACHINE_EXISTS	Machine with the specified name already exists.
MQ_ERROR_MACHINE_NOT _FOUND	Specified machine could not be found in MQIS.
MQ_ERROR_MESSAGE _ALREADY_RECEIVED	Message pointed at by the cursor has already been removed from the queue.
MQ_ERROR_MESSAGE _STORAGE_FAILED	Recoverable message could not be stored on the local computer.
MQ_ERROR_MISSING _CONNECTOR_TYPE	Specified a property typically generated by MSMQ but did not specify PROPID_M_CONNECTOR_TYPE
MQ_ERROR_MQIS_READONLY _MODE	MQIS database is in read-only mode.
MQ_ERROR_MQIS_SERVER _EMPTY	The list of MSMQ information store servers (in registry) is empty.
MQ_ERROR_NO_DS	No connection with the Site Controller server. Cannot access the MQIS.
MQ_ERROR_NO_INTERNAL _USER_CERT	No internal certificate available for this user.
MQ_ERROR_NO_RESPONSE _FROM_OBJECT_SERVER	No response from MQIS server. Operation status is unknown.

MQ_ERROR_NO_TRANSPORT	No network transport to the remote computer.
MQ_ERROR_OBJECT_SERVER _NOT_AVAILABLE	Object's MSMQ information store server is not available. Operation failed.
MQ_ERROR_OPERATION _CANCELLED	Operation was canceled before it could be started.
MQ_ERROR_PRIVILEGE _NOT_HELD	Application does not have the required privileges to perform the operation.
MQ_ERROR_PROPERTY	One or more of the specified properties caused an error.
MQ_ERROR_PROPERTY _NOTALLOWED	Specified property is not valid for the operation (for example, specifying PROPID_Q_INSTANCE when setting queue properties).
MQ_ERROR_PROV_NAME _BUFFER_TOO_SMALL	The provider name buffer for cryptographic service provider is too small.
MQ_ERROR_QUEUE_DELETED	Queue was deleted before the message could be read. The specified queue handle is no longer valid and the queue must be closed.
MQ_ERROR_QUEUE_EXCEEDS _QUOTA	Specified queue is full.
MQ_ERROR_QUEUE_EXISTS	Queue (public or private) with the identical MSMQ pathname is registered. Public queues are registered in MQIS. Private queues are registered in the local computer.
MQ_ERROR_QUEUE _NOT_AVAILABLE	Error while reading from queue residing on a remote computer.
MQ_ERROR_QUEUE_NOT_FOUND	Public queue is not registered in MQIS. This error does not apply to private queues.
MQ_ERROR_RECOVER _TRANSACTIONS	MSMQ could not recover the transactions.
MQ_ERROR_REGISTRATION _ERROR	Error in registration access.
MQ_ERROR_RESULT _BUFFER_TOO_SMALL	Supplied result buffer is too small. **MQLocateNext** could not return at least one complete query result.
MQ_ERROR_SECURITY _DESCRIPTOR_TOO_SMALL	Supplied security buffer is too small.
MQ_ERROR_SENDER_CERT _BUFFER_TOO_SMALL	Supplied sender certificate buffer is too small.
MQ_ERROR_SENDERID _BUFFER_TOO_SMALL	Supplied sender identification buffer is too small to hold sender

	identification.
MQ_ERROR_SERVICE_NOT _AVAILABLE	Application was unable to connect to the Queue Manager.
MQ_ERROR_SIGNATURE _BUFFER_TOO_SMALL	The signature buffer is too small.
MQ_ERROR_SHARING_VIOLATION	Sharing violation when opening queue. The application is trying to open an already opened queue that has exclusive read rights.
MQ_ERROR_STALE_HANDLE	Specified handle was obtained in a previous session of the Queue Manager service.
MQ_ERROR_SYMM_KEY _BUFFER_TOO_SMALL	The symmetric key buffer is too small.
MQ_ERROR_TOO_MANY _TRANSACTIONS	Too many active transactions.
MQ_ERROR_TRANSACTION _ENLIST	Cannot enlist transaction.
MQ_ERROR_TRANSACTION_FILE	MSMQ cannot open or create a transaction file.
MQ_ERROR_TRANSACTION _IMPORT	MSMQ could not import the specified transaction.
MQ_ERROR_TRANSACTION _PREPAREINFO	MSMQ could not get the Prepare Info from the Microsoft Distributed Transaction Coordinator (MS DTC).
MQ_ERROR_TRANSACTION _QUEUE	The application could not create or use the transaction queue.
MQ_ERROR_TRANSACTION _SEQUENCE	Transaction operation sequence is incorrect.
MQ_ERROR_TRANSACTION _USAGE	Either the queue or the message is not transactional. Transaction messages can only be sent to a transaction queue, and transaction queues can only receive transaction messages.
MQ_ERROR_WRITE_NOT _ALLOWED	Write operations to MQIS are not allowed while an MSMQ information store server is being installed.
MQ_ERROR_UNSUPPORTED _ACCESS_MODE	Specified access mode is not supported. Supported access modes include MQ_PEEK_MESSAGE, MQ_SEND_MESSAGE, and MQ_RECEIVE_MESSAGE.
MQ_ERROR_UNSSUPORTED _DBMS	Current version of Database Management System is not supported
MQ_ERROR_UNSUPPORTED _FORMATNAME_OPERATION	Requested operation is not supported for the specified format name (for example, trying to open a queue to receive messages using a

	direct format name).
MQ_ERROR_UNSUPPORTED_PROTOCOL	Remote machine network protocol is not supported.
MQ_ERROR_USER_BUFFER_TOO_SMALL	Supplied buffer for user is too small to hold the returned information.
MQ_INFORMATION_DUPLICATE_PROPERTY	Property already specified with same value. When duplicate settings are found, the first entry is used and subsequent settings are ignored.
MQ_INFORMATION_FORMATNAME_BUFFER_TOO_SMALL	Supplied format name buffer is too small. Queue was still created.
MQ_INFORMATION_ILLEGAL_PROPERTY	Specified identifier in property identifier array **aPropID** is not valid.
MQ_INFORMATION_OPERATION_OPERATION_PENDING	Asynchronous operation is pending.
MQ_INFORMATION_PROPERTY	One or more of the specified properties resulted in a warning. Operation completed anyway.
MQ_INFORMATION_PROPERTY_IGNORED	Specified property is not valid for this operation (for example, PROPID_M_SENDERID is not valid; it is set by MSMQ when sending messages).
MQ_INFORMATION_UNSUPPORTED_PROPERTY	Specified property is not supported by this operation. This property is ignored.

MSMQ Properties

MSMQ uses message, queue, and machine properties to specify the characteristics of messages, queues, and machines. A property can be an IN property, an OUT property, or both, depending on how it is used. For a description of IN and OUT properties, see MSMQ Object Properties.

MSMQ properties include:

Queue Properties

- PROPID_Q_AUTHENTICATE
- PROPID_Q_BASEPRIORITY
- PROPID_Q_CREATE_TIME
- PROPID_Q_INSTANCE
- PROPID_Q_JOURNAL
- PROPID_Q_JOURNAL_QUOTA
- PROPID_Q_LABEL
- PROPID_Q_MODIFY_TIME
- PROPID_Q_PATHNAME
- PROPID_Q_PRIV_LEVEL
- PROPID_Q_QUOTA
- PROPID_Q_TRANSACTION
- PROPID_Q_TYPE

Message properties

- PROPID_M_ACKNOWLEDGE
- PROPID_M_ADMIN_QUEUE
- PROPID_M_ADMIN_QUEUE_LEN
- PROPID_M_APPSPECIFIC
- PROPID_M_ARRIVEDTIME
- PROPID_M_AUTH_LEVEL
- PROPID_M_AUTHENTICATED
- PROPID_M_BODY
- PROPID_M_BODY_SIZE
- PROPID_M_BODY_TYPE
- PROPID_M_CLASS
- PROPID_M_CONNECTOR_TYPE
- PROPID_M_CORRELATIONID
- PROPID_M_DELIVERY
- PROPID_M_DEST_QUEUE
- PROPID_M_DEST_QUEUE_LEN
- PROPID_M_DEST_SYMM_KEY
- PROPID_M_DEST_SYMM_KEY_LEN
- PROPID_M_ENCRYPTION_ALG
- PROPID_M_EXTENSION
- PROPID_M_EXTENSION_LEN
- PROPID_M_HASH_ALG

- PROPID_M_JOURNAL
- PROPID_M_LABEL
- PROPID_M_LABEL_LEN
- PROPID_M_MSGID
- PROPID_M_PRIORITY
- PROPID_M_PRIV_LEVEL
- PROPID_M_PROV_NAME
- PROPID_M_PROV_NAME_LEN
- PROPID_M_PROV_TYPE
- PROPID_M_RESP_QUEUE
- PROPID_M_RESP_QUEUE_LEN
- PROPID_M_SECURITY_CONTEXT
- PROPID_M_SENDER_CERT
- PROPID_M_SENDER_CERT_LEN
- PROPID_M_SENDERID
- PROPID_M_SENDERID_LEN
- PROPID_M_SENDERID_TYPE
- PROPID_M_SENTTIME
- PROPID_M_SIGNATURE
- PROPID_M_SIGNATURE_LEN
- PROPID_M_SRC_MACHINE_ID
- PROPID_M_TIME_TO_BE_RECEIVED
- PROPID_M_TIME_TO_REACH_QUEUE
- PROPID_M_TRACE
- PROPID_M_VERSION
- PROPID_M_XACT_STATUS_QUEUE
- PROPID_M_XACT_STATUS_QUEUE_LEN

Queue Manager Properties

- PROPID_QM_CONNECTION
- PROPID_QM_ENCRYPTION_PK
- PROPID_QM_MACHINE_ID
- PROPID_QM_PATHNAME
- PROPID_QM_SITE_ID

PROPID_M_ACKNOWLEDGE

The PROPID_M_ACKNOWLEDGE property specifies the type of acknowledgment messages that MSMQ posts (in the administration queue) when the message is sent.

Type Indicator
VT_UI1 (or VT_NULL)

PROPVARIANT Field
bVal

Property Values
This property can be set to one (or more by ORing them together) of the following values:

MQMSG_ACKNOWLEDGMENT_FULL_REACH_QUEUE
Posts positive and negative acknowledgments depending on whether or not the message reaches the queue. This can happen when the *time-to-reach-queue* timer expires, or a message cannot be authenticated.

MQMSG_ACKNOWLEDGMENT_FULL_RECEIVE
Posts a positive or negative acknowledgment depending on whether or not the message is retrieved from the queue before *its time-to-be-received* timer expires.

MQMSG_ACKNOWLEDGMENT_NACK_REACH_QUEUE
Posts a negative acknowledgment when the message cannot reach the queue. This can happen when the *time-to-reach-queue* timer expires, or a message cannot be authenticated.

MQMSG_ACKNOWLEDGMENT_NACK_RECEIVE
Posts a negative acknowledgment when an error occurs and the message cannot be retrieved from the queue before its *time-to-be-received* timer expires.

MQMSG_ACKNOWLEDGMENT_NONE
The default. No acknowledgment messages (positive or negative) are posted.

Remarks

Positive and negative acknowledgments are typically MSMQ-generated messages that are sent to an administration queue specified by the message (acknowledgment messages can also be created by MSMQ connector applications when sending messages to a foreign queue). When asking for acknowledgments, you must also specify the administration queue when you send the message (see PROPID_M_ADMIN_QUEUE). For information on administration queues, see Administration Queues.

Acknowledgment messages contain some of the information found in the original message, however, each acknowledgment message has its own message identifier and class. The message identifier, PROPID_M_MSGID, identifies the acknowledgment in the same way it identifies each message sent by an MSMQ application. The message class, PROPID_M_CLASS, identifies the type of acknowledgment that was posted. Both these properties are set by MSMQ when it creates the acknowledgment message.

To request acknowledgment messages, pass PROPID_M_ACKNOWLEDGE and PROPID_M_ADMIN_QUEUE to **MQSendMessage**.

To find out if an acknowledgment message was requested for a message in a queue, pass PROPID_M_ACKNOWLEDGE to **MQReceiveMessage** and examine its returned value. When passing PROPID_M_ACKNOWLEDGE to MQReceiveMessage, the corresponding VT field in the **aPropVar** array can be set to VT_NULL.

For information on the *time-to-reach-queue* and *time-to-be-received* timers, see Message Timers. To set the *time-to-reach-queue* and *time-to-be-received* timers, set PROPID_M_TIME_TO_REACH_QUEUE and PROPID_M_TIME_TO_BE_RECEIVED, respectively.

To see how PROPID_M_ACKNOWLEDGE is used when sending messages, see Sending Messages that Request Acknowledgments.

Example

The following example sets PROPID_M_ACKNOWLEDGE and PROPID_M_ADMIN_QUEUE as part of preparing MQMSGPROPS.

```
MQMSGPROPS MsgProps;
PROPVARIANT aVariant[10];
MSGPROPID aPropId[10];
DWORD PropIdCount = 0;

HRESULT hr;

QUEUEHANDLE hQueue;

//
// Set PROPID_M_ACKNOWLEDGE.
//
aPropId[PropIdCount] = PROPID_M_ACKNOWLEDGE;                //PropId
aVariant[PropIdCount].vt = VT_UI1;                          //Type
aVariant[PropIdCount].bVal = MQMSG_ACKNOWLEDGMENT_FULL_RECEIVE; //Value

PropIdCount++;

//
// Set the PROPID_M_ADMIN_QUEUE property.
//
aPropId[PropIdCount] = PROPID_M_ADMIN_QUEUE;            //PropId
aVariant[PropIdCount].vt = VT_LPWSTR;                   //Type
aVariant[PropIdCount].pwszVal = szwAdminFormatName;   //An already obtained format
name of the admin queue.

PropIdCount++;

//
// Set other message properties such as PROPID_M_BODY, PROPID_M_LABEL.
//



//
// Set the MQMSGPROPS structure
//
MsgProps.cProp = PropIdCount;          //Number of properties.
MsgProps.aPropID = aPropId;            //Ids of properties.
MsgProps.aPropVar = aVariant;          //Values of properties.
MsgProps.aStatus   = NULL;              //No Error report.




//
// Send message.
//
hr = MQSendMessage(
       hQueue,                     // handle to the Queue.
       &MsgProps,                   // Message properties to be sent.
```

```
        MQ_NO_TRANSACTION            // No transaction
        );

if (FAILED(hr))
    {
     //
     // Handle error condition
     //
     }
```

See Also

PROPID_M_ADMIN_QUEUE, PROPID_M_CLASS, PROPID_M_MSGID, PROPID_M_TIME_TO_BE_RECEIVED, PROPID_M_TIME_TO_REACH_QUEUE

PROPID_M_ADMIN_QUEUE

The PROPID_M_ADMIN_QUEUE property specifies the queue used for MSMQ-generated acknowledgment messages.

Type Indicator
VT_LPWSTR

PROPVARIANT Field
pwszVal

Property Values
Format name of administration queue.

Remarks

Acknowledgment messages are sent to the administration queue by MSMQ. For more information, see Administration Queues.

To request acknowledgment messages, pass PROPID_M_ACKNOWLEDGE and PROPID_M_ADMIN_QUEUE to **MQSendMessage**.

To find out which queue is being used as the administration queue, pass PROPID_M_ADMIN_QUEUE and PROPID_M_ADMIN_QUEUE_LEN to **MQReceiveMessage** and examine the returned values.

To see how PROPID_M_ADMIN_QUEUE is used when sending messages, see Sending Messages that Request Acknowledgments.

Example

The following example sets PROPID_M_ACKNOWLEDGE and PROPID_M_ADMIN_QUEUE as part of preparing MQMSGPROPS.

```
MQMSGPROPS MsgProps;
PROPVARIANT aVariant[10];
MSGPROPID aPropId[10];
DWORD PropIdCount = 0;

HRESULT hr;

QUEUEHANDLE hQueue;

//
// Set PROPID_M_ACKNOWLEDGE.
//
aPropId[PropIdCount] = PROPID_M_ACKNOWLEDGE;                //PropId
aVariant[PropIdCount].vt = VT_UI1;                          //Type
aVariant[PropIdCount].bVal = MQMSG_ACKNOWLEDGMENT_FULL_RECEIVE; //Value

PropIdCount++;

//
// Set the PROPID_M_ADMIN_QUEUE property.
//
aPropId[PropIdCount] = PROPID_M_ADMIN_QUEUE;            //PropId
aVariant[PropIdCount].vt = VT_LPWSTR;                   //Type
aVariant[PropIdCount].pwszVal = szwAdminFormatName;   //An already obtained format
name of the admin queue.

PropIdCount++;
```

```
//
// Set other message properties such as PROPID_M_BODY, PROPID_M_LABEL.
//



//
// Set the MQMSGPROPS structure
//
MsgProps.cProp = PropIdCount;          //Number of properties.
MsgProps.aPropID = aPropId;            //Ids of properties.
MsgProps.aPropVar = aVariant;          //Values of properties.
MsgProps.aStatus   = NULL;             //No Error report.



//
// Send message.
//
hr = MQSendMessage(
       hQueue,                      // handle to the Queue.
       &MsgProps,                   // Message properties to be sent.
       MQ_NO_TRANSACTION            // No transaction
       );

if (FAILED(hr))
    {
     //
     // Handle error condition
     //
     }
```

See Also

MQReceiveMessage, **MQSendMessage**, PROPID_M_ACKNOWLEDGE, PROPID_M_ADMIN_QUEUE_LEN

PROPID_M_ADMIN_QUEUE_LEN

The PROPID_M_ADMIN_QUEUE_LEN property indicates the length (in Unicode characters) of the administration queue's format name buffer.

Type Indicator
VT_UI4

PROPVARIANT Field
ulVal

Property Values
Length (in Unicode characters) of the administration queue's format name buffer.

Remarks

The PROPID_M_ADMIN_QUEUE_LEN property is only used by the receiving application when looking for the administration queue used by the sending application

To find out which queue is being used as the administration queue, pass PROPID_M_ADMIN_QUEUE and PROPID_M_ADMIN_QUEUE_LEN to **MQReceiveMessage** and examine the returned values.

On input, PROPID_M_ADMIN_QUEUE_LEN specifies the length of the format name buffer (in Unicode characters) allocated by the receiving application. The buffer should be large enough to hold the format name string including the null-terminating character.

On return, this property holds the length (in Unicode characters) of the PROPID_M_ADMIN_QUEUE format name string including the null-terminating character.

MQReceiveMessage fails if the buffer is not large enough to hold the format name, and PROPID_M_ADMIN_QUEUE_LEN is set to the required buffer length of the format name string.

To see how PROPID_M_ADMIN_QUEUE_LEN is used when reading an acknowledgment message, see Sending Messages that Request Acknowledgments.

Example

The following example allocates a buffer of size 60 for the format name of the administration queue, then sets the PROPID_M_ADMIN_QUEUE and PROPID_M_ADMIN_QUEUE_LEN properties.

```
MQMSGPROPS MsgProps;
PROPVARIANT aVariant[10];
MSGPROPID aPropId[10];
DWORD PropIdCount = 0;

HRESULT hr;
QUEUEHANDLE hQueue;

//
// Prepare the PROPVARIANT array.
//

#define ADMIN_QUEUE_BUFF_LEN 60
WCHAR szwAdminQueueFormatName[ADMIN_QUEUE_BUFF_LEN];

//
// Set the PROPID_M_ADMIN_QUEUE property.
//
aPropId[PropIdCount] = PROPID_M_ADMIN_QUEUE;                    //PropId
aVariant[PropIdCount].vt = VT_LPWSTR;                           //Type
aVariant[PropIdCount].pwszVal = szwAdminQueueFormatName;    //allocated buffer
```

```
PropIdCount++;

//
// Set the PROPID_M_ADMIN_QUEUE_LEN property.
//
aPropId[PropIdCount] = PROPID_M_ADMIN_QUEUE_LEN;          //PropId
aVariant[PropIdCount].vt = VT_UI4;                        //Type
aVariant[PropIdCount].ulVal = ADMIN_QUEUE_BUFF_LEN;       //Value
PropIdCount++;

//
// Set the MQMSGPROPS structure.
//
MsgProps.cProp = PropIdCount;          //Number of properties.
MsgProps.aPropID = aPropId;            //Id of properties.
MsgProps.aPropVar = aVariant;          //Value of properties.
MsgProps.aStatus   = NULL;             //No Error report.
```

See Also

MQReceiveMessage, PROPID_M_ACKNOWLEDGE, PROPID_M_ADMIN_QUEUE

PROPID_M_APPSPECIFIC

The PROPID_M_APPSPECIFIC property specifies application-generated information such as single integer values or application defined message classes.

Type Indicator
VT_UI4 (or VT_NULL)

PROPVARIANT Field
ulVal

Property Values
Unsigned integer. The default is 0.

Remarks

When passing PROPID_M_APPSPECIFIC to **MQReceiveMessage**, the corresponding VT field in the **aPropVar** array can be set to VT_NULL.

In addition to PROPID_M_APPSPECIFIC, you can use the PROPID_M_CORRELATIONID message property for sorting.

Example

This example shows how PROPID_M_APPSPECIFIC is specified in the MQMSGPROPS structure:

```
MsgProps.aPropID[i] = PROPID_M_APPSPECIFIC;        //PropId
MsgProps.aPropVar[i].vt = VT_UI4;                  //Type
MsgProps.aPropVar[i].ulVal = 444;                  //Value
```

For an example of using PROID_M_APPSPECIFIC, see Reading Messages Using a Cursor.

See Also

MQReceiveMessage, **MQSendMessage**, PROPID_M_CORRELATIONID

PROPID_M_ARRIVEDTIME

The PROPID_M_ARRIVEDTIME property indicates when the message arrived at the queue.

Type Indicator
VT_UI4 (or VT_NULL)

PROPVARIANT Field
ulVal

Property Values
Unsigned integer.

Remarks

The PROPID_M_ARRIVEDTIME property is attached to the message by MSMQ. The arrival time returned is the number of seconds elapsed since midnight (00:00:00), January 1, 1970 (Coordinated Universal time) according to the system clock.

To see when a message arrived, pass PROPID_M_ARRIVEDTIME to **MQReceiveMessage** and look at the returned results. When passing PROPID_M_ARRIVEDTIME to **MQReceiveMessage**, the corresponding VT field in the **aPropVar** array can be set to VT_NULL.

When reading messages from a journal queue, PROPID_M_ARRIVETIME indicates when the original message reached its queue, not when the original message was removed from the queue and a copy placed in the journal queue.

When reading messages from a machine journal, dead letter queue, or transactional dead letter queue, PROPID_M_ARRIVETIME indicates when the message reached the system queue where the application is reading the message.

For information on how to find out when the message was sent, see PROPID_M_SENTTIME.

See Also

MQReceiveMessage, PROPID_M_SENTTIME

PROPID_M_AUTH_LEVEL

The PROPID_M_AUTH_LEVEL property specifies if the message needs to be authenticated.

Type Indicator
VT_UI4

PROPVARIANT Field
ulVal

Property Values
This property can be set to one of the following values:

MQMSG_AUTH_LEVEL_NONE
The default. The message is not signed. MSMQ does not need to authenticate the message when it reaches the queue.

MQMSG_AUTH_LEVEL_ALWAYS
MSMQ must sign the message when it is sent and authenticate the message when it reaches the queue.

Remarks

The PROPID_M_AUTH_LEVEL property is only used by the sending application.

To authenticate a message, MSMQ digitally signs the message when it is sent, then uses the digital signature to authenticate the message when it is received. For information on how MSMQ creates the digital signature and how it authenticates the received message, see Message Authentication.

If PROPID_M_AUTH_LEVEL is set to MQMSG_AUTH_LEVEL_NONE and the target queue is set to force authentication, the message will be rejected when it reaches the queue.

PROPID_M_AUTHENTICATED

The PROPID_M_AUTHENTICATED property indicates if MSMQ authenticated the message when it was received by the target queue.

Type Indicator
VT_UI1 (or VT_NULL)

PROPVARIANT Field
bVal

Property Values
This property can be set to one of the following values:

0
MSMQ did not authenticate the message when it was received.

1
MSMQ authenticated the message when it was received.

Remarks

This property is only used by the receiving application when calling **MQReceiveMessage**. When passing PROPID_M_AUTHENTICATED to **MQReceiveMessage**, the corresponding VT field in the **aPropVar** array can be set to VT_NULL.

For information on how MSMQ authenticates messages, see Message Authentication.

See Also

MQReceiveMessage, PROPID_M_AUTH_LEVEL, PROPID_Q_AUTHENTICATE

PROPID_M_BODY

The PROPID_M_BODY property contains the body of the message.

Type Indicator
VT_VECTOR | VT_UI1

PROPVARIANT Field
caub

Property Values
Body of the message.

Remarks

The body of a message can consist of any type of information. It is the sending and receiving application's responsibility to understand the type of information that is in the body. For example, the sending application could send a binary file with any internal structure, and it would be the receiving application's responsibility to know how to decipher what was sent.

It is recommended that the sending application set PROPID_M_BODY_TYPE whenever sending messages. If PROPID_M_BODY_TYPE is not set, the application reading the message should assume the message is an array of bytes. MSMQ's ActiveX implementation does this automatically.

Note MSMQ's ActiveX implementation supports the following specific types: VT_I2. VT_UI2, VT_I4, VT_UI4, VT_R4, VT_R8, VT_CY, VT_DATE, VT_BOOL, VT_I1, VT_UI1, VT_BSTR, VT_ARRAY, VT_STREAMED_OBJECT, VT_STORED_OBJECT, where the last two indicate serialized objects that support IPersistStream and IPersistStorage. There are many persistent objects, such as all Microsoft® Office documents, that can be sent as MSMQ messages.

The receiving application can find the size of the message and its type by passing PROPID_M_BODY_SIZE and PROPID_M_BODY_TYPE to **MQReceiveMessage**.

When reading acknowledgment messages from an administration queue, PROPID_M_BODY only returns the original message's body if the acknowledgment message is a negative acknowledgment. Positive acknowledgment messages do not contain the body of the original message. For information on acknowledgment messages, see Acknowledgment Messages.

Example

This example shows how PROPID_M_BODY is specified in the MQMSGPROPS structure:

```
MsgProps.aPropID[i] = PROPID_M_BODY;                    //PropId
MsgProps.aPropVar[i].vt = VT_VECTOR|VT_UI1;             //Type
MsgProps.aPropVar[i].caub.pElems = "Hash hash";         //Value
MsgProps.aPropVar[i].caub.cElems = strlen ("Hash hash")+1;
```

For an example of using PROID_M_BODY, see:

- Sending Private Messages
- Reading Messages Using a Cursor

See Also

MQReceiveMessage, **MQSendMessage**, PROPID_M_BODY_SIZE, PROPID_M_BODY_TYPE

PROPID_M_BODY_SIZE

The PROPID_M_BODY_SIZE property indicates the actual size of the message body.

Type Indicator
VT_UI4 (or VT_NULL)

PROPVARIANT Field
ulVal

Property Values
Size of the message body.

Remarks

PROPID_M_BODY_SIZE is only used when reading messages from a queue. To find the size of a message, pass PROPID_M_BODY_SIZE to **MQReceiveMessage** and look at the returned value. PROPID_M_BODY_SIZE returns the actual size of the message body.

When passing PROPID_M_BODY_SIZE to **MQReceiveMessage**, the corresponding VT field in the **aPropVar** array can be set to VT_NULL.

See Also

MQReceiveMessage, PROPID_M_BODY

PROPID_M_BODY_TYPE

The PROPID_M_BODY_TYPE property indicates the type of body the message contains.

Type Indicator
VT_UI4 (or VT_NULL)

PROPVARIANT Field
ulVal

Property Values
Type of message body. The default is 0.

Remarks

The body of a message can consist of any type of information. It is the sending and receiving application's responsibility to understand the type of information that is in the queue. For example, the sending application could send a binary file with any internal structure, and it would be the receiving application's responsibility to know how to decipher what was sent.

It is recommended that the sending application set PROPID_M_BODY_TYPE whenever sending messages. If PROPID_M_BODY_TYPE is not set, the application reading the message should assume the message is an array of bytes. MSMQ's ActiveX implementation does this automatically.

Note MSMQ's ActiveX implementation supports the following specific types: VT_I2. VT_UI2, VT_I4, VT_UI4, VT_R4, VT_R8, VT_CY, VT_DATE, VT_BOOL, VT_I1, VT_UI1, VT_BSTR, VT_ARRAY, VT_STREAMED_OBJECT, VT_STORED_OBJECT, where the last two indicate serialized objects that support IPersistStream and IPersistStorage. There are many persistent objects, such as all Microsoft Office documents, that can be sent as MSMQ messages.

The receiving application can find the type of a message by passing PROPID_M_BODY_TYPE to **MQReceiveMessage**. When passing this property to **MQReceiveMessage**, the corresponding VT field in the **aPropVar** array can be set to VT_NULL.

See Also

MQReceiveMessage, **MQSendMessage**, PROPID_M_BODY, PROPID_M_BODY_TYPE

PROPID_M_CLASS

The PROPID_M_CLASS property indicates message type. A message can be a normal MSMQ message, a positive or negative (arrival and read) acknowledgment message, or a report message. Typically this property is set by MSMQ when it sends the message, however it can also be set by an MSMQ connector application when the connector application sends a message.

Type Indicator
VT_UI1 (or VT_NULL)

PROPVARIANT Field
uiVal

Property Values

Normal messages (all messages created by your application):

MQMSG_CLASS_NORMAL
A normal MSMQ message.

Positive acknowledgment messages (typically generated by MSMQ):

MQMSG_CLASS_ACK_REACH_QUEUE
The original message reached its destination queue.

MQMSG_CLASS_ACK_RECEIVE
The original message was retrieved by the receiving application.

Negative arrival acknowledgment messages (typically generated by MSMQ):

MQMSG_CLASS_NACK_ACCESS_DENIED
The sending application does not have access rights to the destination queue.

MQMSG_CLASS_NACK_BAD_DST_Q
The destination queue is not available to the sending application.

MQMSG_CLASS_NACK_BAD_ENCRYPTION
The destination Queue Manager could not decrypt a private (encrypted) message (see PROPID_M_PRIV_LEVEL).

MQMSG_CLASS_NACK_BAD_SIGNATURE
MSMQ could not authenticate the original message. The original message's digital signature is not valid.

MQMSG_CLASS_NACK_COULD_NOT_ENCRYPT
The source Queue Manager could not encrypt a private message (PROPID_M_PRIV_LEVEL).

MQMSG_CLASS_NACK_HOP_COUNT_EXCEEDED
The original message's hop count is exceeded.

MQMSG_CLASS_NACK_Q_EXCEED_QUOTA
The original message's destination queue is full.

MQMSG_CLASS_NACK_REACH_QUEUE_TIMEOUT
Either the time-to-reach-queue or time-to-be-received timer expired before the original message could reach the destination queue.

MQMSG_CLASS_NACK_PURGED
The message was purged before reaching the destination queue.

MQMSG_CLASS_NACK_NOT_TRANSACTIONAL_Q
A transaction message was sent to a non-transaction queue.

MQMSG_CLASS_NACK_NOT_TRANSACTIONAL_MSG
A non-transaction message was sent to a transaction queue.

Negative read acknowledgment messages (typically generated by MSMQ):

MQMSG_CLASS_NACK_Q_DELETED
The queue was deleted before the message could be read from the queue.

MQMSG_CLASS_NACK_Q_PURGED

The queue was purged and the message no longer exists.

MQMSG_CLASS_NACK_RECEIVE_TIMEOUT
The original message was not removed from the queue before its time-to-be-received timer expired.

Report messages (typically generated by MSMQ):

MQMSG_CLASS_REPORT
Sent each time the message enters or leaves an MSMQ server.

Remarks

Acknowledgment messages are typically generated by MSMQ whenever the sending application requests them. The acknowledgment message is returned to the administration queue that is specified by the sending application. For information on administration queues, see Administration Queues.

Note Acknowledgment messages can also be created by MSMQ connector applications. When the connector application creates an acknowledgment message, it must set PROPID_M_CLASS and PROPID_CONNECTOR_TYPE. For additional information, see MSMQ Connector Applications.

Report messages are typically generated by MSMQ whenever a report queue is defined at the source Queue Manager. For information on report queues, see Report Queues.

To find the class of a message, pass PROPID_M_CLASS to **MQReceiveMessage** and examine the returned value. When passing PROPID_M_CLASS to **MQReceiveMessage**, the corresponding VT field in the **aPropVar** array can be set to VT_NULL.

When reading messages in an administration queue or dead letter queue, retrieve PROPID_M_CLASS to find out why the message was sent to the queue.

See Also

MQSendMessage, PROPID_M_ACKNOWLEDGE, PROPID_M_PRIV_LEVEL

PROPID_M_CONNECTOR_TYPE

The PROPID_M_CONNECTOR_TYPE property indicates that some message properties typically generated by MSMQ are generated externally from MSMQ. These properties include several security and acknowledgment properties.

Type Indicator
VT_CLSID

PROPVARIANT Field
*puuid

Property Values
Pointer to a connector application identifier.

Remarks

This property is typically required whenever a message is passed by a connector application. The connector type is required so that the sending and receiving applications know how to interpret the security and acknowledgment properties of the messages.

For more information, see MSMQ Connector Server.

PROPID_M_CORRELATIONID

The PROPID_M_CORRELATIONID property specifies the correlation identifier of the message.

Type Indicator
VT_VECTOR | VT_UI1

PROPVARIANT Field
caub

Property Values
Application-defined message identifier (default is 0). Maximum length is 20-bytes.

Remarks

When sending a message, PROPID_M_CORRELATIONID provides an application-defined identifier that the receiving application can use to sort the message.

When sending response messages to the sending application, PROPID_M_CORRELATIONID can be set to the message identifier (PROPID_M_MSGID) of the message that is in the queue. This provides an easy mechanism that the sending application can use to match the response message with the message that was sent.

When MSMQ generates an acknowledgment message, it uses the PROPID_M_CORRELATIONID property to specify the message identifier of the original message. The application can then look at the PROPID_M_CORRELATIONID property to find the message identifier of the original message.

For information on negative and positive acknowledgment messages in administration queues, see Administration Queues.

When sending messages to a foreign queue, the value of PROPID_M_CORRELATIONID is used to verify the sender's signature.

PROPID_M_DELIVERY

The PROPID_M_DELIVERY property specifies how the message is delivered.

Type Indicator
VT_UI1 (or VT_NULL)

PROPVARIANT Field
bVal

Property Values
This property can be set to one of the following values:

MQMSG_DELIVERY_RECOVERABLE
In every hop along its route, the message is forwarded to the next hop or stored locally in a backup file until delivered. This guarantees delivery even in case of a computer crash.

MQMSG_DELIVERY_EXPRESS
The default. The message stays in memory until it can be delivered. The message is not recovered if the computer is rebooted.

Remarks

When the message's delivery mechanism is set to MQMSG_DELIVERY_EXPRESS, the message has faster throughput. When set to MQMSG_DELIVERY_RECOVERABLE, throughput may be slower, but MSMQ guarantees that the message will be delivered, even if a computer crashes while the message is en-route to the queue.

MSMQ always sets the delivery property of transactional messages to MQMSG_DELIVERY_RECOVERABLE. For information on transactions, see MSMQ Transactions.

To see how a message will be delivered, pass PROPID_M_DELIVERY to **MQSendMessage**.

To determine how a message was delivered, pass PROPID_M_DELIVERY to **MQReceiveMessage** and examine its returned value. When passing PROPID_M_DELIVERY to **MQReceiveMessage**, the corresponding VT field in the **aPropVar** array can be set to VT_NULL.

PROPID_M_DEST_QUEUE

The PROPID_M_DEST_QUEUE property identifies the target queue of the message.

Type Indicator
VT_LPWSTR

PROPVARIANT Field
pwszVal

Property Values
Format name of the target queue.

Remarks

The PROPID_M_DEST_QUEUE property is only used by the receiving application. MSMQ attaches this property to the message according to the destination queue specified in the call to **MQSendMessage**.

When PROPID_M_DEST_QUEUE is passed to **MQReceiveMessage**, PROPID_M_DEST_QUEUE_LEN must be passed as well.

PROPID_M_DEST_QUEUE is typically used when the receiving application wants to determine the destination queue of a message in a journal or dead letter queue. For information on journal and dead letter queues, see Journal Queues and Dead Letter Queues.

See Also

MQReceiveMessage, **MQSendMessage**, PROPID_M_DEST_QUEUE_LEN

PROPID_M_DEST_QUEUE_LEN

The PROPID_M_DEST_QUEUE_LEN property indicates the length (in Unicode characters) of the target queue's format name buffer.

Type Indicator
VT_UI4 (or VT_NULL)

PROPVARIANT Field
ulVal

Property Values
Length (in Unicode characters) of the target queue's format name buffer.

Remarks

The PROPID_M_DEST_QUEUE_LEN property is required whenever PROPID_M_DEST_QUEUE is passed as a property in **MQReceiveMessage**.

On input, PROPID_M_DEST_QUEUE_LEN specifies the length of the format name buffer passed in PROPID_M_DEST_QUEUE. The buffer should be large enough to hold the format name string including the null-terminating character.

On return, this property holds the length (in Unicode characters) of the PROPID_M_DEST_QUEUE format name string including the null-terminating character.

If the buffer is too small, **MQReceiveMessage** fails and PROPID_M_DEST_QUEUE_LEN can be used to obtain the required buffer length of the format name string.

See Also

MQReceiveMessage, PROPID_M_DEST_QUEUE

PROPID_M_DEST_SYMM_KEY

The PROPID_M_DEST_SYMM_KEY property specifies the symmetric key used to encrypt messages.

Type Indicator
VT_UI1 | VT_VECTOR

PROPVARIANT Field
caub

Property Values
Encrypted symmetric key

Remarks

This property is used when sending encrypted messages to a foreign queue. The symmetric key is encrypted with the public key of the receiving Queue Manager.

When a connector application receives an encrypted message, it forwards the encrypted message with the attached symmetric key to the receiving application. It is the receiving application's responsibility to decrypt the symmetric key and the body of the message.

See Also

MQReceiveMessage, PROPID_M_DEST_SYMM_KEY_LEN

PROPID_M_DEST_SYMM_KEY_LEN

The PROPID_M_DEST_SYMM_KEY_LEN property specifies the length of the symmetric key used to encrypt the message.

Type Indicator
VT_UI4

PROPVARIANT Field
ulVal

Property Values
Length (in bytes) of the symmetric key.

Remarks

The PROPID_M_DEST_SYMM_KEY_LEN property is set by MSMQ whenever when the message is sent. It is required whenever the receiving application passes PROPID_M_DEST_SYMM_KEY to **MQReceiveMessage**.

When passing PROPID_M_DEST_SYMM_KEY_LEN to **MQReceiveMessage**, the corresponding VT field in the **aPropVar** array can be set to VT_NULL.

See Also

MQReceiveMessage, PROPID_M_DEST_SYMM_KEY

PROPID_M_ENCRYPTION_ALG

The PROPID_M_ENCRYPTION_ALG property specifies the encryption algorithm used to encrypt the message body of a private message.

Type Indicator
VT_UI4

PROPVARIANT Field
ulVal

Property Values
This property can be set to one of the following values:

CALG_RC2 (the default)

CALG_RC4

Remarks

If the message is a private message (PROPID_M_PRIV_LEVEL is set to MQMSG_PRIV_LEVEL_BODY), the source Queue Manager uses this algorithm to encrypt the message and the target Queue Manager uses the matching decryption algorithm to decrypt the message. PROPID_M_ENCRYPTION_ALG is ignored if the message is not a private message (PROPID_M_PRIV_LEVEL). For a complete example of sending private messages (including creating a queue that can only accept private messages), see Sending Private Messages.

CALG_RC2 and CALG_RC4 are defined by the ALG_ID data type in wincrypt.h. For more information on the various encryption methods, see the Microsoft® Cryptographic API (CryptoAPI) in the Microsoft Platform SDK.

When passing PROPID_M_ENCRYPTION_ALG to **MQReceiveMessage**, the corresponding VT field in the **aPropVar** array can be set to VT_NULL.

Example

This example shows how PROPID_M_ENCRYPTION_ALG is specified in the MQMSGPROPS structure:

```
MsgProps.aPropID[i] = PROPID_M_ENCRYPTION_ALG ;   //PropId
MsgProps.aPropVar[i].vt = VT_UI4;                 //Type
MsgProps.aPropVar[i].ulVal = CALG_RC4;            //Value
```

For an example of using PROID_M_ENCRYPTION_ALG, see Sending Private Messages.

See Also

PROPID_M_PRIV_LEVEL

PROPID_M_EXTENSION

The PROPID_M_EXTENSION property provides a place to put additional information that is associated with the message.

Type Indicator
VT_UI1 | VT_VECTOR

PROPVARIANT Field
caub

Property Values
Array of bytes

Remarks

The PROPID_M_EXTENSION property is typically used by applications sending messages to or reading messages from a foreign queue. It is the application's responsibility to understand the content of this property.

The receiving application can determine the length of the information in this property by calling PROPID_M_EXTENSION_LEN.

See Also

PROPID_M_EXTENSION_LEN

PROPID_M_EXTENSION_LEN

The PROPID_M_EXTENSION_LEN property specifies the length of the information in the PROPID_M_EXTENSION property.

Type Indicator
VT_UI4

PROPVARIANT Field
ulVal

Property Values
Length (in bytes) of PROPID_M_EXTENSION.

Remarks

When passing PROPID_M_EXTENSION_LEN to **MQReceiveMessage**, the corresponding VT field in the **aPropVar** array can be set to VT_NULL.

See Also

MQReceiveMessage, PROPID_M_EXTENSION

PROPID_M_HASH_ALG

The PROPID_M_HASH_ALG property identifies the hashing algorithm used when authenticating messages.

Type Indicator
VT_UI4

PROPVARIANT Field
ulVal

Property Values
This property can be set to any of the values defined by the ALG_ID data type in wincrypt.h (the default is CALG_MD5).

Remarks

The MSMQ run-time DLL on the source computer uses the hashing algorithm when creating a digital signature for a message. The target Queue Manger then uses the same hashing algorithm to authenticate the message when it is received.

When passing PROPID_M_HASH_ALG to **MQReceiveMessage**, the corresponding VT field in the **aPropVar** array can be set to VT_NULL.

For information on what MSMQ does to authenticate messages, see How MSMQ Authenticates Messages.

PROPID_M_JOURNAL

The PROPID_M_JOURNAL property specifies if the message should be kept in a machine journal (on the originating machine), sent to a dead letter queue, or neither.

Type Indicator
VT_UI1

PROPVARIANT Field
bVal

Property Values
This property can be set to one or more of the following values:

MQMSG_DEADLETTER
If the message is not delivered to the receiving application (for example, a message timer expired), it is kept on the computer where the message is located.

MQMSG_JOURNAL
If the message is transmitted (from the originating machine to the next hop), it is kept in a journal queue on the originating machine.

MQMSG_JOURNAL_NONE
The default. The message is not kept in the originating machine's journal queue.

Remarks

PROPID_M_JOURNAL does not create a queue. Machine journals and dead letter queues are system queues generated by MSMQ. For information on machine journals and dead letter queues, see Journal Queues and Dead Letter Queues. For an example of reading messages from a machine journal or dead letter queue, see Reading Messages In a Queue.

MSMQ always sends transactional messages to the transaction dead letter queue (DEADXACT) on the source machine if the message is not delivered. For information on transactions, see MSMQ Transactions.

To use a machine journal or deadletter queue, pass PROPID_M_JOURNAL to **MQSendMessage**.

To see if the sending application is using a machine journal or dead letter queue, pass PROPID_M_JOURNAL to **MQReceiveMessage** and check the returned value. When passing PROPID_M_JOURNAL to **MQReceiveMessage**, the corresponding VT field in the **aPropVar** array can be set to VT_NULL.

See Also

MQReceiveMessage, **MQSendMessage**

PROPID_M_LABEL

The PROPID_M_LABEL property specifies a label of the message.

Type Indicator
VT_LPWSTR

PROPVARIANT Field
pwszVal

Property Values
Arbitrary string (the default is none). Maximum length is 250 Unicode characters (including the end-of-string character).

Remarks

If the sending application specifies a message label longer than 250 Unicode characters, MSMQ returns an MQ_ERROR_LABEL_TOO_LONG error to the **aStatus** array.

To find the label of a message, pass PROPID_M_LABEL and PROPID_M_LABEL_LEN to **MQReceiveMessage** and examine the returned values.

Example

This example shows how PROPID_M_LABEL is specified in the MQMSGPROPS structure:

```
MsgProps.aPropID[i] = PROPID_M_LABEL;          //PropId
MsgProps.aPropVar[i].vt = VT_LPWSTR;           //Type
MsgProps.aPropVar[i].pwszVal = L"Hash hash"; //Value
```

For an example of using PROID_M_LABEL, see Sending Private Messages.

See Also

MQReceiveMessage, PROPID_M_LABEL_LEN

PROPID_M_LABEL_LEN

The PROPID_M_LABEL_LEN property identifies the length (in Unicode characters) of the message label buffer.

Type Indicator
VT_UI4

PROPVARIANT Field
ulVal

Property Values
Length (in Unicode characters) of the message label buffer.

Remarks

The PROPID_M_LABEL_LEN property is only used by the receiving application when calling **MQReceiveMessage**. It is required whenever PROPID_M_LABEL is passed to **MQReceiveMessage**.

On input, PROPID_M_LABEL_LEN specifies the size (in Unicode characters) of the message label buffer allocated by the receiving application. The buffer should be large enough to hold the complete label string including the end-of-string character.

On return, this property holds the length (in Unicode characters) of the label string plus the end-of-string character.

MQReceiveMessage succeeds if the buffer is large enough to hold the message label. If the buffer is too small, **MQReceiveMessage** fails and PROPID_M_LABEL_LEN can be used to obtain the required buffer length of the message label string.

Example

The following example allocates a buffer of size 40 for the message label.

```
WCHAR buffer[40];                    //Allocate buffer of size 40.
Prop[0]=PROPID_M_LABEL;
PropVar[0].vt=VT_LPWSTR;
PropVar[0].pwszVal=&buffer[0];

Prop[1]=PROPID_M_LABEL_LEN;
PropVar[1].vt=VT_UI4;
PropVar[1].ulVal=40;                 //Buffer length.

MQReceiveMessage(...);
```

See Also

MQReceiveMessage, PROPID_M_LABEL

PROPID_M_MSGID

The PROPID_M_MSGID property indicates the MSMQ-generated identifier of the message.

Type Indicator
VT_VECTOR | VT_UI1

PROPVARIANT Field
caub

Property Values
20-byte MSMQ-generated message identifier.

Remarks

MSMQ generates a 20-byte message identifier and attaches it to the message when the message is sent. The identifier is an array of bytes that can be read by either the sending or receiving application.

MSMQ generates message identifiers for all messages, including acknowledgment messages generated by MSMQ and MSMQ connector applications. When an acknowledgment message is created, the identifier of the original message can be found in the acknowledgment message's PROPID_M_CORRELATIONID property.

To read a message's identifier, pass PROPID_M_MSGID to **MQSendMessage** when sending the message, or to **MQReceiveMessage**, when reading the message in the queue.

See Also

MQReceiveMessage, **MQSendMessage**, PROPID_M_CORRELATIONID

PROPID_M_PRIORITY

The PROPID_M_PRIORITY property specifies the message's priority. A low number means low priority.

Type Indicator
VT_UI1 (or VT_NULL)

PROPVARIANT Field
bVal

Property Values
An integer value between 7 and 0 (the default is 3).

Remarks

Message priority effects how MSMQ handles the message while it is in route, as well as where the message is placed in the queue. Higher priority messages are given preference during routing, and inserted towards the front of the queue. Messages with the same priority are placed in the queue according to their arrival time.

MSMQ automatically sets the priority level of transactional messages to 0: PROPID_M_PRIORITY is ignored by the transaction. For information on transactions, see MSMQ Transactions.

To set the priority of a message, pass PROPID_M_PRIORITY to **MQSendMessage**.

To determine the priority of a message in the queue, pass PROPID_M_PRIORITY to **MQReceiveMessage** and examine its returned value. When passing PROPID_M_PRIORITY to **MQReceiveMessage**, the corresponding VT field in the **aPropVar** array can be set to VT_NULL.

If the message is sent to a public queue, a second priority (the queue's PROPID_Q_BASEPRIORITY property) is added to PROPID_M_PRIORITY for routing purposes. However, the queue's base priority has no effect on how messages are placed in the queue.

See Also

MQReceiveMessage, PROPID_Q_BASEPRIORITY

PROPID_M_PRIV_LEVEL

The PROPID_M_PRIV_LEVEL property specifies the privacy level of the message to be sent.

Type Indicator
VT_UI4 (or VT_NULL)

PROPVARIANT Field
ulVal

Property Values
This property can be set to one or more of the following values:

MQMSG_PRIV_LEVEL_BODY
Privacy is enforced. End-to-end encryption of the message body is used.

MQMSG_PRIV_LEVEL_NONE
The default. No privacy. The message body is sent as clear text.

Remarks

MSMQ can send private messages throughout the MSMQ enterprise. For a discussion on privacy issues, see Private Messages. For a complete example of sending private messages (including creating a queue that can only accept private messages), see Sending Private Messages.

To send a private message, pass PROPID_M_PRIV_LEVEL to **MQSendMessage**. If PROPID_M_PRIV_LEVEL is set to MQMSG_PRIV_LEVEL_BODY, the body of the message is encrypted using the algorithm specified by PROPID_M_ENCRYPTION_ALG.

To find out if a message was sent encrypted, pass PROPID_M_PRIV_LEVEL to **MQReceiveMessage** and look at the returned value. When passing PROPID_M_PRIV_LEVEL to **MQReceiveMessage**, the corresponding VT field in the **aPropVar** array can be set to VT_NULL.

The target queue can also have its own privacy level (PROPID_Q_PRIV_LEVEL), indicating that it will only receive private (encrypted) messages. If the target queue forces privacy, non-encrypted messages will be rejected.

Example

This example shows how PROPID_M_PRIV_LEVEL is specified in the MQMSGPROPS structure:

```
MsgProps.aPropID[i] = PROPID_M_PRIV_LEVEL;          //PropId
MsgProps.aPropVar[i].vt = VT_UI4;                   //Type
MsgProps.aPropVar[i].ulVal = MQMSG_PRIV_LEVEL_BODY; //Value
```

For an example of using PROID_QM_MACHINE_ID, see Sending Private Messages.

See Also

MQReceiveMessage, **MQSendMessage**, PROPID_M_ENCRYPTION_ALG, PROPID_Q_PRIV_LEVEL

PROPID_M_PROV_NAME

The PROPID_M_PROV_NAME property specifies the name of the cryptographic provider used to generate the message's digital signature.

Type Indicator
VT_LPWSTR

PROPVARIANT Field
pwszVal

Property Values
Unicode string of the provider name (the default is "Microsoft Base Cryptographic Provider, Ver. 1.0").

Remarks

The PROPID_M_PROV_NAME property is typically used when working with foreign queues. The name and type (PROPID_M_PROV_TYPE) of the cryptographic provider is required to validate the digital signature of messages sent to a foreign queue or messages passed to MSMQ from a foreign queue. For information on sending authenticated messages to a foreign queue, see MSMQ Connector Server.

When PROPID_M_PROV_NAME is passed to **MQSendMessage**, MSMQ automatically sets PROPID_M_PROV_NAME_LEN to the length of the provider name.

To find out the name of the cryptographic provider used, pass PROPID_M_PROV_NAME and PROPID_M_PROV_NAME_LEN to **MQReceiveMessage** and examine the returned values.

See Also

MQSendMessage, PROPID_M_PROV_NAME_LEN, PROPID_M_PROV_TYPE

PROPID_M_PROV_NAME_LEN

The PROPID_M_PROV_NAME_LEN property identifies the length of the name of the cryptographic provider used for validating the message signature.

Type Indicator
VT_UI4 (or VT_NULL)

PROPVARIANT Field
ulVal

Property Values
Length of the cryptographic provider name (in Unicode characters).

Remarks

When a message is sent, PROPID_M_PROV_NAME_LEN is set by MSMQ when PROPID_M_PROV_NAME is passed to **MQSendMessage**.

The PROPID_M_PROV_NAME_LEN property is only used by the receiving application when calling **MQReceiveMessage**. It is required whenever PROPID_M_PROV_NAME is passed to **MQReceiveMessage**.

On input, PROPID_M_PROV_NAME_LEN specifies the size (in Unicode characters) of the buffer allocated by the receiving application. The buffer should be large enough to hold the complete provider name string including the end-of-string character.

On return, this property holds the length (in Unicode characters) of the provider name string plus the end-of-string character.

MQReceiveMessage succeeds if the buffer is large enough to hold the provider name. If the buffer is too small, **MQReceiveMessage** fails and PROPID_M_PROV_NAME_LEN can be used to obtain the required buffer length of the provider name string.

When passing PROPID_M_PROV_NAME_LEN to **MQReceiveMessage**, the corresponding VT field in the **aPropVar** array can be set to VT_NULL.

PROPID_M_PROV_NAME is typically used when working with foreign queues. For information on sending authenticated messages to a foreign queue, see MSMQ Connector Server.

See Also

MQSendMessage, PROPID_M_PROV_NAME

PROPID_M_PROV_TYPE

The PROPID_M_PROV_TYPE property specifies the type of cryptographic provider used to generate the message's digital signature.

Type Indicator
VT_UI4

PROPVARIANT Field
ulVal

Property Values
Possible values provided in wincrypt.h (the default is PROV_RSA_FULL).

Remarks

The PROPID_M_PROV_TYPE property is typically used when working with foreign queues. The type and name (PROPID_M_PROV_NAME) of the cryptographic provider is required to validate the digital signature of a message sent to a foreign queue or messages passed to MSMQ from a foreign queue. For information on sending authenticated messages to a foreign queue, see MSMQ Connector Server.

When passing PROPID_M_PROV_TYPE to **MQReceiveMessage**, the corresponding VT field in the **aPropVar** array can be set to VT_NULL.

See Also

PROPID_M_PROV_NAME

PROPID_M_RESP_QUEUE

The PROPID_M_RESP_QUEUE property specifies the queue where application-generated response messages are returned.

Type Indicator
VT_LPWSTR

PROPVARIANT Field
pwszVal

Property Values
Format name of the response queue (the default is none).

Remarks

PROPID_M_RESP_QUEUE is used to send the format name of another queue to the receiving application. Typically, this is done so that the receiving application can send response messages back to the sending application. For information on response queues, see Response Queues.

Note The format name of a private queue (which would be inaccessible otherwise) can also be sent using PROPID_M_RESP_QUEUE.

Messages returned to the queue are application defined. The application must define what is in the messages, as well as what is to be done when a message is received.

To request a response message, pass PROPID_M_RESP_QUEUE to **MQSendMessage**, specifying the queue that will receive the response message.

To check if a response message is required, pass PROPID_M_RESP_QUEUE and PROPID_M_RESP_QUEUE_LEN to **MQReceiveMessage**. If the returned value of PROPID_M_RESP_QUEUE is not NULL, send the response message to the returned queue. If the returned value is NULL a response is not needed.

PROPID_M_RESP_QUEUE can also be used to send the format name of a private queue to another application. This is typically done when the sending application wants to make a private queue available to other applications.

Example

The following example sets PROPID_M_RESP_QUEUE as part of preparing MQMSGPROPS.

```
MQMSGPROPS MsgProps;
PROPVARIANT aVariant[10];
MSGPROPID aPropId[10];
DWORD PropIdCount = 0;

HRESULT hr;

QUEUEHANDLE hQueue;

//
// Set the PROPID_M_RESP_QUEUE property.
//
aPropId[PropIdCount] = PROPID_M_RESP_QUEUE;     //Property identifier.
aVariant[PropIdCount].vt = VT_LPWSTR;             //property type.
aVariant[PropIdCount].pwszVal = szwRESPFormatName;   //An already obtained format
name of the response queue.

PropIdCount++;
```

```
//
// Set other message properties such as PROPID_M_BODY, PROPID_M_LABEL.
//



//
// Set the MQMSGPROPS structure
//
MsgProps.cProp = PropIdCount;          //Number of properties.
MsgProps.aPropID = aPropId;            //Id of properties.
MsgProps.aPropVar = aVariant;          //Value of properties.
MsgProps.aStatus   = NULL;             //No Error report.



//
// Send message.
//
hr = MQSendMessage(
       hQueue,                      // handle to the Queue.
       &MsgProps,                   // Message properties to be sent.
       MQ_NO_TRANSACTION            // No transaction
       );

if (FAILED(hr))
    {
     //
     // Handle error condition
     //
     }
```

See Also

MQReceiveMessage, **MQSendMessage**, PROPID_M_RESP_QUEUE_LEN

PROPID_M_RESP_QUEUE_LEN

The PROPID_M_RESP_QUEUE_LEN property indicates the length (in Unicode characters) of the response queue buffer.

Type Indicator
VT_UI4

PROPVARIANT Field
ulVal

Property Values
Length (in Unicode characters) of the response queue buffer.

Remarks

The PROPID_M_RESP_QUEUE_LEN property is only used by the receiving application when looking to see if a response message is expected or when the sending application has passed the format name of a private queue to the receiving application.

To find the format name of the response queue or private queue, pass PROPID_M_RESP_QUEUE and PROPID_M_RESP_QUEUE_LEN to **MQReceiveMessage**.

On input, PROPID_M_RESP_QUEUE_LEN specifies the length of the format name buffer (in Unicode characters) allocated by the receiving application. The buffer should be large enough to hold the format name string including the null-terminating character.

On return, this property holds the length (in Unicode characters) of the PROPID_M_RESP_QUEUE format name string including the null-terminating character.

MQReceiveMessage succeeds if the buffer is large enough to hold the format name of the administration queue. **MQReceiveMessage** fails if the buffer is not large enough to hold the format name, and PROPID_M_RESP_QUEUE_LEN is set to the required buffer length of the format name string.

Example

The following example allocates a buffer of size 60 for the format name of the response queue, then sets the PROPID_M_RESP_QUEUE and PROPID_M_RESP_QUEUE_LEN properties.

```
MQMSGPROPS MsgProps;
PROPVARIANT aVariant[10];
MSGPROPID aPropId[10];
DWORD PropIdCount = 0;

HRESULT hr;
QUEUEHANDLE hQueue;

//
// Prepare the PROPVARIANT array.
//

#define RESP_QUEUE_BUFF_LEN    = 60
WCHAR szwRespQueueFormatName[RESP_QUEUE_BUFF_LEN];

//
// Set the PROPID_M_RESP_QUEUE property.
//
aPropId[PropIdCount] = PROPID_M_RESP_QUEUE;       //Property identifier.
aVariant[PropIdCount].vt = VT_LPWSTR;              //Property type.
```

```
aVariant[PropIdCount].pwszVal = szwRespQueueFormatName;     //Allocated buffer.
PropIdCount++;

//
// Set the PROPID_M_RESP_QUEUE_LEN property.
//
aPropId[PropIdCount] = PROPID_M_RESP_QUEUE_LEN; //Property identifier.
aVariant[PropIdCount].vt = VT_UI4;                  //Property type.
aVariant[PropIdCount].ulVal = RESP_QUEUE_BUFF_LEN;     //Property value.
PropIdCount++;

//
// Set the MQMSGPROPS structure.
//
MsgProps.cProp = PropIdCount;          //Number of properties.
MsgProps.aPropID = aPropId;            //Id of properties.
MsgProps.aPropVar = aVariant;          //Value of properties.
MsgProps.aStatus   = NULL;              //No Error report.
```

See Also

MQReceiveMessage, PROPID_M_RESP_QUEUE

PROPID_M_SECURITY_CONTEXT

The PROPID_M_SECURITY_CONTEXT property specifies security information that MSMQ uses to authenticate messages.

Type Indicator
VT_UI4

PROPVARIANT Field
ulVal

Property Values
Handle to security context buffer returned by **MQGetSecurityContext**.

Remarks

The PROPID_M_SECURITY_CONTEXT property is only used by the sending application (it is not used when receiving messages) and should be deleted (see **MQFreeSecurityContext**) when the security context is no longer needed to send messages.

The PROPID_M_SECURITY_CONTEXT property is an opaque handle to the security information returned by **MQGetSecurityContext**. This information includes details about the user, as well as information about the user's security certificate (either an external security certificate provided by a certificate authority or the internal security certificate provided by MSMQ).

When authenticating messages, MSMQ must track which sender certificate is associated with which message. Consequently, calling **MQSendMessage** must be done in the same user-context as the call to **MQGetSecurityContext**. If **MQGetSecurityContext** is not called before the message is sent, the security context of the user who originally ran the process is used.

When the application is impersonating a user, the security context of the original user should not be used.

There are two ways to provide the security information for an external certificate. The sending application can provide the complete certificate using PROPID_M_SENDER_CERT, or it can call **MQGetSecurityContext** to retrieve security information from the certificate and place it (along with the user information) in PROPID_M_SECURITY_CONTEXT. When PROPID_M_SENDER_CERT is used, the certificate information in PROPID_M_SECURITY_CONTEXT is ignored but the user information is still valid. Either property can be used when authenticating messages with an external certificate.

See Also

MQFreeSecurityContext, **MQGetSecurityContext**, **MQSendMessage**, PROPID_M_SENDER_CERT

PROPID_M_SENDER_CERT

The PROPID_M_SENDER_CERT property specifies the external certificate used to authenticate messages.

Type Indicator
VT_VECTOR | VT_UI1

PROPVARIANT Field
caub

Property Values
Security certificate (the default is the internal certificate provided by MSMQ).

Remarks

The sending application must obtain an external certificate from a certificate authority, or use the internal certificate provided by MSMQ.

There are two ways to specify the security information provided by an external certificate.

- If the sending application is only going to use the certificate once, place the complete certificate in PROPID_M_SENDER_CERT and attach the property to the message.
- If the sending application is going to use the same certificate over and over, call **MQGetSecurityContext** to retrieve the security information from the certificate, place it in PROPID_M_SECURITY_CONTEXT, and attach the property to the message.

When PROPID_M_SENDER_CERT is used, any certificate information in PROPID_M_SECURITY_CONTEXT is ignored but the user information is still valid. Either property can be used when authenticating messages with an external certificate.

When an external certificate is specified, the receiving application can use the information in the certificate to verify who sent the message. When the internal certificate is specified, the information in this property is not useful to the receiving application.

For information on using an external certificate, see Authenticating Messages Using an External Certificate.

For information on using an internal certificate, see Authenticating Messages Using an Internal Certificate.

See Also

MQGetSecurityContext, **MQReceiveMessage**, PROPID_M_SECURITY_CONTEXT, PROPID_M_SENDER_CERT_LEN

PROPID_M_SENDER_CERT_LEN

The PROPID_M_SENDER_CERT_LEN property specifies the length of the sender certificate buffer.

Type Indicator
VT_UI4

PROPVARIANT Field
ulVal

Property Values
Length (in bytes) of the sender certificate buffer.

Remarks

The PROPID_M_SENDER_CERT_LEN property is only used when the receiving application passes PROPID_M_SENDER_CERT to **MQReceiveMessage**.

On return, PROPID_M_SENDER_CERT_LEN holds the length (in bytes) of the sender certificate.

MQReceiveMessage succeeds if the buffer is large enough to hold the sender certificate. If the buffer is too small, **MQReceiveMessage** fails and PROPID_M_SENDER_CERT_LEN can be used to obtain the required length for the sender certificate buffer. When passing PROPID_M_SENDER_CERT_LEN to **MQReceiveMessage**, the corresponding VT field in the **aPropVar** array can be set to VT_NULL.

See Also

MQReceiveMessage, PROPID_M_SENDER_CERT

PROPID_M_SENDERID

The PROPID_M_SENDERID property identifies who sent the message.

Type Indicator
VT_VECTOR | VT_UI1

PROPVARIANT Field
caub

Property Values
An array of bytes generated by MSMQ.

Remarks

The PROPID_M_SENDERID property is primarily used by MSMQ security to authenticate messages when they are retrieved from the queue. For information on authenticating messages, see Authenticating Messages Using API Functions.

The PROPID_M_SENDERID property is also used internally to verify that the sender has access rights to the queue. Verification is done by the receiving Queue Manager when it receives the message.

When a message is sent (and PROPID_M_SENDERID_TYPE is not set to MQMSG_SENDERID_TYPE_NONE), MSMQ attaches PROPID_M_SENDERID to the message.

Receiving applications can pass PROPID_M_SENDERID to **MQReceiveMessage** to verify who sent a message. To find out the length of the buffer needed for the sender identifier, peek at the message by calling **MQReceiveMessage** (specifying PROPID_M_SENDERID_LEN) to find out the length of the sender identifier. Then call **MQReceiveMessage** again (specifying PROPID_M_SENDERID), setting the caub.cElems field of PROPID_M_SENDERID to the actual length returned by the first call to **MQReceiveMessage**.

See Also

MQReceiveMessage, PROPID_M_SENDERID_LEN, PROPID_M_SENDERID_TYPE

PROPID_M_SENDERID_LEN

The PROPID_M_SENDERID_LEN property indicates the length of the sender identifier.

Type Indicator
VT_UI4

PROPVARIANT Field
ulVal

Property Values
Length (in bytes) of the sender identifier.

Remarks

The PROPID_M_SENDERID_LEN property is only used by the receiving application when calling **MQReceiveMessage**. MSMQ attaches this property to the message, along with PROPID_M_SENDERID, when the message is sent.

The receiving application can use PROPID_M_SENDERID_LEN to determine the length of the buffer that is needed for the sender identifier. To do this, peek at the message by calling **MQReceiveMessage** (specifying PROPID_M_SENDERID_LEN) to find out the length of the sender identifier. Then call **MQReceiveMessage** again (specifying PROPID_M_SENDERID), setting the caub.cElems field of PROPID_M_SENDERID to the actual length returned by the first call to **MQReceiveMessage**.

When PROPID_M_SENDERID_LEN is passed to **MQReceiveMessage**, the corresponding VT field in the **aPropVar** array can be set to VT_NULL.

See Also

MQReceiveMessage, PROPID_M_SENDERID

PROPID_M_SENDERID_TYPE

The PROPID_M_SENDERID_TYPE property specifies the type of sender identifier found in PROPID_M_SENDERID.

Type Indicator
VT_UI4 (or VT_NULL)

PROPVARIANT Field
ulVal

Property Values
This property can have one of the following values:

MQMSG_SENDERID_TYPE_SID
The default. PROPID_M_SENDERID contains the SID of the user sending the message.

MQMSG_SENDERID_TYPE_NONE
No identifier is attached to the message.

Remarks

If the sending application does not want MSMQ to attach a sender identifier to a message, it can specify MQMSG_SENDERID_TYPE_NONE when passing PROPID_M_SENDERID_TYPE to **MQSendMessage**. This suppresses the message's PROPID_M_SENDERID property.

The receiving application can pass PROPID_M_SENDERID_TYPE to **MQReceiveMessage** to determine what type of sender identifier was attached to the message. The returned value for PROPID_M_SENDERID_TYPE can be one of the following:

- MQMSG_SENDERID_TYPE_NONE: No identifier was attached to the message.
- MQMSG_SENDERID_TYPE_SID: The sending user's SID was attached to the message.

When the receiving application specifies PROPID_M_SENDERID_TYPE, the corresponding VT field in the **aPropVar** array can be set to VT_NULL.

Local users (users not logged into a Windows NT domain) cannot attach an SID to a message. The SID of a local user is only valid locally.

See Also

MQReceiveMessage, **MQSendMessage**, PROPID_M_SENDERID

PROPID_M_SENTTIME

The PROPID_M_SENTTIME property indicates the date and time that the message was sent by the source Queue Manager.

Type Indicator
VT_UI4 (or VT_NULL)

PROPVARIANT Field
ulVal

Remarks

The PROPID_M_SENTTIME property is attached to the message by MSMQ. The time returned is the number of seconds elapsed since midnight (00:00:00), January 1, 1970 (Coordinated Universal time) according to the system clock.

To see when a message was sent, pass PROPID_M_SENTTIME to **MQReceiveMessage**, and examine the returned value. When passing PROPID_M_SENTTIME to **MQReceiveMessage**, the corresponding VT field in the **aPropVar** array can be set to VT_NULL.

For information on how to find out when the message arrived at the target queue, see PROPID_M_ARRIVEDTIME.

See Also

MQReceiveMessage, PROPID_M_ARRIVEDTIME

PROPID_M_SIGNATURE

The PROPID_M_SIGNATURE property specifies the digital signature used to authenticate the message.

Type Indicator
VT_UI1 | VT_VECTOR

PROPVARIANT Field
caub

Property Values
Digital signature.

Remarks

Typically MSMQ attaches PROPID_M_SIGNATURE to a message when it is sent. However, connector applications can also attach this property to a message when calling **MQSendMessage**.

When a connector application attaches this property to a message, it must generate the digital signature of the user sending the message. (MSMQ does not generate a signature when it sees that the connector application has supplied the signature.) The following properties are used to computed the digital signature in the order shown:

- PROPID_M_CORRELATIONID
- PROPID_M_APPSPECIFIC
- PROPID_M_BODY
- PROPID_M_LABEL
- PROPID_M_RESP_QUEUE
- PROPID_M_ADMIN_QUEUE

For a information on authenticating messages sent by a connector application. see Connector Application Security.

See Also

MQReceiveMessage, PROPID_M_SIGNATURE_LEN

PROPID_M_SIGNATURE_LEN

The PROPID_M_SIGNATURE_LEN property indicates the length of the message's digital signature.

Type Indicator
VT_UI4

PROPVARIANT Field
ulVal

Property Values
Length (in bytes) of the digital signature.

Remarks

This property is used for messages sent with a digital signature to an MSMQ Connector.

When passing PROPID_M_SIGNATURE_LEN to **MQReceiveMessage**, the corresponding VT field in the **aPropVar** array can be set to VT_NULL.

See Also

MQReceiveMessage, PROPID_M_SIGNATURE

PROPID_M_SRC_MACHINE_ID

The PROPID_M_SRC_MACHINE_ID property specifies the computer where the message originated.

Type Indicator
VT_CLSID

PROPVARIANT Field
*puuid

Property Values
Machine identifier (GUID) of source machine.

Remarks

This property is only used by the receiving application when calling **MQReceiveMessage**.

PROPID_M_SRC_MACHINE_ID returns the GUID of the computer without the enclosing brackets {}.

See Also

MQReceiveMessage

PROPID_M_TIME_TO_BE_RECEIVED

The PROPID_M_TIME_TO_BE_RECEIVED property specifies the total time (in seconds) the message is allowed to live. This includes the time spent getting to its destination queue plus the time spent waiting in the queue before it is retrieved by an application.

Type Indicator
VT_UI4 (or VT_NULL)

PROPVARIANT Field
ulVal

Property Values
Integer value (the default is INFINITE).

Remarks

PROPID_M_TIME_TO_BE_RECEIVED sets the *time-to-be-received* timer. For a discussion of message timers, see Message Timers. If the timer expires before the message is removed from the queue, MSMQ discards the message, sending it to the dead letter queue if the message's PROPID_M_JOURNAL property is set to MQMSG_DEADLETTER.

MSMQ can also send a return negative acknowledgment message back to the sending application if the message is not removed in time and the message's PROPID_M_ACKNOWLEDGE property is set accordingly.

To set the time-to-be-received timer, pass PROPID_M_TIME_TO_BE_RECEIVED to **MQSendMessage**.

To find out how much time remains in the time-to-be-received timer, pass PROPID_M_TIME_TO_BE_RECEIVED to **MQReceiveMessage** and look at the returned value. When passing PROPID_M_TIME_TO_BE_RECEIVED to **MQReceiveMessage**, the corresponding VT field in the **aPropVar** array can be set to VT_NULL.

If the time-to-be-received and time-to-reach-queue timers are both specified, the time-to-be-received timer takes precedence over the time-to-reach-queue timer.

MSMQ uses the *time-to-be-received* timer of the first message when several messages are sent in a transaction. For information on transactions, see MSMQ Transactions.

When MSMQ creates an acknowledgment message, it always sets the message's *time-to-be-received* timer to INFINITE.

See Also

MQReceiveMessage, **MQSendMessage**, PROPID_M_ACKNOWLEDGE, PROPID_M_JOURNAL, PROPID_M_TIME_TO_REACH_QUEUE

PROPID_M_TIME_TO_REACH_QUEUE

The PROPID_M_TIME_TO_REACH_QUEUE property specifies a time limit (in seconds) for the message to reach the queue.

Type Indicator
VT_UI4

PROPVARIANT Field
ulVal

Property Values
Integer value (the default is LONG_LIVED).

Remarks

PROPID_M_TIME_TO_REACH_QUEUE sets the message's *time-to-reach-queue* timer. For a discussion of message timers, see Message Timers. If the *time-to-reach-queue* timer expires before the message reaches its destination, MSMQ discards the message, sending it to the dead letter queue if the message's PROPID_M_JOURNAL property is set to MQMSG_DEADLETTER.

MSMQ can also send a return negative acknowledgment messages back to the sending application if the message does not arrive and the message's PROPID_M_ACKNOWLEDGE property is set accordingly.

To set the time-to-reach-queue timer, pass PROPID_M_TIME_TO_REACH_QUEUE to **MQSendMessage**.

The default value LONG_LIVED is an enterprise-wide setting that can be adjusted by the MSMQ Administrator. Typically, LONG_LIVED is set to 90 days. Although this timer can be set to INFINITE, MSMQ automatically uses the LONG_LIVED value in its place.

To find out how much time remains in the time-to-reach-queue timer, pass PROPID_M_TIME_TO_REACH_QUEUE to **MQReceiveMessage** and look at the returned value. A value of 0 indicates the timer has expired.

If the time-to-be-received and time-to-reach-queue timers are both specified, the time-to-be-received timer takes precedence over the time-to-reach-queue timer.

No matter what value PROPID_M_TIME_TO_REACH_QUEUE is set to (even if set to 0), MSMQ always gives each message one chance to reach its destination if the queue is waiting for the message. If the queue is local, the message always reaches the queue.

MSMQ uses the *time-to-reach-queue* timer of the first message when several messages are sent in a transaction. For information on transactions, see MSMQ Transactions.

When MSMQ creates an acknowledgment message, it always sets the message's *time-to-reach-queue* timer to LONG_LIVED.

See Also

MQReceiveMessage, **MQSendMessage**, PROPID_M_ACKNOWLEDGE, PROPID_M_JOURNAL, PROPID_M_TIME_TO_BE_RECEIVED

PROPID_M_TRACE

The PROPID_M_TRACE property specifies where report messages will be sent when tracing a message.

Type Indicator
VT_UI1 (or VT_NULL)

PROPVARIANT Field
bVal

Property Values
This property can be set to one of the following values:

MQMSG_SEND_ROUTE_TO_REPORT_QUEUE
Each hop made by the original message generates a report that is recorded in a report message. The elements of the report are the source Queue Manager, message identifier, target, time, and next hop. The report message is sent to the report queue specified by the source Queue Manager.

MQMSG_TRACE_NONE
The default. No tracing for this message.

Remarks

If MQMSG_SEND_ROUTE_TO_REPORT_QUEUE is specified but the report queue is not defined by the MSMQ Administrator for the message's source Queue Manager, this property is ignored.

When passing PROPID_M_TRACE to **MQReceiveMessage**, the corresponding VT field in the **aPropVar** array can be set to VT_NULL.

For a description of report queues and messages, see Report Queues and Report Messages.

For information on machine journals, see Journal Queues.

PROPID_M_VERSION

The PROPID_M_VERSION property specifies the version of MSMQ used to send the message.

Type Indicator
VT_UI4

PROPVARIANT Field
ulVal

Property Values
0x0010

Remarks

MSMQ attaches PROPID_M_VERSION to the message when it is sent.

The receiving application can pass PROPID_M_VERSION to **MQReceiveMessage** to find out what version of MSMQ the sending application is using.

See Also

MQReceiveMessage

PROPID_M_XACT_STATUS_QUEUE

The PROPID_M_XACT_STATUS_QUEUE property identifies the transaction status queue on the source computer. It is only used when sending messages to a foreign queue.

Type Indicator
VT_LPWSTR

PROPVARIANT Field
pwszVal

Property Values
Format name of transaction status queue.

Remarks

PROPID_M_XACT_STATUS_QUEUE is only used by MSMQ connector applications to send positive (read receipt) or negative acknowledgment messages back to the sending application. MSMQ attaches this property to each transaction message sent to a foreign queue. For information on transactions and connector applications, see Using the MSMQ Connector in a Transaction.

The transaction status queue should receive these acknowledgments even if the sending application does not request other acknowledgments. See PROPID_M_ACKNOWLEDGE.

For information on the other properties that must be set when creating acknowledgment messages, see

When the connector application passes PROPID_M_XACT_STATUS_QUEUE to **MQReceiveMessage**, PROPID_M_XACT_STATUS_QUEUE_LEN must be passed as well. If the length property is not included, MQ_ERROR_INSUFFICIENT_PROPERTIES is returned to **MQReceiveMessage**.

See Also

MQReceiveMessage, PROPID_M_ACKNOWLEDGE, PROPID_M_XACT_STATUS_QUEUE_LEN

PROPID_M_XACT_STATUS_QUEUE_LEN

The PROPID_M_XACT_STATUS_QUEUE_LEN property indicates the length (in Unicode characters) of the transaction status queue's format name buffer.

Type Indicator
VT_UI4

PROPVARIANT Field
ulVal

Property Values
Length (in Unicode characters) of the transaction status queue's format name buffer.

Remarks

The PROPID_M_XACT_STATUS_QUEUE_LEN property is only used by the connector application when calling **MQReceiveMessage**. It is required whenever PROPID_M_XACT_STATUS_QUEUE is passed in **MQReceiveMessage**.

On input, PROPID_M_XACT_STATUS_QUEUE_LEN specifies the length of the format name buffer (in Unicode characters) allocated by the receiving application. The buffer should be large enough to hold the format name string including the null-terminating character.

On return, this property holds the length (in Unicode characters) of the PROPID_M_XACT_STATUS_QUEUE format name string including the null-terminating character.

See Also

MQReceiveMessage, PROPID_M_XACT_STATUS_QUEUE

PROPID_Q_AUTHENTICATE

Optional. The PROPID_Q_AUTHENTICATE property specifies whether or not the queue only accepts authenticated messages.

Type Indicator
VT_UI1

PROPVARIANT Field
bVal

Property Values
This property can be set to one of the following values:

MQ_AUTHENTICATE_NONE
The default. The queue accepts authenticated and non-authenticated messages.

MQ_AUTHENTICATE
The queue only accepts authenticated messages.

Remarks

If the authentication level of the message (PROPID_M_AUTH_LEVEL) does not match the authentication level of the queue, the message is rejected by the queue. In addition, if the sending application requested a negative acknowledgment message when it sent the message, MQMSG_CLASS_NACK_BAD_SIGNATURE will be returned to the sending application to indicate the message was rejected.

For information on how MSMQ authenticates messages, see Message Authentication.

To set the authentication level of the queue, pass PROPID_Q_AUTHENTICATE to **MQCreateQueue** when creating the queue.

To change the authentication level of the queue, pass PROPID_Q_AUTHENTICATE to **MQSetQueueProperties**. When changing the authentication level of the queue, the new setting only impacts arriving messages; it does not affect messages already in the queue.

To determine the authentication level of a queue, pass PROPID_Q_AUTHENTICATE to **MQGetQueueProperties** and examine its returned value.

The receiving application can also check if a message was authenticated by looking at the message's PROPID_M_AUTHENTICATED property.

See Also

MQCreateQueue, **MQGetQueueProperties**, **MQSetQueueProperties**, PROPID_M_AUTH_LEVEL, PROPID_M_AUTHENTICATED

PROPID_Q_BASEPRIORITY

Optional. The PROPID_Q_BASEPRIORITY property specifies a single base priority for all messages sent to a public queue.

Type Indicator
VT_I2

PROPVARIANT Field
iVal

Property Values
Integer value between -32768 and +32767 (the default is 0).

Remarks

A public queue's base priority is used for routing the queue's messages over the network. It can be used to give the messages sent to the queue a higher (or lower) priority than messages sent to other queues. For example, when a queue's base priority is set, all the messages sent to it are given a higher priority than messages sent to queues with a lower base priority. The queue's base priority has no effect on the order of the messages in the queue, or how messages are read from the queue.

PROPID_Q_BASEPRIORITY only applies to public queues that can be located through MQIS (using a public format name). The base priority of private queues, as well as public queues accessed directly, is always 0.

MSMQ combines the queue's base priority with the message's priority (PROPID_M_PRIORITY) to determine the overall priority of a message when it is sent to the queue.

To set the base priority of a public queue, pass PROPID_Q_BASEPRIORITY to **MQCreateQueue**.

To change the base priority of a public queue, pass PROPID_Q_BASEPRIORITY to **MQSetQueueProperties**.

To determine the base priority of a queue, pass PROPID_Q_BASEPRIORITY to **MQGetQueueProperties** and examine its returned value.

See Also

MQCreateQueue, **MQGetQueueProperties**, **MQSetQueueProperties**, PROPID_M_PRIORITY

PROPID_Q_CREATE_TIME

Optional read-only. The PROPID_Q_CREATE_TIME property indicates the time and date when the queue was created.

Type Indicator
VT_I4

PROPVARIANT Field
lVal

Property Values
Time when the queue was created.

Remarks

This property is set by MSMQ when **MQCreateQueue** is called. An error is returned (MQ_ERROR_PROPERTY_NOTALLOWED) if any attempt is made to set this property. The time returned is the number of seconds elapsed since midnight (00:00:00), January 1, 1970 (Coordinated Universal time) according to the system clock.

There are several related C run-time functions that can be used to manipulate the value of PROPID_Q_CREATE_TIME. For example, **ctime()** can be used to display the local date and time that the queue was created.

To determine when the queue was created, pass PROPID_Q_CREATE_TIME to **MQGetQueueProperties** and examine its returned value.

See Also

MQCreateQueue, **MQGetQueueProperties**

PROPID_Q_INSTANCE

Optional read-only. The PROPID_Q_INSTANCE property identifies a specific public queue.

Type Indicator
VT_CLSID

PROPVARIANT Field
*puuid

Property Values
GUID (globally unique identifier) of queue.

Remarks

The PROPID_Q_INSTANCE property identifies the created public queue (it is not an instance of an open queue). This property is set by MSMQ when the application calls **MQCreateQueue**. An MQ_ERROR_PROPERTY_NOTALLOWED error is returned if any attempt is made to set this property.

PROPID_Q_INSTANCE only applies to public queues. An MQ_INFORMATION_PROPERTY_IGNORED error is returned if an attempt is made to get this property for a private queue.

To find the identifier of a public queue, pass PROPID_Q_INSTANCE to **MQGetQueueProperties** or **MQLocateBegin** (when starting a query) and examine its returned value.

See Also

MQCreateQueue, **MQGetQueueProperties**, **MQLocateBegin**

PROPID_Q_JOURNAL

Optional. The PROPID_Q_JOURNAL property specifies if messages retrieved from the queue are also copied to its journal queue.

Type Indicator
VT_UI1

PROPVARIANT Field
bVal

Property Values
This property can be set to one of the following values:

MQ_JOURNAL
All messages removed from the specified queue are stored in its journal queue.

MQ_JOURNAL_NONE
The default. Messages are not stored in a journal queue. All messages removed from the specified queue are discarded.

Remarks

The PROPID_Q_JOURNAL property does not create a journal, it specifies whether or not messages removed from the queue are stored in the journal queue. Journal queues are system queues created by MSMQ (for information on journal queues, see Journal Queues). The application can only read the messages in a journal.

To save removed messages in a journal queue, pass PROPID_Q_JOURNAL to **MQCreateQueue** when creating the queue.

To start or stop storing messages in the journal queue, pass PROPID_Q_JOURNAL to **MQSetQueueProperties**. When the property value is changed, the remaining messages retrieved from the specified queue will be stored or discarded according to the new setting.

To determine if removed messages are being stored in the journal queue, pass PROPID_Q_JOURNAL to **MQGetQueueProperties** and examine its returned value.

To specify the size of a queue journal, see PROPID_Q_JOURNAL_QUOTA.

See Also

MQCreateQueue, **MQGetQueueProperties**, **MQSetQueueProperties**, PROPID_Q_JOURNAL_QUOTA

PROPID_Q_JOURNAL_QUOTA

Optional. The PROPID_Q_JOURNAL_QUOTA property specifies the maximum size (in kilobytes) of the journal queue.

Type Indicator
VT_UI4

PROPVARIANT Field
ulVal

Property Values
Size (in kilobytes) of the journal queue (the default is INFINITE).

Remarks

To set the size of journal queue, pass PROPID_Q_JOURNAL_QUOTA to **MQCreateQueue**.

To change the size of a journal queue, pass PROPID_Q_JOURNAL_QUOTA to **MQSetQueueProperties**.

To find the size of a journal queue, pass PROPID_Q_JOURNAL_QUOTA to **MQGetQueueProperties** and examine its returned value.

See Also

MQCreateQueue, **MQGetQueueProperties**, **MQSetQueueProperties**, PROPID_Q_JOURNAL

PROPID_Q_LABEL

Optional. The PROPID_Q_LABEL property specifies a description of the queue.

Type Indicator
VT_LPWSTR

PROPVARIANT Field
pwszVal

Property Value
String (default is ""). The maximum length of the string is MQ_MAX_Q_LABEL_LEN (124 Unicode characters).

Remarks

The queue's label can be used to identify the queue.

For public queues, the queue's label can be used as the search criteria for a query. By using the same label for several queues, the application can later run a query on the queue label and locate all the queues. (A query can also be used to retrieve the label of a public queue.)

To specify the label of a queue, pass PROPID_Q_LABEL to **MQCreateQueue**.

To change the label of a queue, pass PROPID_Q_LABEL to **MQSetQueueProperties**.

To find the label of a queue, pass PROPID_Q_LABEL to **MQGetQueueProperties** and examine its returned value. When calling **MQGetQueueProperties**, set the type indicator for PROPID_Q_LABEL to VT_NULL so that MSMQ will allocate the memory needed for the label. Later, the allocated memory must be freed using **MQFreeMemory**.

Example

This example shows how PROPID_Q_LABEL is specified in the MQQUEUEPROPS structure:

```
aPropId[PropIdCount] = PROPID_Q_LABEL;      'PropId
aVariant[PropIdCount].vt = VT_LPWSTR;       'Type
aVariant[PropIdCount].pwszVal = L"MyPublicQueue";
```

For an example of using PROID_Q_LABEL, see Creating a Queue.

See Also

MQCreateQueue, **MQGetQueueProperties**, **MQSetQueueProperties**

PROPID_Q_MODIFY_TIME

Optional. The PROPID_Q_MODIFY_TIME property indicates the last time the properties of a queue were modified.

Type Indicator
VT_I4

PROPVARIANT Field
lVal

Property Values
Time when queue properties were last set.

Remarks

This property is set by MSMQ when **MQCreateQueue** is called, then reset by MSMQ each time the queue properties are modified by calls to **MQSetQueueProperties**. The time returned is the number of seconds elapsed since midnight (00:00:00), January 1, 1970 (Coordinated Universal time) according to the system clock.

There are several related C run-time functions that can be used to manipulate the value of PROPID_Q_MODIFY_TIME. For example, **ctime()** can be used to display the local date and time when the queue properties were last modified.

To determine when the queue properties were last modified, pass PROPID_Q_MODIFY_TIME to **MQGetQueueProperties** or **MQLocateBegin** (when starting a query) and examine its returned value.

An MQ_ERROR_PROPERTY_NOTALLOWED error is returned if any attempt is made to set this property.

See Also

MQCreateQueue, **MQGetQueueProperties**, **MQLocateBegin**, **MQSetQueueProperties**

PROPID_Q_PATHNAME

Required (to create the queue). The PROPID_Q_PATHNAME property specifies the MSMQ pathname of the queue. The MSMQ pathname includes the name of the computer where the queue's message's are stored, if the queue is public or private, and the name of the queue.

Type Indicator
VT_LPWSTR

PROPVARIANT Field
pwszVal

Property Values
MSMQ pathname.

Remarks

The PROPID_Q_PATHNAME property is the only property required when creating a queue. To specify the queue's MSMQ pathname, pass PROPID_Q_PATHNAME to **MQCreateQueue**. An MQ_ERROR_PROPERTY_NOTALLOWED error is returned if any attempt is made to set this property after the queue is created.

For public queues, PROPID_Q_PATHNAME includes the name of the computer where the queue's messages are stored, followed by the name of the queue. For private queues, add PRIVATE$\ between the name of the local computer and the queue name (private queues can only be created on the local computer).

Here are three examples of MSMQ pathnames. The first two examples indicate two public queues (one on a local computer and the other on a remote computer), and the third example indicates a private queue.

```
"myMachine\myPublicQueue"
"otherMachine\otherPublicQueue"
"myMachine\Private$\myPrivateQueue"
```

As a shortcut, you can substitute a period "." for the local machine. So *myPublicQueue* and *myPrivateQueue* could be specified on the local machine as:

```
".\myPublicQueue"
".\Private$\myPrivateQueue"
```

Private queues are only created on the local computer. It is the application's responsibility to ensure that all queue names on the local computer are unique. If a queue name already exists when **MQCreateQueue** is called, MSMQ returns an MQ_ERROR_QUEUE_EXISTS error to the application.

To find the MSMQ pathname of a queue, pass PROPID_Q_PATHNAME to **MQGetQueueProperties** or **MQLocateBegin** (when starting a query) and examine its returned value. When passing PROPID_Q_PATHNAME to **MQGetQueueProperties**, set its type indicator to VT_NULL. This tells MSMQ to allocate the memory needed for the pathname. Later, this allocated memory must be freed using **MQFreeMemory**.

To create a foreign queue, specify the name of the *foreign machine* as it is defined in MQIS. (For information on defining *foreign machines*, see “To create a foreign computer” in the Administrator’s Guide.)

Example

This example shows how PROPID_Q_PATHNAME is specified in the MQQUEUEPROPS structure (for a public queue):

```
QProps.aPropID[i] = PROPID_Q_PATHNAME;                  //PropId
```

```
    QProps.aPropVar[i].vt = VT_LPWSTR;                  //Type
    QProps.aPropVar[i].pwszVal = L".\\MyPublicQueue";   //Value
```

For an example of using PROID_Q_PATHNAME, see Creating a Queue.

See Also

MQCreateQueue, **MQGetQueueProperties**, **MQLocateBegin**

PROPID_Q_PRIV_LEVEL

Optional. The PROPID_Q_PRIV_LEVEL property specifies the privacy level that is required by the queue. The privacy level determines how the queue handles encrypted messages.

Type Indicator
VT_UI4

PROPVARIANT Field
ulVal

Property Values
This property can be set to one of the following values:

MQ_PRIV_LEVEL_NONE
The queue accepts only non-private (clear) messages.

MQ_PRIV_LEVEL_BODY
The queue accepts only private (encrypted) messages.

MQ_PRIV_LEVEL_OPTIONAL
The default. The queue does not force privacy. It accepts private (encrypted) messages and non-private (clear) messages.

Remarks

The application can set the privacy level of queues and messages. If the privacy level of the message (PROPID_M_PRIV_LEVEL) does not match the privacy level of the queue, the message is rejected by the queue. In addition, if the sending application requested a negative acknowledgment message when it sent the message, MQMSG_CLASS_BAD_ENCRYPTION will be returned to the sending application to indicate the message was rejected.

The privacy level of a message is set by PROPID_M_PRIV_LEVEL. When a message is marked private, its message body is encrypted by MSMQ when the message is sent. For information on security issues, see MSMQ Security Services.

To set the privacy level of a queue, pass PROPID_Q_PRIV_LEVEL to **MQCreateQueue** when creating the queue.

To change the privacy level of a queue, pass PROPID_Q_PRIV_LEVEL to **MQSetQueueProperties**. When changing the privacy level of the queue, the new setting only impacts arriving messages; it does not affect messages already in the queue.

To determine the privacy level of a queue, pass PROPID_Q_PRIV_LEVEL to **MQGetQueueProperties** and examine its returned value.

See Also

MQCreateQueue, **MQGetQueueProperties**, **MQSetQueueProperties**, PROPID_M_PRIV_LEVEL

PROPID_Q_QUOTA

Optional. The PROPID_Q_QUOTA property specifies the maximum size (in kilobytes) of the queue.

Type Indicator
VT_UI4

PROPVARIANT Field
ulVal

Property Values
Size (in kilobytes) of the queue. The default is INFINITE.

Remarks

The PROPID_Q_QUOTA property is typically set when calling **MQCreateQueue**.

When a queue's quota is reached, a negative acknowledgment is returned to the administration queue of the sending application to indicate that the queue is full. MSMQ continues to send negative acknowledgments until the cumulative size of messages in the queue drop below the queue's quota.

To find the size of a queue, pass PROPID_Q_QUOTA to **MQGetQueueProperties** and examine its returned value.

To change the size of a queue, pass PROPID_Q_QUOTA to **MQSetQueueProperties** with the new value. When the queue's quota is changed, the new quota only impacts arriving messages; it does not affect messages already in the queue.

See Also

MQCreateQueue, **MQGetQueueProperties**, **MQSetQueueProperties**

PROPID_Q_TRANSACTION

Optional. The PROPID_Q_TRANSACTION property specifies whether the queue is a transaction queue or a non-transaction queue.

Type Indicator
VT_UI1

PROPVARIANT Field
bVal

Property Values
This property can be set to one of the following values:

MQ_TRANSACTIONAL
All messages sent to the queue must be done through an MSMQ transaction.

MQ_TRANSACTIONAL_NONE
The default. No transaction operations can be performed on the queue.

Remarks

If a queue is transactional, it can only accept messages that are sent as part of a transaction (see **MQSendMessage**). However, messages can be retrieved from a local transaction queue with or without using a transaction (see **MQReceiveMessage**).

For information on how MSMQ performs transactions, see MSMQ Transactions.

To create a transaction queue, set PROPID_Q_TRANSACTION to MQ_TRANSACTIONAL and pass it to **MQCreateQueue**.

PROPID_Q_TRANSACTION cannot be changed once the queue is created. If an attempt is made to set it afterward, an MQ_ERROR_PROPERTY error is returned to the call and the property's associated **aStatus** entry will contain MQ_PROPERTY_NOTALLOWED.

To determine if the queue is a transaction queue, pass PROPID_Q_TRANSACTION to **MQGetQueueProperties** and examine its returned value.

See Also

MQCreateQueue, **MQGetQueueProperties**, **MQReceiveMessage**, **MQSendMessage**

PROPID_Q_TYPE

Optional. The PROPID_Q_TYPE property specifies the type of service provided by the queue. The queue's type allows applications to categorize their queues according to how they are used.

Type Indicator
VT_CLSID

PROPVARIANT Field
*puuid

Property Values
GUID value. The default is NULL_GUID.

Remarks

The queue's service type can be used to identify the queue.

It is recommended that the service type of the queue be specified when the queue is created. In most cases, the service type of the queue can be defined by the application. However, some queues used by MSMQ require a specific MSMQ-defined service type. For example, input queues used by the MSMQ Mail Services have a specific MSMQ-defined MAPI service type.

Note To generate a GUID, run the UUIDGEN.EXE program provided by Microsoft Developer Network. (For information about UUIDGEN.EXE, see the Microsoft Platform SDK.)

The queue's service type can also be used to locate public queues registered in MQIS (see **MQLocateBegin** and **MQLocateNext**,).

To set the queue's service type, pass PROPID_Q_TYPE to **MQCreateQueue**.

To change the queue's service type, pass PROPID_Q_TYPE to **MQSetQueueProperties** with a new GUID.

To find the service type of a queue, pass PROPID_Q_TYPE to **MQGetQueueProperties** and examine its returned value.

See Also

MQCreateQueue, **MQGetQueueProperties**, **MQLocateBegin**, **MQLocateNext**, **MQSetQueueProperties**

PROPID_QM_CONNECTION

The PROPID_QM_CONNECTION property identifies the CN (Connected Network) list of the computer.

Type Indicator
VT_LPWSTR | VT_VECTOR

PROPVARIANT Field
calpwstr

Property Values
<CN TYPE><CN GUID><CN Name>

Remarks

This property is typically used by connector applications when retrieving the CN list of a computer.

To retrieve the CN list of a computer, pass PROPID_QM_MACHINE_ID to **MQGetMachineProperties** and examine its returned value.

See Also

MQGetMachineProperties

PROPID_QM_ENCRYPTION_PK

The PROPID_QM_ENCRYPTION_PK property indicates the public encryption key of the computer.

Type Indicator
VT_UI1 | VT_VECTOR

PROPVARIANT Field
caub

Property Values
Public encryption key.

Remarks

The PROPID_QM_ENCRYPTION_PK property of the computer is set by MSMQ when MSMQ is installed.

To find the public encryption key of the computer, pass PROPID_QM_ENCRYPTION_PK to **MQGetMachineProperties** and examine its returned value.

See Also

MQGetMachineProperties

PROPID_QM_MACHINE_ID

The PROPID_QM_MACHINE_ID property identifies the computer.

Type Indicator
VT_CLSID

PROPVARIANT Field
*puuid

Property Values
Machine GUID.

Remarks

The PROPID_QM_MACHINE_ID property is set by MSMQ when the computer is defined. It is used when specifying the format name of a private queue, a machine journal, or a dead letter queue.

To determine the identifier of the machine, pass PROPID_QM_MACHINE_ID to **MQGetMachineProperties** and examine its returned value.

Note For dependent clients, MSMQ sets this property to the identifier of the client's server computer.

Example

This examples shows how PROPID_QM_MACHINE_ID is specified in the MQQMPROPS structure:

```
PropId = PROPID_QM_MACHINE_ID;          //PropId
Variant.vt = VT_CLSID;                  //Type
Variant.puuid = &guidMachineId;         //Value
```

For an example of using PROID_QM_MACHINE_ID, see:

- Reading Messages in a Dead Letter Queue
- Reading Messages in a Machine Journal

See Also

MQGetMachineProperties

PROPID_QM_PATHNAME

The PROPID_QM_PATHNAME property specifies the MSMQ pathname of the computer.

Type Indicator
VT_LPWSTR

PROPVARIANT Field
pwszVal

Property Values
MSMQ pathname.

Remarks

To find the MSMQ pathname of a computer, pass PROPID_QM_PATHNAME to **MQGetMachineProperties** and examine its returned value. When calling **MQGetMachineProperties**, the type identifier of this property must be set to VT_NULL.

Note For dependent clients, MSMQ sets this property to the pathname of the client's server computer.

See Also

MQGetMachineProperties

PROPID_QM_SITE_ID

The PROPID_QM_SITE_ID property identifies the site where the computer is located.

Type Indicator
VT_CLSID

PROPVARIANT Field
*puuid

Property Values
Site identifier.

Remarks

PROPID_QM_SITE_ID is set by MSMQ when the computer is created. It is typically used to determine the site where the Queue Manager resides

The computer's site identifier indicates where the Queue Manager is currently located. When the computer changes location to another site, MSMQ automatically updates PROPID_QM_SITE_ID to indicate the new site.

To find the site of the computer, pass PROPID_QM_SITE_ID to **MQGetMachineProperties** and examine its returned value.

See Also

MQGetMachineProperties

MSMQ Structures

MSMQ uses structures for message properties, queue properties, and Queue Manager properties, as well as structures for queries to locate queues. For information about how these structures work together, see Property Structures.

MSMQ structures include:

- **aPropID**
- **aStatus**
- **MQCOLUMNSET**
- **MQMailTnefData**
- **MQMSGPROPS**
- **MQPROPERTYRESTRICTION**
- **MQQMPROPS**
- **MQQUEUEPROPS**
- **MQRESTRICTION**
- **MQSORTKEY**
- **MQSORTSET**
- **PROPVARIANT**

aPropID

The **aPropID** structure is an array used to specify queue, message, and machine property identifiers. It is used to identify which properties are specified for a call. MSMQ uses three different types of property identifiers: QUEUEPROPID, MSGPROPID, and QMPROPID.

```
typedef PROPID    MSGPROPID
typedef PROPID    QUEUEPROPID
typedef PROPID    QMPROPID
```

aStatus

The **aStatus** structure is an optional array that contains errors returned by MSMQ. It can be used in **MQQUEUEPROPS**, **MQMSGPROPS**, and **MQQMPROPS**.

Position **i** in this array is a reported status code of the property whose identifier and value are in position **i** in the corresponding property identifier and property value arrays.

MSMQ errors are divided into four categories (by increasing order of severity); the category can be determined by looking at the upper two bits of the error code (success = 00, informational = 01, warning = 10, fatal = 11).

For information on how property structures work together, see Property Structures.

See Also

MQMSGPROPS, **MQQMPROPS**, **MQQUEUEPROPS**

MQCOLUMNSET

The **MQCOLUMNSET** structure specifies the number of queue properties to be retrieved from each queue and their property identifiers. You can specify any number of queue properties, from a single queue property to all the properties provided by MSMQ. MSMQ can return any number of queue properties to the single query.

```
typedef struct   tagMQCOLUMNSET
{
    ULONG                cCol;
    PROPID _RPC_FAR    *aCol;
}    MQCOLUMNSET;
```

Members

cCol

Number of properties to be retrieved.

***aCol**

An array of property identifiers (for example, PROPID_Q_PATHNAME, PROPID_Q_INSTANCE, and so on).

See Also

MQLocateBegin

MQMSGPROPS

The **MQMSGPROPS** structure describes a set of message properties. It specifies the number of properties (**cProp**) in the set, the identifier of each property (**aPropID**), the values (or placeholder for values) of each property (**aPropVar**), and it also provides an optional status array (**aStatus**) for errors (some properties do not return a status) associated with the property.

For information on how property structures work together, see Property Structures.

```
typedef struct   tagMQMSGPROPS
{
    DWORD              cProp;
    MSGPROPID          aPropID[];
    PROPVARIANT        aPropVar[];
    HRESULT            aStatus[];
} MQMSGPROPS;
```

Members

cProp
Number of properties in the set.

aPropID
Property identifiers (for example, PROPID_M_LABEL, PROPID_M_MSGID, PROPID_M_BODY, and so on). For information about the structure, see **aPropID**.

aPropVar
Value of properties in set. For information about the structure used by **aPropVar**, see **PROPVARIANT**.

aStatus
Optional. Returns errors that indicate the status of the properties in the set, for example, when a property cannot be set by the call. For information about the structure, see **aStatus**.

See Also

aPropID, **aStatus**, **PROPVARIANT**

MQPROPERTYRESTRICTION

The **MQPROPERTYRESTRICTION** structure defines a property restriction for a query. The property restriction consists of a comparison operator, a property identifier, and a property value.

```
typedef struct   tagMQPROPERTYRESTRICTION
{
  ULONG                rel;
  PROPID             prop;
  PROPVARIANT       prval;
} MQPROPERTYRESTRICTION;
```

Members

rel

Comparison operator for this property. Possible values include: less than (PRLT), less than or equal to (PRLE), equal (PREQ), not equal (PRNE), greater than or equal to (PRGE), and greater than (PRGT).

prop

Queue property identifier.

prval

Comparison value (includes variant type and value for the property). For information about the structure used by **prval**, see **PROPVARIANT**.

See Also

MQLocateBegin, **MQRESTRICTION**, **PROPVARIANT**

MQQMPROPS

The **MQQMPROPS** structure describes a set of Queue Manager properties. It specifies the number of properties (**cProp**) in the set, the identifier of each property (**aPropID**), and the values (or placeholder for values) of each property (**aPropVar**). It also provides an optional status array (**aStatus**) for errors (some properties do not return a status) associated with the property.

For information on how property structures work together, see Property Structures.

```
typedef struct   tagMQQMPROPS
{
    DWORD                cProp;
    QMPROPID          aPropID[];
    PROPVARIANT     aPropVar[];
    HRESULT           aStatus[];
} MQQMPROPS;
```

Members

cProp
Number of properties in the set.

aPropID
Property identifiers (for example, PROPID_QM_PATHNAME). For information about the structure, see **aPropID**.

aPropVar
Value of properties in the set. For information about the structure used by **aPropVar**, see **PROPVARIANT**.

aStatus
Optional. Returns errors that indicate the status of the properties in the set. For example, when a property cannot be set by the call. For information about the structure, see **aStatus**.

See Also

aPropID, **aStatus**, **PROPVARIANT**

MQQUEUEPROPS

The **MQQUEUEPROPS** structure describes a set of queue properties. It specifies the number of properties (**cProp**) in the set, the identifier of each property (**aPropID**), and the values (or placeholder for values) of each property (**aPropVar**). It also provides an optional status array (**aStatus**) for errors (some properties do not return a status) associated with the property.

For information on how property structures work together, see Property Structures.

```
typedef struct   tagMQQUEUEPROPS
{
    DWORD                cProp;
    QUEUEPROPID       aPropID[];
    PROPVARIANT      aPropVar[];
    HRESULT           aStatus[];
} MQQUEUEPROPS;
```

Members

cProp
Number of properties in the set.

aPropID
Property identifiers (for example, PROPID_Q_PATHNAME). For information about the structure, see **aPropID**.

aPropVar
Value of properties in the set. For information about the structure used by **aPropVar**, see **PROPVARIANT**.

aStatus
Optional. Returns errors that indicate the status of the properties in the set. For example, when a property cannot be set by the call. For information about the structure, see **aStatus**.

See Also

aPropID, **aStatus**, **PROPVARIANT**

MQRESTRICTION

The **MQRESTRICTION** structure indicates the properties used to locate queues in a query. Only those public queues that match all the restrictions specified here are returned by the query.

For information on starting the query, see **MQLocateBegin**.

```
typedef struct   tagMQRESTRICTION
{
  ULONG                                cRes;
  MQPROPERTYRESTRICTION   _RPC_FAR *paPropRes;

} MQRESTRICTION;
```

Members

cRes

Number of property restrictions to use in the query.

***paPropRes**

An array of property restrictions. To specify a property restriction, see **MQPROPERTYRESTRICTION**.

See Also

MQLocateBegin, **MQPROPERTYRESTRICTION**

MQSORTKEY

The **MQSORTKEY** structure specifies a sort key for the query. Each key contains a queue property and sort order.

To specify multiple sort keys, see **MQSORTSET**.

```
typedef struct tagMQSORTKEY
{
    PROPID   propColumn;
    ULONG    dwOrder;
}    MQSORTKEY;
```

Members

propColumn
Queue property to sort on (for example PROPID_Q_QUOTA).

dwOrder
Order of sort. Possible values are QUERY_SORTASCEND and QUERY_SORTDESCEND.

See Also

MQLocateBegin, **MQSORTSET**

MQSORTSET

The **MQSORTSET** structure specifies the sort keys for a query.

```
typedef struct   tagMQSORTSET
{
    ULONG                   cCol;
    MQSORTKEY   _RPC_FAR *aCol;
}    MQSORTSET;
```

Members

cCol

Number of sort keys used to sort the results of the query.

***aCol**

An array of sort keys. Sort keys are used in the order they appear in the array. For more information on specifying a sort key, see **MQSORTKEY**.

See Also

MQLocateBegin, **MQSORTKEY**

PROPVARIANT

The **PROPVARIANT** structure is a general structure used to store property values. It is used for the elements of the **aPropVar** array used in **MQQUEUEPROPS**, **MQMSGPROPS**, and **MQQMPROPS**; the **prval** array of **MQPROPERTYRESTRICTION**; and as a parameter in **MQLocateNext**.

Property values are tagged values, where the tag is the type indicator (an integer value) passed as an instance of **PROPVARIANT** (a data type that will be part of Automation in the future). MSMQ uses a similar structure (included here for reference) along with some other Automation types and definitions on which this structure relies. The complete definition of the Automation **PROPVARIANT** structure can be found in the header file ‹oleext.h› and in the additional header files it includes.

For information on how property structures work together, see Property Structures.

```
struct     MQtagPROPVARIANT   {
    VARTYPE vt;                         /* value tag                */
    WORD wReserved1;
    WORD wReserved2;
    WORD wReserved3;
    union    {
             UCHAR               bVal;        /* VT_UI1                   */
             short               iVal;        /* VT_I2                    */
             USHORT              uiVal;       /* VT_UI2                   */
             VARIANT_BOOL        bool;        /* VT_BOOL                  */
             long                lVal;        /* VT_I4                    */
             ULONG               ulVal;       /* VT_UI4                   */
             SCODE               scode;       /*                          */
             DATE                date;        /* VT_DATE                  */
             CLSID   _RPC_FAR   *puuid;       /* VT_CLSID                 */
             BLOB                blob;        /* VT_BLOB                  */
             LPOLESTR            bstrVal;     /*                          */
             LPSTR               pszVal;      /* VT_LPSTR                 */
             LPWSTR              pwszVal;     /* VT_LPWSTR                */
             CAUI1               caub;        /* VT_VECTOR | VT_UI1       */
             CAI2                cai;         /* VT_VECTOR | VT_I2        */
             CAUI2               caus;        /* VT_VECTOR | VT_UI2       */
             CABOOL              cabool;      /* VT_VECTOR | VT_BOOL      */
             CAI4                cal;         /* VT_VECTOR | VT_I4        */
             CAUI4               caul;        /* VT_VECTOR | VT_UI4       */
             CACLSID             cauuid;      /* VT_VECTOR | VT_CLSID     */
             CABSTR              cabstr;      /* VT_VECTOR | VT_BSTR      */
             CALPWSTR            calpwstr;    /* VT_VECTOR | VT_LPWSTR */
             CAPROPVARIANT       capropvar;   /*                          */
            };
};

typedef struct MQtagPROPVARIANT PROPVARIANT;
```

Members

vt
The type indicator of the property. The valid indicators for the VT field are a subset of the Automation **VARENUM** enumeration type (including VT_NULL, VT_I2, VT_I4, VT_LPWSTR, VT_UI1, VT_UI4, VT_CLSID, VT_VECTOR|VT_UI1, VT_VECTOR|VT_LPWSTR).

wReserved1, wReserved2, wReserved3
Reserved by MSMQ.

union

Specifies the value of the property. Depending on the type identifier specified by **VT**, the corresponding member of the union holds the value of the property.

Remarks

To specify a property (with the exception of some message properties passed to **MQReceiveMessage**) and queue properties passed to **MQGetQueueProperties**, you must know its type indicator and the member of the **union** associated with the type indicator (both are provided with each property description). For example, to specify the message body in PROPID_M_BODY, the application must set **VT** to VT_UI1| VT_VECTOR and assign the message body to the **caub** member of the **union**.

CA prefixed members of the **union** (**caub** through **capropvar**) are constructs used to pass buffers as counted arrays.

For example, **CAUI1** is a counted array of bytes:

```
typedef struct tagCAUI1 {
    ULONG cElems;              /* Byte Counter                    */
    unsigned char *pElems;     /* Pointer to a buffer of bytes   */
} CAUI1;
```

See Also

MQLocateNext, **MQMSGPROPS**, **MQPROPERTYRESTRICTION**, **MQQMPROPS**, **MQQUEUEPROPS**

MSMQ Mail Structures

The following topics describe the MSMQ Mail structures.

MSMQ Mail structures include:

- **MQMailDeliveryReportData**
- **MQMailEMail**
- **MQMailEMailType**
- **MQMailFormData**
- **MQMailFormField**
- **MQMailFormFieldData**
- **MQMailFormFieldList**
- **MQMailFormFieldType**
- **MQMailMessageData**
- **MQMailNonDeliveryReportData**
- **MQMailRecip**
- **MQMailRecipList**
- **MQMailRecipType**
- **MQMailTnefData**

MQMailDeliveryReportData

The **MQMailDeliveryReportData** structure describes a delivery report message. It lists the recipients who received the original mail, the original mail subject, and the original mail submission time.

The *pftDeliverTime* member of each recipient in the *pDeliveredRecips* recipient list specifies the time when the original mail was delivered to the recipient.

```
typedef struct MQMailDeliveryReportData_tag
  {
   LPMQMailRecipList   pDeliveredRecips;   //Delivered recipients.
   LPSTR               szOriginalSubject; //Original subject.
   LPFILETIME          pftOriginalDate;    //Original submission time.
} MQMailDeliveryReportData, FAR * LPMQMailDeliveryReportData;
```

See Also

MQMailEMail, **MQMailRecipList**

MQMailEMail

The **MQMailEMail** structure describes mail information in a mail format. It is used for translating information between the basic mail format and the MSMQ Mail format used by MSMQ Mail services.

```
typedef struct   MQMailEMail_tag
{
   LPSTR                    szSubject;   //Subject of mail.
   BOOL                     fRequestDeliveryReport;
   BOOL                     fRequestNonDeliveryReport;
   LPFILETIME               pftDate;      //Time sent.
   LPMQMailRecip            pFrom;        //Sender.
   LPMQMailRecipList        pRecips;      //List of recipients.
   LPMQMailEMailType        iType;        //Type of Email (message, form)
   union                                  //Union of available Email types.
   {
      MQMailFormData        form;         //when type is MQMailEMail_FORM
      MQMailMessageData     message;      //when type is MQMailEMail_MESSAGE
      MQMailTnefData        tnef          //when type is MQMailEMail_TNEF
      MQMailDeliveryReportData          DeliveryReport; //when type is
MQMailEMail_DELIVERY_REPORT
      MQMailNonDeliveryReportData       NonDeliveryReport; //when type is
MQMailEMail_NON_DELIVERY_REPORT
   };
   LPVOID                   pReserved;   //Should be set to NULL.
} MQMailEMail, FAR* LPMQMailEMail;
```

For information on MSMQ Mail services, see MSMQ MAPI Transport Provider and MSMQ Exchange Connector.

See Also

MQMailEMailType, **MQMailFormData**, **MQMailDeliveryReportData**, **MQMailMessageData**, **MQMailNonDeliveryReportData**, **MQMailRecip**, **MQMailRecipList**, **MQMailTnefData**

MQMailEMailType

The **MQMailEMailType** structure describes the type of mail: text message, form, TNEF message, delivery report, or non-delivery report e-mail.

```
typedef enum    MQMailEMailType_enum
{
    MQMailEMail_MESSAGE,                    //Mail is text message.
    MQMailEMail_FORM,                       //Mail is form with fields.
    MQMailEMail_TNEF,                       //Mail is in MAPI TNEF format.
    MQMailEMail_DELIVERY_REPORT             //Mail is a delivery report.
    MQMailEMail_NON_DELIVERY_REPORT         //Mail is a non-delivery report.
} MQMailEMailType;
```

See Also

MQMailEMail

MQMailFormData

The **MQMailFormData** structure describes a message form, giving the name of the form and listing the fields the form contains.

```
typedef struct   MQMailFormData_tag
{
   LPSTR                    szName;       //Name of form.
   LPMQMailFormFieldList    pFields;      //List of fields.
} MQMailFormData, FAR* LPMQMailFormData;
```

See Also

MQMailEMail, **MQMailFormFieldList**

MQMailFormField

The **MQMailFormField** structure describes a form field. The MSMQ Mail SDK supports String, Integer, Boolean, Double, and Currency fields.

```
typedef struct   MQMailFormField_tag
{
   LPSTR                    szName;    //Name of field.
   MQMailFormFieldType   iType;     //Type of value.
   MQMailFormFieldData   Value;     //Union of available types.
} MQMailFormField, FAR*LPMQMailFormField;
```

See Also

MQMailEMail, **MQMailFormFieldData**, **MQMailFormFieldType**

MQMailFormFieldData

The **MQMailFormFieldData** structure describes the data in a form field.

```
typedef union    MQMailFormFieldData_tag
{
    BOOL                b;       //Specifies MQMailFormField_BOOL.
    LPSTR               lpsz;    //Specifies MQMailFormField_STRING.
    LONG                l        //Specifies MQMailFormField_LONG.
    CY                  cy       //Specifies MQMailFormField_CURRENCY.
    double              dbl      //Specifies MQMailFormField_DOUBLE.

} MQMailFormFieldData, FAR*LPMQMailFormFieldData;
```

See Also

MQMailEMail, **MQMailFormField**

MQMailFormFieldList

The **MQMailFormFieldList** structure describes a list of form fields.

```
typedef struct    MQMailFormFieldList_tag
{
    ULONG                       cFields;    //Number of fields.
    LPMQMailFormField FAR*      apField;    //Pointer to array of field ptrs.
} MQMailFormFieldList, FAR*LPMQMailFormFieldList;
```

See Also

MQMailEMail, **MQMailFormField**

MQMailFormFieldType

The **MQMailFormFieldDataType** structure describes the field type.

```
typedef enum    MQMailFormFieldType_enum
{
    MQMailFormField_BOOL,              //Boolean data.
    MQMailFormField_STRING,            //String data.
    MQMailFormField_LONG,              //Long data.
    MQMailFormField_CURRENCY,           //Currency data.
    MQMailFormField_DOUBLE,            //Double data.
} MQMailFormFieldType;
```

See Also

MQMailEMail, **MQMailFormField**

MQMailMessageData

The **MQMailMessageData** structure describes a mail message.

```
typedef struct    MQMailMessageData_tag
{
    LPSTR szText;                              //Text of message.
} MQMailMessageData, FAR* LPMQMailMessageData;
```

See Also

MQMailEMail

MQMailNonDeliveryReportData

The **MQMailNonDeliveryReportData** structure describes a non-delivery report message. It contains the recipients to whom the original mail was not delivered, and the original mail in the *pOriginalEMail* member.

The *szNonDeliveryReason* member of each recipient in the *pNonDeliveredRecips* recipient list specifies the reason why the original mail was not delivered to the recipient.

```
typedef struct MQMailNonDeliveryReportData_tag
  {
   LPMQMailRecipList   pNonDeliveredRecips;   //Non-delivered recipients.
   LPMQMailEMail       pOriginalEMail;        //Original mail.
} MQMailNonDeliveryReportData, FAR * LPMQMailNonDeliveryReportData;
```

See Also

MQMailEMail, **MQMailRecipList**

MQMailRecip

The **MQMailRecip** structure describes a recipient.

```
typedef struct   MQMailMessageData_tag
{
   LPSTR       szName;                 //Name of recipient.
   LPSTR       szQueueLabel;           //Label of recipient's queue.
   LPSTR       szAddress;              //Recipient address.
   LPFILETIME pftDeliveryTime;        //Used in delivery reports.
   LPSTR       szNonDeliveryReason;   //Used in non-delivery reports.
} MQMailRecip, FAR*LPMQMailRecip;
```

Remarks

The recipient address can be an application input queue label (for MSMQ applications), a MAPI client queue label (for the MSMQ MAPI Transport Provider) or the MSMQ Exchange user mail alias plus the MSMQ Exchange Connector's queue label in the format user-mail-alias@Exchange connector-queue-label (for MSMQ Exchange Connector).

pftDeliveryTime is valid only when the recipient is in a delivery report recipient list, e.g. the *pDeliveredRecips* member of the **MQMailDeliveryReportData** structure.

szNonDeliveryReason is valid only when the recipient is in a non-delivery report recipient list, e.g. the *pNonDeliveredRecips* member of the **MQMailNonDeliveryReportData** structure.

See Also

MQMailEMail, **MQMailDeliveryReportData**, **MQMailNonDeliveryReportData**, **MQMailRecipList**

MQMailRecipList

The **MQMailRecipList** structure lists the recipients of the mail.

```
typedef struct    MQMailRecipList_tag
{
    ULONG                    cRecips;    //Number of recipients.
    LPMQMailRecip FAR*    apRecip;    //Array of recipient pointers.
} MQMailRecipList, FAR*LPMQMailRecipList;
```

See Also

MQMailEMail, **MQMailRecip**

MQMailRecipType

The **MQMailRecipType** structure describes the type of recipient.

```
typedef enum MQMailRecipType_enum
{
    MQMailRecip_TO,      //Primary recipient.
    MQMailRecip_CC,      //Copied recipient.
    MQMailRecip_BCC,     //Hidden (blind copied) recipient.

} MQMailRecipType;
```

See Also

MQMailEMail

MQMailTnefData

The **MQMailTnefData** structure describes a TNEF message, providing the binary TNEF data.

TNEF is a MAPI internal format that encapsulates the MAPI properties, and is used by the MSMQ Mail services (the MSMQ Exchange Connector and the MSMQ MAPI Transport) to send mail to recipients who are defined as rich-text recipients. Rich-text recipients are recipients who have the “Send to this recipient in Microsoft Exchange rich text format” check-box checked in their Exchange/MAPI address.

```
typedef struct MQMailTnefData_tag
{
  ULONG      cbData;                  //Size of the TNEF data.
  LPBYTE lpbData;               //The tnef data buffer.
} MQMailTnefData, FAR * LPMQMailTnefData;
```

See Also

MQMailEMail

MSMQ ActiveX Components

MSMQ provides ActiveX components that support queue lookup, queue management, message management, queue administration, and transaction support. As a group, these components provide most of the MSMQ API functionality.

MSMQ ActiveX Objects

The MSMQ objects include:

- **MSMQQuery**
- **MSMQQueueInfos**
- **MSMQQueueInfo**
- **MSMQQueue**
- **MSMQEvent**
- **MSMQMessage**
- **MSMQCoordinatedTransactionDispenser**
- **MSMQTransaction**
- **MSMQTransactionDispenser**
- **MSMQApplication**

MSMQQuery

Methods

The **MSMQQuery** object allows you to query MQIS for existing public queues.

Queries are based on the properties of the queue, and their results are returned to **MSMQQuery** in an **MSMQQueueInfos** object.

MSMQQueueInfos

Methods

The **MSMQQueueInfos** object allows you to select a specific public queue from a collection of queues returned by calling the **LookupQueue** method of **MSMQQuery**.

The information in the collection of queues is dynamic, with other clients looking at the queues (PEEK access) or deleting them from the collection at any time. As a result, there is no queue count available for moving through the collection. In place of a queue count, **MSMQQueueInfos** provides an end-of-list (EOL) mechanism to indicate when you have completely moved through the collection.

The following example shows how the EOL mechanism is used in a standard Microsoft® Visual Basic® type **While** loop.

```
'Display the format name of all MAPI input queues.
Dim queryMyQuery as new MSMQQuery
Dim qinfoCurrent as MSMQQueueInfo
'Get some interesting queues.
Dim qs as MSMQQueueInfos
Set qs = queryMyQuery.LookupQueue (
    ServiceTypeGuid := "{5EADC0D0-7182-11CF-A8FF-0020AFB8FB50}")
qs.Reset
Set qinfoCurrent = qs.Next
While not qinfoCurrent is Nothing
       msgbox qinfoCurrent.FormatName
       Set qinfoCurrent = qs.Next
Wend
```

MSMQQueueInfo

Properties **Methods**

The **MSMQQueueInfo** object provides queue management. It allows you to create a queue (either a transaction or non-transaction queue), open an existing queue, change a queue's properties, and delete a queue.

MSMQQueueInfo objects are either returned by a query or created by you. There is a one-to-one relationship between each **MSMQQueueInfo** object and the queue it represents. However, there is also a one-to-many relationship between the queue's **MSMQQueueInfo** object and each open instance of the queue. (Each instance of a queue is referenced by an **MSMQQueue** object.)

MSMQQueue

Properties **Methods**

The **MSMQQueue** object represents an MSMQ queue. It provides cursor-like behavior for traversing the messages of an open queue. At any given moment, it refers to a particular position in the queue.

For information on how to create or open a queue, see the **Create** and **Open** methods of **MSMQQueueInfo**.

MSMQEvent

Events

The **MSMQEvent** object can be used to implement a single event handler that can support multiple queues. These events include a message arriving at the queue, an error occurring while the message is being delivered to the queue, or (when asynchronously reading messages) no message arriving at the queue before its receive timeout timer expires.

Each MSMQ queue can be associated with an instance of the **MSMQEvent** object. This allows the applications to have one generic event handler that can treat all common tasks identically, and use the queue handle passed to the event handler for special-case situations.

Note The formal parameter passed to the **MSMQEvent_Arrived** event is of type Object (this is required because event firing is implemented in terms of **IDispatch**), which means that the event handler uses ActiveX late-binding by default instead of early-binding. However, early-binding is more efficient and the applications may choose to assign the formal parameter to a local variable of type **MSMQQueue** to regain the benefits of early-binding.

For an example of an event handler, see Reading Messages Asynchronously.

See Also

MSMQQueue

MSMQMessage

Properties **Methods**

The **MSMQMessage** object provides properties to define MSMQ messages, plus a single **Send** method for sending the message to its destination queue. This single method is used for transaction and non-transaction messages (for information on transactions, see MSMQ Transactions).

The body of an MSMQ message can be a string, an array of bytes, any numeric, date, and currency type that a variant can contain, or any persistent ActiveX object that supports **IDispatch** and **IPersist** (**IPersistStream** or **IPersistStorage**).

MSMQCoordinatedTransactionDispenser

Methods

The **MSMQCoordinatedTransactionDispenser** object is used to obtain an MS DTC transaction object. When the transaction is obtained, an **MSMQTransaction** object is returned that can be to send or retrieve messages.

For information on all MS DTC transactions, see MS DTC External Transactions.

MSMQTransaction

Methods **Properties**

The **MSMQTransaction** object represents a transaction object obtained externally using **MSMQCoordinatedTransactionDispenser**, or created internally using **MSMQTransactionDispenser**. **MSMQTransaction** provides methods for committing to or terminating the transaction, as well as a single read-only property that represents the underlying transaction object.

In order to use the transaction object, the **MSMQTransaction** object must be specified by the *pTransaction* parameter of one of the following methods:

- **MSMQQueue.Receive**
- **MSMQQueue.ReceiveCurrent**
- **MSMQMessage.Send**

See Also

Receive, **ReceiveCurrent**, **Send**

MSMQTransactionDispenser

Methods

The **MSMQTransactionDispenser** object is used to create a new MSMQ internal transaction object. When the internal transaction object is created, an **MSMQTransaction** object is returned that can be used to send or retrieve messages.

For information on internal transactions, see MSMQ Internal Transactions.

MSMQApplication

Methods

The **MSMQApplication** object provides a single method for obtaining the machine identifier of a computer.

Note Creating a new instance of this object does not start a new instance of MSMQ.

Since it is an application object, **MSMQApplication** does not have to be explicitly referenced when calling **MachineIdOfMachineName**. For example, the following three calls to **MachineIdOfMachineName** all return the same computer identifier.

```
Dim strId As String
Dim myapp As New MSMQApplication
strId = MachineIdOfMachineName("machinename")
Debug.Print strId
strId = MSMQApplication.MachineIdOfMachineName("machinename")
Debug.Print strId
strId = myapp.MachineIdOfMachineName("machinename")
Debug.Print strId
```

ActiveX Methods

The following topics describe the methods associated with the ActiveX components provided by MSMQ.

The methods of the MSMQ objects include:

MSMQQuery

LookupQueue

MSMQQueueInfos

Next

Reset

MSMQQueueInfo

Create

Delete

Open

Refresh

Update

MSMQQueue

Close

EnableNotification

Peek

PeekCurrent

PeekNext

Receive

ReceiveCurrent

Reset

MSMQMessage

Send

AttachCurrentSecurityContext

MSMQCoordinatedTransactionDispenser

BeginTransaction

MSMQTransaction

Abort

Commit

MSMQTransactionDispenser

BeginTransaction

MSMQApplication

MachineIdOfMachineName

Abort

MSMQTransaction

The **Abort** method terminates the transaction associated with the **MSMQTransaction** object.

Syntax

object.**Abort** ([*fRetaining*][*, fAsync*])

Syntax Element	Description
object	Transaction (**MSMQTransaction**) object that identifies the transaction.
fRetaining	Optional. Boolean.
fAsync	Optional. Boolean.

Remarks

After an abort, all actions taken on the queue are rolled back. For example, if a message is sent to a queue and the transaction is aborted, the messages are not sent to the queue.

Abort is a wrapper for ITransaction::Abort. For information on ITransaction::Abort, see the Microsoft Platform SDK.)

Example

This example starts a transaction, sends two messages, and then terminates the transaction.

```
Dim xdispenser as New MSMQCoordinatedTransactionDispenser
Dim xact as MSMQTransaction

Dim qSend as MSMQQueue
Dim msg1 as New MSMQMessage
Dim msg2 as New MSMQMessage

Set xact = xdispenser.BeginTransaction
Set qSend = qinfo.Open(MQ_SEND_ACCESS, MQ_DENY_NONE)
msg1.Label = "MyTransaction message"
msg1.Body = "Message 1 Body"
msg1.Send qSend, xact                    'Associates send with xact.

msg2.Label = "MyTransaction message"
msg2.Body = "Message 2 Body"
msg2.Send qSend, xact                    'Associates send with xact.

xact.Abort                               'Aborts transaction.
```

See Also

BeginTransaction, **Body**, **Label**, **Open**, **Send**

AttachCurrentSecurityContext

MSMQMessage

The **AttachCurrentSecurityContext** method retrieves security context information from a specific certificate.

Quick Info

Type:	**Long**
Run time:	read/write

Syntax

object.**AttachCurrentSecurityContext**

Syntax Element	Description
object	Message (**MSMQMessage**) object that defines the message.

Remarks

Security context information can be retrieved from the security certificate specified by **SenderCertificate**. **SenderCertificate** can specify an external certificate or an internal certificate.

Although the sending application can explicitly retrieve security context information from the security certificate, **AttachCurrentSecurityContext** can retrieve and cache the needed information using a single call, then automatically pass the information along with the message when it is sent.

For information on how MSMQ authenticates messages, see Message Authentication.

BeginTransaction

MSMQCoordinatedTransactionDispenser **MSMQTransactionDispenser**

The **BeginTransaction** method creates a new MSMQ transaction object. It returns an **MSMQTransaction** object that is used when sending and retrieving messages during the transaction.

Syntax

Set object1 = object2.**BeginTransaction**

Syntax Element	Description
object1	Transaction (**MSMQTransaction**) object that identifies the transaction.
object2	Transaction dispenser (**MSMQTransactionDispenser** or **MSMQCoordinatedTransactionDispenser**) object that creates the transaction.

Return Values

MSMQTransaction object that identifies the transaction.

Remarks

The **MSMQTransaction** object returned by **BeginTransaction** must be associated with all transaction queues and messages associated with this transaction. However, this does not mean that a transaction queue associated with this transaction cannot be used by other transactions. A single transaction queue can be associated with any number of transactions.

Example

This example starts a transaction and sends two messages.

```
Dim xdispenser as New MSMQCoordinatedTransactionDispenser
Dim xact as MSMQTransaction

Dim qSend as MSMQQueue
Dim msg1 as New MSMQMessage
Dim msg2 as New MSMQMessage

Set xact = xdispenser.BeginTransaction

'Assumes queue already exists and is transactional.
Set qSend = qinfo.Open(MQ_SEND_ACCESS, 0)
msg1.Label = "MyTransaction message"
msg1.Body = "Message 1 Body"
msg1.Send qSend, xact                    'Associates send with xact.

msg2.Label = "MyTransaction message"
msg2.Body = "Message 2 Body"
msg2.Send qSend, xact                    'Associates send with xact.

xact.Commit
```

See Also

MSMQTransaction

Close

MSMQQueue

The **Close** method closes this instance of the queue.

Syntax

object.**Close**

Syntax Element	Description
object	Queue (**MSMQQueue**) object that represents the open instance of the queue.

Remarks

When this method succeeds, MSMQ sets **Handle** to INVALID_HANDLE_VALUE.

The **MSMQQueue** object still exists after **Close** is called. Use the **MSMQQueue** object's **IsOpen** property to test if the queue is opened or closed.

Example

This example opens a queue for sending messages, then closes the queue. To try this example using Microsoft® Visual Basic® (version 5.0), paste the code into the Code window of a form, and then run the example and click the form.

```
Dim qinfo As MSMQQueueInfo
Dim q As MSMQQueue

Private Sub Form_Click()

  Set qinfo = New MSMQQueueInfo
  qinfo.PathName = ".\CloseTest"
  qinfo.Label = "Test Queue"
  qinfo.Create

  Set q = qinfo.Open(MQ_SEND_ACCESS, MQ_DENY_NONE)

  If q.IsOpen Then
     MsgBox "The " + qinfo.Label + " is open."
  Else
     MsgBox "The " + qinfo.Label + " is closed."
  End If

  q.Close

  If q.IsOpen Then
     MsgBox "The " + qinfo.Label + " is open."
  Else
     MsgBox "The " + qinfo.Label + " is closed."
  End If

End Sub
```

See Also

Create, **IsOpen**, **MSMQQueue**, **MSMQQueueInfo**, **Label**, **Open**, **PathName**

Commit

MSMQTransaction

The **Commit** method commits the operations requested by the transaction.

Syntax

object.**Commit** *(*[*fRetaining*][*, grfTC*][*, grfRM*]*)*

Syntax Element	Description
object	Transaction (**MSMQTransaction**) object that identifies the transaction.
fRetaining	Optional. Reserved by Microsoft DTC.
grfTC	Optional. Specifies if the commit is synchronous (default) or asynchronous. XACTTC_ASYNC: When specified, the call to commit returns as soon as the two-phase commit protocol is initiated. XACTTC_SYNCPHASEONE: When specified, the call to commit returns after phase one of the two-phase commit protocol.
grfRM	Optional. Reserved by Microsoft DTC.

Remarks

Calling **Commit** does not mean that the operations requested are performed. It only means that MSMQ guarantees that the operations will be performed as an atomic unit. (**Commit** is a wrapper for ITransaction::Commit. For information on ITransaction::Commit, see the Microsoft Platform SDK.)

An application must call **Commit** for the messages to be sent. If **Commit** is not called, the transaction will be terminated when the application exits.

For information on Microsoft® DTC, refer to the *Guide to Microsoft Distributed Transaction Coordinator* in the Microsoft Platform SDK.

Example

This example starts a transaction, sends two messages, and then terminates the transaction.

```
Dim xdispenser as New MSMQCoordinatedTransactionDispenser
Dim xact as MSMQTransaction

Dim qSend as MSMQQueue
Dim msg1 as New MSMQMessage
Dim msg2 as New MSMQMessage

'Begins transaction, opening an existing transaction queue.
Set xact = xdispenser.BeginTransaction
Set qSend = qinfo.Open(MQ_SEND_ACCESS, MQ_DENY_NONE)

'Sends message 1.
msg1.Label = "MyTransaction first message"
msg1.Body = "Message 1 Body"
msg1.Send qSend, xact                    'Associates send with xact.

'Sends message 2.
msg2.Label = "MyTransaction second message"
```

```
msg2.Body = "Message 2 Body"
msg2.Send qSend, xact                    'Associates send with xact.

'Commits transaction.
xact.Commit                              'Commits transaction.
```

Create

MSMQQueueInfo

The **Create** method produces a queue based on the queue properties of the **MSMQQueueInfo** object.

Syntax

object.**Create** ([*IsTransactional*][, *IsWorldReadable*])

Syntax Element	Description
object	Queue information (**MSMQQueueInfo**) object that defines queue.
IsTransactional	Optional (default is FALSE). BOOLEAN. When TRUE, indicates that the queue is a transaction queue. All messages sent to a transaction queue or read from a transaction queue must be done as part of a transaction.
IsWorldReadable	Optional (default is FALSE). BOOLEAN. When TRUE, anyone can read the messages in the queue and its queue journal. When FALSE, only the owner can read the messages.

Return Codes

For information on return codes, see MSMQ Error and Information Codes.

Remarks

You must always specify the queue's MSMQ pathname (**PathName**) before calling **Create**. The **PathName** property tells MSMQ where to store the queue's messages, if the queue is public or private, and the name of the queue.

Public queues are registered in MQIS, and private queues are registered on the local computer. All queues exist until deleted explicitly.

Private queues can only be created on the local computer. The applications responsibility to ensure that no other private queues with the same name exists on the local computer (if a queue with the same name already exists, MSMQ will return an MQ_ERROR_QUEUE_EXISTS error when the **MQCreateQueue** is called).

Setting other queue properties is optional. If a queue property is not set before **Create** is called, its default value is used when the queue is created. For a list of the queue properties that can be set by an application, see Creating a Queue.

To attach a journal to the queue, set **Journal** and **JournalQuota**. The journal keeps a copy of all messages retrieved from the queue. For information about journals, see Journal Queues.

To create a transactional queue, set the *IsTransactional* parameter to TRUE.

Typically queues are not created from **MSMQQueueInfo** objects found in a query because the queue already exists. However, you can create a new queue from an **MSMQQueueInfo** object found in a query if you delete the existing queue then change the **PathName** or **FormatName** property of the queue.

After the queue is created, the **MSMQQueueInfo** object can be opened multiple times. For example, the same **MSMQQueueInfo** object can be opened for sending messages to the queue and for reading the messages in the queue.

Access control can be changed by setting *isWorldReadable*. If this parameter is not set, its default value specifies MSMQ default security. Following are the default values for the security descriptor.

Default Value	Meaning
Owner	The process user.
Group	The process group.
DACL	Full control for the creator of the queue. All other processes can get queue properties, get queue security, and send messages to the queue.
SACL	None.

For information on access control, see Access Control.

Foreign public queues (queues located outside the MSMQ enterprise) are created in the same way as an MSMQ public queue. For foreign queues, the **PathName** property specifies the name of the foreign computer as it is defined in MQIS. For information on foreign computers, see MSMQ Connector Server.

When creating public queues, some clients may not see the new queue registered in the MSMQ information even though the queue was registered. Changes to MQIS (such as creating a public queue) must be propagated from site to site, which can cause delays in viewing the most current information. Consequently, clients in some sites may not be able to open the queue, even though it exists. Propagation delays, including communication network delays such as down links, are controlled by the MSMQ Administrator.

Example

This example creates a private queue on the local machine, then displays the queue's format name (the queue's format name is used to open the queue). To try this example using Microsoft Visual Basic (version 5.0), paste the code into the Code window of a form, and then run the example and click the form.

```
Dim qinfo As MSMQQueueInfo

Private Sub Form_Click()

  Set qinfo = New MSMQQueueInfo
  qinfo.PathName = ".\PRIVATE$\CreateTest"
  qinfo.Label = "Test Queue"
  qinfo.Create

  MsgBox "The queue's format name is: " + qinfo.FormatName

End Sub
```

See Also

IsTransactional, **FormatName**, **Journal**, **JournalQuota**, **Label**, **MSMQQueueInfo**, **PathName**

Delete

MSMQQueueInfo

The **Delete** method deletes an existing queue (it does not delete the **MSMQQueueInfo** object used to create the queue).

Syntax

object.**Delete**

Syntax Element	Description
object	The queue information (**MSMQQueueInfo**) object that defines the queue.

Return Codes

For information on return codes, see MSMQ Error and Information Codes.

Remarks

Deleting the queue does not delete the **MSMQQueueInfo** object, only the existing queue. You can still create a new queue based on the current properties of the **MSMQQueueInfo** object.

When deleting public queues, some clients may still see the queue registered in MQIS after the queue was deleted. Changes to MQIS (such as deleting a public queue) are propagated from site to site, which can cause delays in viewing the most current information. Consequently, clients in some sites may still try to send messages to the queue, even though it was deleted. Propagation delays, including communication network delays such as down links, are controlled by the MSMQ Administrator.

Example

This example assumes at least one queue whose label is "Test Queue" already exist. It runs a query for the test queues, asking if you want to delete each queue it finds.

To try this example using Microsoft Visual Basic (version 5.0), paste the code into the Code window of a form, and then run the example and click the form.

```
Dim query As New MSMQQuery
Dim qinfos As MSMQQueueInfos
Dim qinfo As MSMQQueueInfo
Dim Response As String


Private Sub Form_Click()

    Set qinfos = query.LookupQueue(Label:="Test Queue")
    qinfos.Reset
    Set qinfo = qinfos.Next

    While Not qinfo Is Nothing
        Response = MsgBox("Delete queue: " + qinfo.Label, vbYesNo)

        If Response = vbYes Then
            qinfo.Delete
        End If

        Set qinfo = qinfos.Next
    Wend
```

```
End Sub
```

See Also

Label, **LookupQueue**, **MSMQQuery**, **MSMQQueueInfos**, **MSMQQueueInfo**, **Next**, **Reset**

EnableNotification

MSMQQueue

The **EnableNotification** method starts event notification for asynchronously reading messages in the queue. Once this method is called, applications can asynchronously peek at messages or retrieve them from the queue in a user-defined event handler.

Syntax

object.**EnableNotification** (*Event*[*, Cursor*][*,ReceiveTimeout*])

Syntax Element	Description
object	Queue (**MSMQQueue**) object that represents the open instance of the queue.
Event	References an **MSMQEvent** object.
Cursor	Optional. Specifies the action of the cursor. MQMSG_FIRST: Default. Notification starts when a message is in the queue. MQMSG_CURRENT: Notification starts when a message is at the current location of the cursor. MQMSG_NEXT: The cursor is moved, then notification starts when a message is at the new cursor location.
ReceiveTimeout	Optional. Specifies how long (in milliseconds) MSMQ waits for a message to arrive.

Return Codes

For information on return codes, see MSMQ Error and Information Codes.

Remarks

When **EnableNotification** is called, events are triggered when a message is found at the position specified by *Cursor*. When the default setting is used, events are triggered when a message is in the queue.

EnableNotification fires a single **Arrived** event when it finds a message. To read more messages, **EnableNotification** must be explicitly called again from within the event handler.

The **Arrived** event is triggered on the **MSMQEvent** object passed in *Event.* The **MSMQEvent_Arrived** event handler (typically implemented by the user) is passed a reference to the queue where the message arrived.

Note In Microsoft® Visual Basic®, the application's event handler can still be invoked even if the form it is on has been unloaded by the application. When the event handler is fired, Visual Basic will reload the form automatically if any of its members (for example, a message box on the form) are referred to.

When an **Arrived** event is triggered, there is no guarantee that the message that triggered the event will still be available when the application tries to use the message. All queues are dynamic, and other clients may remove the arrived message before it can be used. It is up to the application to determine if the message is there before attempting to peek at the message or retrieve it. (If the queue is not being shared, it is safe to assume that the message is still there.)

Invoking **EnableNotification** with *ReceiveTimeout* set to 0 is similar to synchronously calling **Peek**.

Receive errors (such as timeout errors) trigger an **ArrivedError** event on the associated **MSMQEvent**

object.

LookupQueue

MSMQQuery

The **LookupQueue** method returns a collection of public queues based on the following queue properties: queue identifier, service type, label, create time, and modify time.

Syntax

*object.***LookupQueue** *(*[*QueueGuid*] [, *ServiceTypeGuid*] [, *Label*] [, *CreateTime*] [, *ModifyTime*] [, *RelServiceType*] [, *RelLabel*] [, *RelCreateTime*] [, *RelModifyTime*]*)*

Syntax Element	Description
object	Query.
QueueGuid (String)	Optional. Identifier of queue.
ServiceTypeGuid (String)	Optional. Type of service provided by the queue. See also *RelServiceType*.
Label (String)	Optional. Label of queue. See also *RelLabel*.
CreateTime (Variant Date)	Optional. Time when queue was created. See also *RelCreateTime*.
ModifyTime (Variant Date)	Optional. Time when queue properties were last set (both when the queue was created and the last time **Update** was called). See also *RelModifyTime*.
RelServiceType (Long)	Optional (default is REL_EQ). Relationship parameter for *ServiceTypeGuid.*
RelLabel (Long)	Optional (default is REL_EQ). Relationship parameter for *Label*.
RelCreateTime (Long)	Optional (default is REL_EQ). Relationship parameter for *CreateTime*.
RelModifyTime (Long)	Optional (default is REL_EQ). Relationship parameter for *ModifyTime*.

Return Values

MSMQQueueInfos object.

Return Codes

For information on return codes, see MSMQ Error and Information Codes.

Remarks

LookupQueue returns a single **MSMQQueueInfos** object that represents a set of **MSMQQueueInfo** objects that each contain information describing a single MSMQ queue.

The relationship parameters *RelServiceType, RelLabel, RelCreateTime* and *RelModifyTime* provide simple Boolean comparison operators that can be used in conjunction with their respective lookup parameter. These comparison operators include: REL_EQ, REL_NEQ, REL_LT, REL_GT, REL_LE, REL_GE, and REL_NOP (REL_NOP indicates that the associated lookup parameter should be ignored). These operators give you greater control over which queues are returned from to the query.

The *CreateTime* and *ModifyTime* parameters return both the date and time. Consequently, using the REL_EQ value for *RelCreateTime* or *RelModifyTime* may not prove very useful.

When running a query, MSMQ can locate queues faster when the query is based on *QueueGuid*, *ServiceTypeGuid*, or *Label (RelLabel = REL_EQ)*. The query runs faster because these properties are indexed in MQIS, providing a faster way for MSMQ to locate the property specified in the call.

LookupQueue can only return queues that are in MQIS when **LookupQueue** is called. Queues created after **LookupQueue** is called are not included.

To open a specific queue from the query, use the **MSMQQueueInfos** object's **Reset** and **Next** methods to locate the queue. **Reset** points the cursor to the front of the query results (not the first queue in the query) and **Next** points to the next queue. Once the queue is located, use the **MSMQQueueInfo** object's **Open** method to open the queue.

Example

This example assumes that at least one queue with the label "Test Queue" already exists. A query is run for the test queues, displaying the format name of each queue it finds.

To try this example using Microsoft® Visual Basic® (version 5.0), make sure the Microsoft Message Queue Object Library is referenced, then paste the following code into the code window of a form, run the example, then click on the form.

```
Dim query As New MSMQQuery
Dim qinfos As MSMQQueueInfos
Dim qinfo As MSMQQueueInfo

Private Sub Form_Click()

    Set qinfos = query.LookupQueue(Label:="Test Queue")
    qinfos.Reset
    Set qinfo = qinfos.Next

    While Not qinfo Is Nothing
        MsgBox "I found a Test Queue! its Format name is: " + qinfo.FormatName
        Set qinfo = qinfos.Next
    Wend

End Sub
```

See Also

FormatName, **Label**, **MSMQQuery**, **MSMQQueueInfo**, **MSMQQueueInfos**, **Next**, **Reset**

Next

MSMQQueueInfos

The **Next** method returns the next queue in the collection.

Syntax

object.**Next**

Syntax Element	Description
object	Collection that contains the queues.

Return Values

MSMQQueueInfo object.

Return Codes

For information on return codes, see MSMQ Error and Information Codes.

Remarks

When the cursor is pointing to the front of the query results (for example, **Reset** has been called), call **Next** to point to the first queue in the query.

Returns NULL if the cursor is at the end of the collection (EOL).

Example

This example assumes that at least one queue whose label is "Test Queue" already exists. It runs a query for the test queues, displaying the format name of each queue it finds.

To try this example using Microsoft® Visual Basic® (version 5.0), make sure the Microsoft Message Queue Object Library is referenced, then paste the following code into the code window of a form, run the example, then click on the form.

```
Dim query As New MSMQQuery
Dim qinfos As MSMQQueueInfos
Dim qinfo As MSMQQueueInfo

Private Sub Form_Click()

    Set qinfos = query.LookupQueue(Label:="Test Queue")
    qinfos.Reset
    Set qinfo = qinfos.Next

    While Not qinfo Is Nothing
        MsgBox "I found a Test Queue! its Format name is: " + qinfo.FormatName
        Set qinfo = qinfos.Next
    Wend

End Sub
```

See Also

FormatName, **Label**, **MSMQQuery**, **MSMQQueueInfo**, **MSMQQueueInfos**, **Next**, **Reset**

MachineIdOfMachineName

MSMQApplication

The **MachineIdOfMachineName** method returns a machine identifier for a specified computer. The machine identifier can be used to create the format name of a machine journal.

Syntax

machineId=MachineIdOfMachineName("MachineName")As String

Return Codes

For information on return codes, see MSMQ Error and Information Codes.

Open

MSMQQueueInfo

The **Open** method opens a queue for sending messages to the queue, for reading messages in the queue (peek and receive access), or for looking at the queue's information.

The properties of the queue are based on the current properties of the queue information (**MSMQQueueInfo**) object.

Syntax

set object2 = object1.**Open** *(Access, ShareMode)*

Syntax Element	Description
object1	Queue information (**MSMQQueueInfo**) object that defines the queue.
object2	Queue (**MSMQQueue**) object that represent the open instance of the queue.
Access	Specifies how the application accesses the queue (peek, send, or receive). This setting cannot be changed while the queue is open. *Access* can be set to one of the following: MQ_PEEK_ACCESS: Messages can only be looked at. They cannot be removed from the queue. MQ_SEND_ACCESS: Messages can only be sent to the queue. MQ_RECEIVE_ACCESS: Messages can be retrieved from the queue or peeked at. See *ShareMode* for limiting who can retrieve messages.
ShareMode	Specifies who can access the queue. Set to one of the following: MQ_DENY_NONE: Default. The queue is available to everyone. This setting must be used if *Access* is set to MQ_PEEK_ACCESS or MQ_SEND_ACCESS. MQ_DENY_RECEIVE_SHARE: Limits those who can retrieve messages from the queue to this process. If the queue is already opened for retrieving messages by another process, this call fails and returns MQ_ERROR_SHARING_VIOLATION. Applicable only when *Access* is set to MQ_RECEIVE_ACCESS.

Return Codes

For information on return codes, see MSMQ Error and Information Codes.

Remarks

Open returns an **MSMQQueue** object each time it is called. Consequently, the **MSMQQueueInfo** object that opens the queue can be associated with any number of **MSMQQueue** objects.

When the queue is opened, MQIS is automatically updated with the current property values of the **MSMQQueueInfo** object.

When a queue is opened, the cursor points to the front of the queue, not the first message in the queue. For information on how to move the cursor to the first message in the queue, see

MSMQQueue.

Direct format names can only be used when sending messages to a queue. A direct format name tells MSMQ not to use MQIS (for public queues) or the local computer (for private queues) to get routing information. When a direct format name is used, all routing information is taken from the format name and MSMQ sends the messages to the queue in a single hop.

Setting *ShareMode* to MQ_DENY_RECEIVE_SHARE means that until the calling application calls the **MSMQQueue** object's **Close** method, no other MSMQ applications can read the messages in the queue. This includes applications that may have the correct access rights to read messages from the queue.

If the calling application does not have sufficient access rights to a queue, two things can happen. If *Access* is set to MQ_PEEK_ACCESS or MQ_RECEIVE_ACCESS, **Open** will fail and throw MQ_ERROR_ACCESS_DENIED. If *Access* is set to MQ_ACCESS_SEND, **Open** will succeed, but errors will be thrown when the application tries to send a message.

To change the access rights of the queue, call **MQSetQueueSecurity**. The following table lists the access right needed to open the queue in peek, send, or receive access mode.

Queue Access Mode	Queue Access Right
MQ_PEEK_MESSAGE	MQSEC_PEEK_MESSAGE
MQ_SEND_MESSAGE	MQSEC_WRITE_MESSAGE
MQ_RECEIVE_MESSAGE	MQSEC_RECEIVE_MESSAGE

After opening a queue, there is no provision to change the access mode of the queue. Either close and open the queue in the current thread, or open a new thread with new access rights.

When the queue is opened, MQIS is automatically updated with the current property values of the **MSMQQueueInfo** object.

When a queue is opened, the cursor points to the front of the queue, not the first message in the queue. For information on how to move the cursor to the first message in the queue, see Reading Messages In a Queue.

Journal queues and dead letter queues can only be opened with *Access* set to MQ_PEEK_ACCESS or MQ_RECEIVE_ACCESS. You cannot send messages to a journal queue.

When opening a queue on a remote computer, MSMQ does not check for the existence of the queue when *Access* is set to MQ_SEND_ACCESS. In addition, if *Access* is set to MQ_RECEIVE_ACCESS, the computer opening the queue must support the same protocol as the remote computer where the queue is located.

There is one exception, however. Foreign queues (a queue located outside the MSMQ enterprise) are opened the same way queues located within the enterprise are opened. Applications cannot open a foreign queue using a direct format name. MSMQ needs the routing information stored in MQIS to find a MSMQ Connector Server for the foreign queue.

Example

This example creates a public queue, then opens the queue for sending messages. To try this example using Microsoft® Visual Basic® (version 5.0), paste the code into the Code window of a form, and then run the example and click the form.

```
Dim qinfo As MSMQQueueInfo
Dim q As MSMQQueue

Private Sub Form_Click()

  Set qinfo = New MSMQQueueInfo
  qinfo.PathName = ".\OpenTest"
```

```
  qinfo.Label = "Test Queue"
  qinfo.Create

  Set q = qinfo.Open(MQ_SEND_ACCESS, MQ_DENY_NONE)
  If q.IsOpen Then
     MsgBox "The queue " + qinfo.PathName + " is open."
     Else
     MsgBox "The queue is not open!"
  End If

End Sub
```

See Also

Create, **Label**, **MSMQQueue**, **MSMQQueueInfo**, **PathName**

Peek

MSMQQueue

The **Peek** method returns the first message in the queue, or waits for a message to arrive if the queue is empty. It does not remove the message from the queue.

Syntax

set object2 = object1.**Peek** ([*ReceiveTimeout*][, *WantDestinationQueue*][, *WantBody*])

Syntax Element	Description
object1	Queue (**MSMQQueue**) object that represents the queue where the message resides.
object2	Message (**MSMQMessage**) object that represents message read from the queue.
ReceiveTimeout	Optional. Specifies how long (in milliseconds) MSMQ waits for a message to arrive.
WantDestinationQueue	Optional (default is FALSE). If TRUE, **DestinationQueueInfo** is updated when the message is read from the queue. Setting this property to TRUE may slow down the operation of the application.
WantBody	Optional (default is TRUE). If the Body of the message is not needed, set this property to FALSE to optimize the speed of the application.

Return Values

MSMQMessage object.

Remarks

The **Peek** method always looks at the first message in the queue. It is completely independent of the implied cursor used by PeekCurrent, PeekNext, and ReceiveCurrent.

Applications can peek at messages in queues opened with peek or receive access (see **Open**).

The *ReceiveTimeout* parameter can be used to control how long MSMQ waits for a message to arrive when the queue is empty.

The *WantDestinationQueue* and *WantBody* parameters can be used to optimize the speed of the application.

Example

This example sends a message to the destination queue to make sure at least one message is there, then reads the first message in the queue. If another message is already in the queue, the message returned by Peek may not be the message sent by the example.

To try this example using Microsoft Visual Basic (version 5.0), paste the code into the Code window of a form, and then run the example and click the form.

```
Option Explicit

Dim query As New MSMQQuery
Dim qinfoDest As MSMQQueueInfo
Dim qDest As MSMQQueue
Dim msgSent As New MSMQMessage
Dim msgDest As MSMQMessage
```

```
Private Sub Form_Click()

    '**************************
    ' Locate destination queue
    '(create one if one doesn't
    ' exist).
    '**************************
    Set qinfos = query.LookupQueue(Label:="Destination Queue")
    qinfos.Reset
    Set qinfoDest = qinfos.Next
    If qinfoDest Is Nothing Then
        Set qinfoDest = New MSMQQueueInfo
        qinfoDest.PathName = ".\DestQueue"
        qinfoDest.Label = "Destination Queue"
        qinfoDest.Create
    End If

    '**************
    ' Open destination queue.
    '**************
    Set qDest = qinfoDest.Open(MQ_SEND_ACCESS, MQ_DENY_NONE)

    '**************
    ' Send Message.
    '**************
    msgSent.Label = "Test Message"
    msgSent.Body = "This message is used to test reading messages."

    msgSent.Send qDest
    qDest.Close

    '***********************
    ' PEEK at first message
    ' in the queue.
    '***********************

    Set qDest = qinfoDest.Open(MQ_PEEK_ACCESS, MQ_DENY_NONE)

    On Error GoTo Handler
    Set msgDest = qDest.Peek(ReceiveTimeout:=100)
    MsgBox "The first message in the queue is: " + msgDest.Label
    Exit Sub

    '***************
    ' Error Handler
    '***************

Handler:
    'Test for errors.

End Sub
```

See Also

Body, **Close**, **Create**, **Label**, **LookupQueue**, **MSMQMessage**, **MSMQQuery**, **MSMQQueue**, **MSMQQueueInfo**, **MSMQQueueInfos**, **Open**, **PathName**, **Reset**, **Send**

PeekCurrent

MSMQQueue

The **PeekCurrent** method returns the current message, depending on the position of the implied cursor.

When **PeekCurrent** is called, execution is stopped until the message is read from the queue or the receive timeout timer (*ReceiveTimeout*) expires.

Syntax

set object2 = object1.**PeekCurrent** ([*ReceiveTimeout*][, *WantDestinationQueue*][, *WantBody*])

Syntax Element	Description
object1	Queue (**MSMQQueue**) object that represents the queue where the message resides.
object2	Message (**MSMQMessage**) object that represents message read from the queue.
ReceiveTimeout	Optional. Specifies how long (in milliseconds) MSMQ waits for a message to arrive when the queue is empty or the cursor is pointing at the end of the queue.
WantDestinationQueue	Optional (default is FALSE). If TRUE, **DestinationQueueInfo** is updated when the message is read from the queue. Setting this property to TRUE may slow down the operation of the application.
WantBody	Optional (default is TRUE). If the Body of the message is not needed, set this property to FALSE to optimize the speed of the application.

Return Values

MSMQMessage object.

PeekNext

MSMQQueue

The **PeekNext** method returns the next message in the queue, but does not remove it from the queue.

When **PeekNext** is called, execution is stopped until the message is read from the queue or the receive timeout timer (*ReceiveTimeout*) expires.

Syntax

set object2 = object1.**PeekNext** ([*ReceiveTimeout*][, *WantDestinationQueue*][*, WantBody*])

Syntax Element	Description
object1	Queue (**MSMQQueue**) object that represents the queue where the message resides.
object2	Message (**MSMQMessage**) object that represents message read from the queue.
ReceiveTimeout	Optional. Specifies how long (in milliseconds) MSMQ waits for a message to arrive when the cursor is pointing at the end of the queue.
WantDestinationQueue	Optional (default is FALSE). If TRUE, **DestinationQueueInfo** is updated when the message is read from the queue. Setting this property to TRUE may slow down the operation of the application.
WantBody	Optional (default is TRUE). If the Body of the message is not needed, set this property to FALSE to optimize the speed of the application.

Return Values

MSMQMessage object.

Remarks

PeekNext moves the cursor first, then looks at the message at the new location.

Before **PeekNext** is called, **Peek Current** must be called to initialize the implied cursor. If PeekCurrent is not called, MQ_ERROR_ILLEGAL CURSOR_ACTION is returned.

The *WantDestinationQueue* and *WantBody* parameters can be used to optimize the speed of the application.

The queue where the message resides can be opened with PEEK or RECEIVE access.

Receive

MSMQQueue

The **Receive** method retrieves the first message in the queue, removing the message from the queue when the message is read.

Syntax

set object2 = object1.**Receive** ([*pTransaction*] [, *WantDestinationQueue*] [, *WantBody*] [, *ReceiveTimeout*])

Syntax Element	Description
object1	Queue (**MSMQQueue**) object that represents queue where the message resides.
object2	Message (**MSMQMessage**) object that represents message retrieved from the queue.
pTransaction	Optional. An **MSMQTransaction** object or one of the following constants. Constants include: MQ_NO_TRANSACTION: Specifies that the call is not part of a transaction. MQ_MTS_TRANSACTION: Default. Specifies that the call is part of the current MTS (Microsoft Transaction Server) transaction. MQ_XA_TRANSACTION: Specifies that the call is part of an externally coordinated, XA-compliant transaction.
WantDestinationQueue	Optional (default is FALSE). If TRUE, **DestinationQueueInfo** is updated when the message is read from the queue. Setting this property to TRUE may slow down the operation of the application.
WantBody	Optional (default is TRUE). If the Body of the message is not needed, set this property to FALSE to optimize the speed of the application.
ReceiveTimeout	Optional (default is INFINITE). Sets the message's *timeout* timer. Specifies how long (in milliseconds) MSMQ waits for a message to arrive.

Return Values

MSMQMessage object that represents the first message in the queue.

Remarks

To retrieve messages from the queue, the queue must be opened with receive access (*Access* = MQ_RECEIVE_ACCESS). Messages retrieved from a queue are also removed from the queue.

The *ReceiveTimeout* parameter is optional. However, if it is not specified the default value of *ReceiveTimeout* (INFINITE) will force the **Receive** call to block execution until a message arrives.

The *pTransaction* parameter can be set to an **MSMQTransaction** object, or one of the constants described above. For information on the different types of transactions that MSMQ supports, see:

- MSMQ Internal Transactions

- MS DTC External Transactions
- MTS Transactions
- XA-Compliant Transactions

The *WantDestinationQueue* and *WantBody* parameters can be used to optimize the speed of the application.

For synchronous calls, execution is stopped until a message is retrieved from the queue or the message's *timeout* timer expires. For an example of reading messages synchronously, see Reading Messages Synchronously.

For an example of reading messages asynchronously, see Reading Messages Asynchronously.

To read messages in a queue without removing them, see the **Peek**, **PeekCurrent**, or **PeekNext**.

Example

This example locates a destination queue, sends a message to the queue to make sure at least one message is in the queue, then removes all the messages from the queue. An error handler is added to trap any errors generated as a result of the Receive call.

To try this example using Microsoft® Visual Basic® (version 5.0), paste the code into the Code window of a form, and then run the example and click the form.

```
Option Explicit
Dim query As New MSMQQuery
Dim qinfos As New MSMQQueueInfos
Dim qinfoDest As MSMQQueueInfo
Dim qDest As MSMQQueue
Dim msgSent As New MSMQMessage
Dim msgDest As MSMQMessage

Private Sub Form_Click()

    '****************************
    ' Locate destination queue
    '(create one if one doesn't
    ' exist).
    '****************************
    Set qinfos = query.LookupQueue(Label:="Destination Queue")
    qinfos.Reset
    Set qinfoDest = qinfos.Next
    If qinfoDest Is Nothing Then
        Set qinfoDest = New MSMQQueueInfo
        qinfoDest.PathName = ".\DestQueue"
        qinfoDest.Label = "Destination Queue"
        qinfoDest.Create
    End If

    '***************
    ' Open destination queue.
    '***************
    Set qDest = qinfoDest.Open(MQ_SEND_ACCESS, MQ_DENY_NONE)

    '***************
    ' Send Message.
    '***************
    msgSent.Label = "Test Message A"
```

```
    msgSent.Body = "This message is for testing how messages are read from the queue."

    msgSent.Send qDest
    qDest.Close

    '**********************
    ' Removes all messages
    ' in the queue.
    '**********************

    Set qDest = qinfoDest.Open(MQ_RECEIVE_ACCESS, MQ_DENY_NONE)

    On Error GoTo Handler

    Do While True
        Set msgDest = qDest.Receive(ReceiveTimeout:=1000)
        If msgDest Is Nothing Then Exit Do
        MsgBox msgDest.Label + " is removed from the queue."
    Loop

    Exit Sub

    '***************
    ' Error Handler
    '***************

Handler:
    If (Err = MQ_ERROR_IO_TIMEOUT) Then
        MsgBox "All messages are removed from the queue."
        Exit Sub
    Else
        MsgBox "Unexpected error!"
    End If

End Sub
```

See Also

Body, **Close**, **Create**, **Label**, **LookupQueue**, **MSMQMessage**, **MSMQQuery**, **MSMQQueue**, **MSMQQueueInfo**, **MSMQQueueInfos**, **Next**, **Open**, **PathName**, **Reset**, **Send**

ReceiveCurrent

MSMQQueue

The **ReceiveCurrent** method reads the message at the current cursor location.

Syntax

set object2 = object1.**ReceiveCurrent** ([*ReceiveTimeout*][, *pTransaction*][, *WantDestinationQueue*][, *WantBody*])

Syntax Element	Description
object1	Queue (**MSMQQueue**) object that represents the open instance of the queue that is receiving messages.
object2	Message (**MSMQMessage**) object that represents message retrieved from the queue.
ReceiveTimeout	Optional. Specifies how long (in milliseconds) MSMQ waits for a message to arrive.
pTransaction	Optional. An **MSMQTransaction** object or one of the following constants. Constants include: MQ_NO_TRANSACTION: Specifies that the call is not part of a transaction. MQ_MTS_TRANSACTION: Default. Specifies that the call is part of the current MTS (Microsoft Transaction Server) transaction. MQ_XA_TRANSACTION: Specifies that the call is part of an externally coordinated, XA-compliant transaction.
WantDestinationQueue	Optional (default is FALSE). If TRUE, **DestinationQueueInfo** is updated when the message is read from the queue. Setting this property to TRUE may slow down the operation of the application.
WantBody	Optional (default is TRUE). If the Body of the message is not needed, set this property to FALSE to optimize the speed of the application.

Return Values

MSMQMessage object.

Remarks

The *pTransaction* parameter can be set to an **MSMQTransaction** object, or one of the constants described above. For information on the different types of transactions the MSMQ supports, see:

- MSMQ Internal Transactions
- MS DTC External Transactions
- MTS Transactions
- XA-Compliant Transactions

Refresh

MSMQQueueInfo

The **Refresh** method refreshes the property values of the **MSMQQueueInfo** object. These values are retrieved from MQIS (public queues) or from the local computer (private queues).

Syntax

object.**Refresh**

Syntax Element	Description
object	The queue information (**MSMQQueueInfo**) object that defines the queue.

Remarks

All queue properties can be retrieved, however, you can only retrieve the properties of private queues if they are located on your local computer.

Refresh is typically used when more than one user is using the queue. For example, if user 1 locates the queue and then user 2 modifies the queue's properties, user 1 needs to call **Refresh** to sync up with user 2's changes.

After a queue is created, the properties of the **MSMQQueueInfo** object are not updated until **Refresh** is explicitly called, or the queue is closed and reopened. For example, even though MSMQ sets **CreateTime**, **ModifyTime**, or QueueGuid when it creates the queue, the application must call **Refresh** to update the properties of the **MSMQQueueInfo** object before it can read those properties.

For a complete discussion on retrieving a queue's properties, see Retrieving a Queue's Properties Using ActiveX Components.

Refresh uses the **MQGetQueueProperties** function.

Example

This example creates a public queue, then uses Refresh to update the MSMQQueueInfo so it can display the queue's identifier (QueueGuid). To try this example using Microsoft Visual Basic (version 5.0), paste the code into the Code window of a form that has a single text box, and then run the example and click the form.

```
Dim qinfo As MSMQQueueInfo

Private Sub Form_Click()

  Set qinfo = New MSMQQueueInfo
  qinfo.PathName = ".\RefreshTest"
  qinfo.Label = "Test Queue"
  qinfo.Create

  qinfo.Refresh             'Required to update QueueGuid
  Text1.Text = "quidQueue = " + CStr(qinfo.QueueGuid)

End Sub
```

See Also

Create,**Label**, **MSMQQueueInfo**, **PathName**, **QueueGuid**

Reset

MSMQQueue **MSMQQueueInfos**

The **Reset** method returns the cursor to the start of the results of a query, or to the start of a queue.

Syntax

object.**Reset**

Syntax Element	Description
object	Collection (**MSMQQuery**) object that represents a collection of queues, or queue (**MSMQQueue**) object that represents an open instance of a queue.

Return Codes

For information on return codes, see MSMQ Error and Information Codes.

Remarks

Next Queue

When looking at the results of a query, the **MSMQQueueInfo** object's **Reset** method moves the cursor to the start of the query. To move to the first queue of the query, call **Next**.

Next Message

When looking at the messages in a queue, the **MSMQQueue** object's **Reset** method moves the cursor to the start of the queue. To move to the first message in the queue, call **PeekCurrent** or **ReceiveCurrent**.

Example: Pointing to the first queue

This example assumes that at least one queue whose label is "Test Queue" already exist. It runs a query for the test queues, then displays the format name of the first queue it found.

To try this example using Microsoft® Visual Basic® (version 5.0), make sure the Microsoft Message Queue Object Library is referenced, then paste the following code into the code window of a form, run the example, then click on the form.

```
Dim query As New MSMQQuery
Dim qinfos As MSMQQueueInfos
Dim qinfo As MSMQQueueInfo

Private Sub Form_Click()

    Set qinfos = query.LookupQueue(Label:="Test Queue")
    qinfos.Reset
    Set qinfo = qinfos.Next
    MsgBox "I found a Test Queue! its Format name is: " + FormatName

End Sub
```

Example: Pointing to the first message

This example assumes that a queue exists, opens the queue for receiving messages, looks at the label of each message in the queue, and then resets the cursor to the start of the queue.

To try this example using Microsoft® Visual Basic® (version 5.0), make sure the Microsoft Message Queue Object Library is referenced, then paste the following code into the code window of a form, run

the example, then click on the form.

```
Dim qinfoMyQueue as MSMQQueueInfo
Dim qMyInputQueue as MSMQQueue
Dim msgMyMessage as MSMQMessage

'Open queue as an input queue (set Access to MQ_RECEIVE_ACCESS).
Set qMyInputQueue = qinfoMyQueue.Open (Access := MQ_RECEIVE_ACCESS,
ShareMode := MQ_DENY_NONE)
Set msgMyMessage = qMyInputQueue.PeekCurrent

'Traverse queue.
While not msgMyMessage is Nothing
        set msgMyMessage = qMyInputQueue.PeekNext
        msgbox msgMyMessage.Label
Wend

qMyInputQueue.Reset                    'Points to start of queue.
```

See Also

FormatName, **Label**, **MSMQQuery**, **MSMQQueueInfo**, **MSMQQueueInfos**, **Next**, **Reset**

Send

MSMQMessage

The **Send** method sends a message to the specified queue.

Syntax

object.**Send** *(DestinationQueue,* [*pTransaction*]*)*

Syntax Element	Description
object	Message to be sent.
DestinationQueue	Reference to destination queue object.
pTransaction	Optional. An **MSMQTransaction** object or one of the following constants. Constants include: MQ_NO_TRANSACTION: Specifies that the call is not part of a transaction. MQ_MTS_TRANSACTION: Default. Specifies that the call is part of the current MTS (Microsoft Transaction Server) transaction. MQ_SINGLE_MESSAGE: Sends a single message as a transaction. MQ_XA_TRANSACTION: Specifies that the call is part of an externally coordinated, XA-compliant transaction.

Remarks

To get the handle of a queue, use the **Handle** property of the queue.

To save a copy of the message in a machine journal, set **Journal** to MQMSG_JOURNAL. For information on machine journals, see Journal Queues.

Messages that do not reach their destination can be sent to a dead letter queue by setting the messages **Journal** property to MQMSG_DEADLETTER (transaction messages are automatically sent to the transaction dead letter queue if the transaction is not successful). For information on dead letter queues, see Dead Letter Queues.

The *pTransaction* parameter can be set to an **MSMQTransaction** object, or one of the constants described above. For information on the different types of transactions the MSMQ supports, see:

- MSMQ Internal Transactions
- MS DTC External Transactions
- MTS Transactions
- XA-Compliant Transactions

Example

This example sends a message to a queue that it creates. To try this example using Microsoft Visual Basic (version 5.0), paste the code into the Code window of a form, and then run the example and click the form.

```
Dim qinfo As MSMQQueueInfo
Dim q As MSMQQueue
Dim msg As New MSMQMessage
```

```
Private Sub Form_Click()
    '***************
    ' Create queue.
    '***************
   Set qinfo = New MSMQQueueInfo
   qinfo.PathName = ".\SendTest"
   qinfo.Label = "Test Queue"
   qinfo.Create
    '***************
    ' Open queue.
    '***************
   Set q = qinfo.Open(MQ_SEND_ACCESS, MQ_DENY_NONE)
    '***************
    ' Send Message.
    '***************
   msg.Label = "Test Message"
   msg.Body = "This is a test message with a string Body."
   msg.Send q


 q.Close

End Sub
```

See Also

Body, **Close**, **Create**, **Label**, **MSMQMessage**, **MSMQQueue**, **MSMQQueueInfo**, **Open**, **PathName**

Update

MSMQQueueInfo

The **Update** method updates MQIS (public queues) or the local computer (private queues) with the current property values of the **MSMQQueueInfo** object.

Syntax

object.**Update**

Syntax Element	Description
object	The queue information (**MSMQQueueInfo**) object that defines the public queue.

Remarks

The **Update** method can only update existing queues. It cannot be called on an **MSMQQueueInfo** object before the queue is created or after the queue is deleted.

MQIS (public queues) and the local computer (private queues) are not updated when an **MSMQQueueInfo** property is set. They are only updated when a queue is created, when the queue is opened, and whenever **Update** is explicitly called.

Update can be used on **MSMQQueueInfo** objects that define public queues or local private queues. Properties for remote private queues are stored on the computer where the queue exists, and therefore cannot be updated.

The following queue properties may not be available to **Update**.

Property Name	Reason
BasePriority	For public queues only. Cannot be set for private queues.
CreateTime	Set by MSMQ.
QueueGuid	Set by MSMQ.
ModifyTime	Set by MSMQ.
PathName	Can only be set when the queue is created.

For a complete description of setting queue properties, including a list of the properties that can be set, see Setting a Queue's Properties Using ActiveX Components.

When setting the properties for public queues, some clients may see the old settings registered in MQIS. Changes to MQIS (such as setting queue properties) are propagated from site to site, which can cause delays in viewing the most current information. Consequently, clients in some sites may still see old settings, even though they were changed by **Update**. Propagation delays, including communication network delays such as down links, are controlled by the MSMQ Administrator.

Example

This example creates a public queue and then uses **Update** to change the queue's label. To try this example using Microsoft Visual Basic (version 5.0), paste the code into the Code window of a form that has a single text box, and then run the example and click the form.

```
Dim qinfo As MSMQQueueInfo

Private Sub Form_Click()

  Set qinfo = New MSMQQueueInfo
```

```
    qinfo.PathName = ".\UpdateTest"
    qinfo.Label = "Test Queue"
    qinfo.Create
    MsgBox "The queue's label is: " + qinfo.Label

    qinfo.Label = "New Queue Label"
    qinfo.Update
    MsgBox "The queue's label is: " + qinfo.Label

End Sub
```

See Also

Create, **Label**, **MSMQQueueInfo**, **PathName**

ActiveX Properties

These are the properties associated with the ActiveX components provided by MSMQ.

The properties of the MSMQ objects include:

MSMQQueueInfo

Authenticate

BasePriority

CreateTime

FormatName

IsTransactional

IsWorldReadable

Journal

JournalQuota

Label

ModifyTime

Pathname

PrivLevel

QueueGuid

Quota

ServiceTypeGuid

MSMQQueue Properties

Access

Handle

IsOpen

QueueInfo

ShareMode

MSMQMessage Properties

Ack

AdminQueueInfo

AppSpecific

ArrivedTime

AuthLevel

Body

BodyLength

Class

CorrelationId

Delivery

DestinationQueueInfo

EncryptAlgorithm

HashAlgorithm

Id

IsAuthenticated

Journal

Label

MaxTimeToReachQueue

MaxTimeToReceive

Priority

PrivLevel

ResponseQueueInfo

SenderCertificate

SenderID

SenderIDType

SentTime

SourceMachineGuid

Trace

MSMQTransaction Properties

Transaction

Access

MSMQQueue

Read-only. The **Access** property indicates the access rights of the queue.

Quick Info

Type:	**Long**
Run time:	read-only

Syntax

object.**Access**

Syntax Element	Description
object	Queue (**MSMQQueue**) object that represents an instance of the queue.

Return Values

The **Access** property returns one of the following values:

MQ_SEND_ACCESS
Messages can only be sent to the queue.

MQ_PEEK_ACCESS
Messages can only be looked at. They cannot be removed from the queue.

MQ_RECEIVE_ACCESS
Messages can be taken out of the queue or peeked at.

Remarks

The **Access** property returns the access rights of the queue when it was last opened, regardless if the queue is currently open or closed.

Example

This example opens a queue for sending messages, then uses the value of **Access** to test how the queue was opened (with what access rights). To try this example using Microsoft Visual Basic (version 5.0), paste the code into the Code window of a form, and then run the example and click the form.

```
Dim qinfo As MSMQQueueInfo
Dim q As MSMQQueue

Private Sub Form_Click()

  Set qinfo = New MSMQQueueInfo
  qinfo.PathName = ".\lAccessTest"
  qinfo.Label = "Test Queue"
  qinfo.Create

  Set q = qinfo.Open(MQ_SEND_ACCESS, MQ_DENY_NONE)

  Select Case q.Access
     Case MQ_SEND_ACCESS
          MsgBox "The queue is open for sending messages."
     Case MQ_RECEIVE_ACCESS
          MsgBox "The queue is open for retrieve messages."
     Case MQ_PEEK_ACCESS
```

```
            MsgBox "The queue is open to peek at messages."
        Case Else
            MsgBox "Not a valid return value!"
    End Select
    q.Close

End Sub
```

See Also

Close, **Create**, **Label**, **MSMQQueue**, **MSMQQueueInfo**, **Open**, **PathName**

Ack

MSMQMessage

The **Ack** property specifies the type of acknowledgment messages that MSMQ posts (in the administration queue) when the message is sent.

Quick Info

Type:	**Long**
Run time:	read/write

Syntax

object.**Ack**

Syntax Element	Description
object	Message (**MSMQMessage**) object that defines the message.

Settings

The **Ack** property can have any one of the following values:

MQMSG_ACKNOWLEDGMENT_FULL_REACH_QUEUE
: Posts positive and negative acknowledgments depending on whether or not the message reaches the queue. This can happen when *the time-to-reach-queue* timer expires, or when a message cannot be authenticated.

MQMSG_ACKNOWLEDGMENT_FULL_RECEIVE
: Posts a positive or negative acknowledgment depending on whether or not the message is retrieved from the queue before its *time-to-be-received* timer expires.

MQMSG_ACKNOWLEDGMENT_NACK_REACH_QUEUE
: Posts a negative acknowledgment when the message cannot reach the queue. This can happen when the *time-to-reach-queue* timer expires, or a message cannot be authenticated

MQMSG_ACKNOWLEDGMENT_NACK_RECEIVE
: Posts a negative acknowledgment when an error occurs and the message cannot be retrieved from the queue before its *time-to-be-received* timer expires.

MQMSG_ACKNOWLEDGMENT_NONE
: The default. No acknowledgment messages (positive or negative) are posted.

Remarks

Positive and negative acknowledgments are MSMQ-generated messages that are sent to an administration queue specified by the message. For an explanation of administration queues, see Administration Queues.

Acknowledgment messages contain some of the information found in the original message; however, each acknowledgment message has its own message identifier and class. The message identifier, **Id**, identifies the acknowledgment in the same way it identifies each message sent by an MSMQ application. The message class, **Class**, identifies the type of acknowledgment that was posted. Both these properties are set by MSMQ when it creates the acknowledgment message.

To indicate that acknowledgment messages are needed, set **Ack** and **AdminQueueInfo** when sending the message.

The receiving application can determine if MSMQ is sending acknowledgments back to the sending application by examining **Ack** and **AdminQueueInfo** when reading the message in the queue.

For information on the *time-to-reach-queue* and *time-to-be-received* timer, see Message Timers. To set

the *time-to-reach-queue* and *time-to-be-received* timers, set **MaxTimeToReachQueue** and **MaxTimeToReceive** properties, respectively.

For a example using acknowledgment messages, see Sending Messages that Request Acknowledgments.

Example

This example uses an administration queue to see if a message reaches its destination queue. It sends a message and then reads the acknowledgment message (returned by MSMQ) to see if the original message reached its destination. The destination and administration queues are created if they don't exist.

To try this example using Microsoft Visual Basic (version 5.0), paste the code into the code window of a form, and then run the example and click the form.

```
Dim query As New MSMQQuery
Dim qinfos As MSMQQueueInfos
Dim qinfoAdmin As MSMQQueueInfo
Dim qinfoDest As MSMQQueueInfo
Dim q As MSMQQueue
Dim msgSent As New MSMQMessage
Dim msgAdmin As MSMQMessage

Private Sub Form_Click()
    '**********************************
    ' Locate administration queue
    '(create one if one doesn't exist).
    '**********************************
    Set qinfos = query.LookupQueue(Label:="Administration Queue")
    qinfos.Reset
    Set qinfoAdmin = qinfos.Next
    If qinfoAdmin Is Nothing Then
        Set qinfoAdmin = New MSMQQueueInfo
        qinfoAdmin.PathName = ".\AdminQueue"
        qinfoAdmin.Label = "Administration Queue"
        qinfoAdmin.Create
    End If

    '**********************************
    ' Locate destination queue
    '(create one if one doesn't exist).
    '**********************************
    Set qinfos = query.LookupQueue(Label:="Destination Queue")
    qinfos.Reset
    Set qinfoDest = qinfos.Next
    If qinfoDest Is Nothing Then
        Set qinfoDest = New MSMQQueueInfo
        qinfoDest.PathName = ".\DestQueue"
        qinfoDest.Label = "Destination Queue"
        qinfoDest.Create
    End If

    '************************
    ' Open destination queue.
    '************************
    Set q = qinfoDest.Open(MQ_SEND_ACCESS, MQ_DENY_NONE)
```

```
    '**************
    ' Send Message.
    '**************
    msgSent.Label = "Test Message"
    msgSent.Body = "This message tests acknowledgment messages."
    msgSent.Ack = MQMSG_ACKNOWLEDGMENT_FULL_REACH_QUEUE
    Set msgSent.AdminQueueInfo = qinfoAdmin
    msgSent.Send q

    MsgBox "The message was sent. Check the MSMQ Explorer to
            see the messages in the queue."
    q.Close

    '********************************
    ' Read Acknowledgment message in the
    ' administration queue.
    '********************************
    Set q = qinfoAdmin.Open(MQ_RECEIVE_ACCESS, MQ_DENY_NONE)
    Set msgAdmin = q.Receive

    If msgAdmin.Class = MQMSG_CLASS_ACK_REACH_QUEUE Then
        MsgBox "The message reached the queue."
    Else
        MsgBox " The message did not reach the queue."
    End If

End Sub
```

See Also

AdminQueueInfo, **Body**, **Class**, **Close**, **Create**, **Label**, **LookupQueue**, **MSMQMessage**, **MSMQQueue**, **MSMQQueueInfo**, **MSMQQueueInfos**, **MSMQQuery**, **Next**, **Open**, **PathName**, **Receive**, **Reset**, **Send**

AdminQueueInfo

MSMQMessage

The **AdminQueueInfo** property specifies the queue used for MSMQ-generated acknowledgment messages. This object is passed to the Queue Manager on the target machine.

Quick Info

Type:	**MSMQQueueInfo**
Run time:	read/write

Syntax

set object1.**AdminQueueInfo =** *object2*

Syntax Element	Description
object1	Message (**MSMQMessage**) object that represents the message.
object2	Queue information (**MSMQQueueInfo**) object that represents the administration queue.

Settings

MSMQQueueInfo object.

Remarks

Acknowledgment messages are sent to the administration queue by MSMQ. For information on acknowledgment messages and administration queues, see Acknowledgment Messages and Administration Queues.

The sending application indicates that it wants MSMQ to return acknowledgment messages by attaching **Ack** and **AdminQueueInfo** to the message.

The receiving application can determine if MSMQ is sending acknowledgments back to the sending application by examining **Ack** and **AdminQueueInfo** when reading the message in the queue.

For a complete discussion of sending messages that return acknowledgment messages, see Sending Messages that Request Acknowledgments.

Example

This example uses an administration queue to see if a message reaches its destination queue. It sends a message and then reads the acknowledgment message (returned by MSMQ) to see if the original message reached its destination. The destination and administration queues are created if they don't exist.

To try this example using Microsoft Visual Basic (version 5.0), paste the code into the code window of a form, and then run the example and click the form.

```
Dim query As New MSMQQuery
Dim qinfos As MSMQQueueInfos
Dim qinfoAdmin As MSMQQueueInfo
Dim qinfoDest As MSMQQueueInfo
Dim q As MSMQQueue
Dim msgSent As New MSMQMessage

Private Sub Form_Click()
    '**********************************
```

```
    ' Locate administration queue
    '(create one if one doesn't exist).
    '***********************************
    Set qinfos = query.LookupQueue(Label:="Administration Queue")
    qinfos.Reset
    Set qinfoAdmin = qinfos.Next
    If qinfoAdmin Is Nothing Then
        Set qinfoAdmin = New MSMQQueueInfo
        qinfoAdmin.PathName = ".\AdminQueue"
        qinfoAdmin.Label = "Administration Queue"
        qinfoAdmin.Create
    End If

    '***********************************
    ' Locate destination queue
    '(create one if one doesn't exist).
    '***********************************
    Set qinfos = query.LookupQueue(Label:="Destination Queue")
    qinfos.Reset
    Set qinfoDest = qinfos.Next
    If qinfoDest Is Nothing Then
        Set qinfoDest = New MSMQQueueInfo
        qinfoDest.PathName = ".\DestQueue"
        qinfoDest.Label = "Destination Queue"
        qinfoDest.Create
    End If

    '*************************
    ' Open destination queue.
    '*************************
    Set q = qinfoDest.Open(MQ_SEND_ACCESS, MQ_DENY_NONE)

    '**************
    ' Send Message.
    '**************
    msgSent.Label = "Test Message"
    msgSent.Body = "This message tests acknowledgment messages."
    msgSent.Ack = MQMSG_ACKNOWLEDGMENT_FULL_REACH_QUEUE
    Set msgSent.AdminQueueInfo = qinfoAdmin
    msgSent.Send q

    MsgBox "The message was sent. Check the MSMQ Explorer to see the messages in the
queue."
    q.Close

    '************************************
    ' Read acknowledgment message in the
    ' administration queue.
    '************************************
    Set q = qinfoAdmin.Open(MQ_RECEIVE_ACCESS, MQ_DENY_NONE)
    Set msgAdmin = q.Receive

    If msgAdmin.Class = MQMSG_CLASS_ACK_REACH_QUEUE Then
        MsgBox "The message reached the queue."
    Else
```

```
        MsgBox " The message did not reach the queue."
    End If

End Sub
```

See Also

Body, **Class**, **Close**, **Create**, **Label**, **LookupQueue**, **MSMQMessage**, **MSMQQuery**, **MSMQQueue**, **MSMQQueueInfo**, **MSMQQueueInfos**, **Next**, **Open**, **PathName**, **Receive**, **Reset**, **Send**

AppSpecific

MSMQMessage

The **AppSpecific** property specifies application-generated information such as single integer values or application defined message classes.

Quick Info

Type:	**Long**
Run time:	read/write

Syntax

object.**AppSpecific**

Syntax Element	Description
object	Message (**MSMQMessage**) object that defines the message.

Remarks

Another message property that can be used to pass application-generated information is **CorrelationId**.

ArrivedTime

MSMQMessage

Read-only. The **ArrivedTime** property indicates when the message arrived at the queue.

Quick Info

Type:	**Date Variant**
Run time:	read-only

Syntax

object.**ArrivedTime**

Syntax Element	Description
object	Message (**MSMQMessage**) object that defines the message.

Return Values

Date and time when message arrived.

Remarks

The returned value for this property can be manipulated using standard Microsoft® Visual Basic® date and time functions such as **Date$**, and **Time$**. For descriptions of Visual Basic functions, see the Visual Basic documentation.

When reading messages from a journal queue, **ArrivedTime** indicates when the original message reached its queue, not when the original message was removed from the queue and a copy placed in the journal queue.

When reading messages from a machine journal, dead letter queue, or transactional dead letter queue, **ArrivedTime** indicates when the message reached the system queue where the application is reading the message. In these cases, the original message never reached its destination.

When **ArrivedTime** is displayed, Visual Basic will automatically convert the parameter's value to the local system time and system date.

Example

This example locates a destination queue (creating one if one does not exist), sends a message to the queue, then reads the message and displays when the message arrived in the queue.

To try this example using Microsoft Visual Basic (version 5.0), paste the code into the Code window of a form, and then run the example and click the form.

```
Dim query As New MSMQQuery
Dim qinfos As MSMQQueueInfos
Dim qinfoDest As MSMQQueueInfo
Dim qDest As MSMQQueue
Dim msgSent As New MSMQMessage
Dim msgRead As MSMQMessage


Private Sub Form_Click()


    '**********************************
    ' Locate destination queue
```

```
    '(create one if one doesn't exist).
    '***********************************
    Set qinfos = query.LookupQueue(Label:="Destination Queue")
    qinfos.Reset
    Set qinfoDest = qinfos.Next
    If qinfoDest Is Nothing Then
        Set qinfoDest = New MSMQQueueInfo
        qinfoDest.PathName = ".\DestQueue"
        qinfoDest.Label = "Destination Queue"
        qinfoDest.Create
    End If

    '**************
    ' Send Message.
    '**************
    Set qDest = qinfoDest.Open(MQ_SEND_ACCESS, MQ_DENY_NONE)
    msgSent.Label = "Test Message"
    msgSent.Body = "This message tests the message timers."
    msgSent.Send qDest
    qDest.Close

    '************************************
    ' Remove the message from destination
    ' queue and display when the message arrived.
    '************************************
    Set qDest = qinfoDest.Open(MQ_RECEIVE_ACCESS, MQ_DENY_NONE)
    Set msgRead = qDest.Receive

    MsgBox "The message arrived at: " + CStr(msgRead.ArrivedTime)

End Sub
```

See Also

Body, **Close**, **Create**, **Label**, **LookupQueue**, **MSMQMessage**, **MSMQQuery**, **MSMQQueue**, **MSMQQueueInfo**, **MSMQQueueInfos**, **Next**, **Open**, **PathName**, **Receive**, **Send**

Authenticate

MSMQQueueInfo

Optional. The **Authenticate** property specifies whether or not the queue only accepts authenticated messages.

Quick Info

Type:	**Long**
Run time:	read/write

Syntax

object.**Authenticate**

Syntax Element	Description
object	Queue information (**MSMQQueueInfo**) object that defines the queue.

Settings

Set **Authenticate** to one of the following values:

MQ_AUTHENTICATE_NONE
The default. The queue accepts authenticated and non-authenticated messages.

MQ_AUTHENTICATE
The queue only accepts authenticated messages.

Remarks

If the authentication level of the message (**AuthLevel**) does not match the authentication level of the queue, the message is rejected by the queue. In addition, if the sending application requested a negative acknowledgment message when it sent the message, MQMSG_CLASS_BAD_SIGNATURE will be returned to the sending application to indicate the message was rejected.

For information on how MSMQ authenticates messages, see Message Authentication.

To set the authentication level of a queue, set **Authenticate** and call the **MSMQQueueInfo** object's **Create** method.

The authentication level of a queue can be reset after the queue is created. To change the authentication level of a queue that is open, set **Authenticate** to a new level and call the **MSMQQueueInfo** object's **Update** method. To change the authentication level of a closed queue, set **Authenticate** to the new level. If the queue is not open there is no need to call **Update**, the queue's properties are updated automatically when the queue is opened.

To find the authentication level of a queue, call the **MSMQQueueInfo** object's **Refresh** method.

The receiving application can check if the message was authenticated by looking at the message's **IsAuthenticated** property.

Example

This example creates a private queue on the local machine, setting the queue's authentication level to MQ_AUTHENTICATE.

To try this example using Microsoft® Visual Basic® (version 5.0), make sure the Microsoft Message Queue Object Library is referenced, then paste the following code into the code window of a form, run the example, then click on the form.

```
Dim qinfo As MSMQQueueInfo
```

```
Private Sub Form_Click()

   Set qinfo = New MSMQQueueInfo
   qinfo.PathName = ".\PRIVATE$\TestQueue"
   qinfo.Label = "Test Queue"
   qinfo.Authenticate = MQ_AUTHENTICATE
   qinfo.Create

   MsgBox "Queue's Format name is: " + qinfo.FormatName

End Sub
```

See Also

AuthLevel, **Create**, **FormatName**, **IsAuthenticated**,**Label**, **PathName**, **Update**

AuthLevel

MSMQMessage

The **AuthLevel** property specifies whether or not the message must be authenticated when it arrives at the target queue.

Quick Info

Type:	**Long**
Run time:	read/write

Syntax

object.**AuthLevel**

Syntax Element	Description
object	Message (**MSMQMessage**) object that defines the queue.

Settings

Set **AuthLevel** to one of the following values:

MQMSG_AUTH_LEVEL_NONE
: The default. The message does not have to be authenticated when it arrives at the target queue

MQMSG_AUTH_LEVEL_ALWAYS
: The message must be authenticated when it arrives at the target queue.

Remarks

This property is only used when sending messages. Messages are authenticated by MSMQ, not by the receiving application.

In addition to messages, queues also have an authentication level. The queue's **Authenticate** property specifies whether or not the queue accepts only authenticated messages.

BasePriority

MSMQQueueInfo

Optional. The **BasePriority** property specifies a single base priority for all messages sent to a public queue.

Quick Info

Type:	**Long**
Run time:	read/write

Syntax

object.**BasePriority**

Syntax Element	Description
object	Queue information (**MSMQQueueInfo**) object that defines the queue.

Settings

Integer value between -32768 and +32767 (default is 0).

Remarks

A public queue's base priority is used for routing the queue's messages over the network. It can be used to give the messages sent to the queue a higher (or lower) priority than messages sent to other queues. When a queue's base priority is set, all the messages sent to it are given a higher priority than messages sent to queues with a lower base priority. The queue's base priority has no effect on the order of the messages in the queue, or how messages are read from the queue.

BasePriority only applies to public queues that can be located through MQIS (using a public format name). The base priority of private queues, as well as public queues accessed directly, is always 0.

MSMQ combines the queue's base priority with the message's priority (**Priority**) to determine the overall priority of a message when it is sent to the queue.

To set the base priority of a public queue, set **BasePriority** and call the **MSMQQueueInfo** object's **Create** method.

To reset the base priority of a public queue after the queue is created, set **BasePriority** to a new level and, if the queue is open, call the **MSMQQueueInfo** object's **Update** method. If the queue is not open do not call **Update**, the queue's properties are updated automatically when the queue is opened.

The base priority of a queue can be reset after the queue is created. To change the base priority of a queue that is open, set **BasePriority** to a new level and call the **MSMQQueueInfo** object's **Update** method. To change the base priority of a closed queue, set **Basepriority** to the new level. If the queue is not open there is no need to call **Update**, the queue's properties are updated automatically when the queue is opened.

To find the base priority of a queue, call the **MSMQQueueInfo** object's **Refresh** method.

Example

This example creates a public queue on the local machine, setting the queue's base priority to 7.

To try this example using Microsoft® Visual Basic® (version 5.0), make sure the Microsoft Message Queue Object Library is referenced, then paste the following code into the code window of a form, run the example, then click on the form.

```
Dim qinfo As MSMQQueueInfo
```

```
Private Sub Form_Click()

   Set qinfo = New MSMQQueueInfo
   qinfo.PathName = ".\basepriorityTest"
   qinfo.Label = "Test Queue"
   qinfo.BasePriority = 7
   qinfo.Create

   MsgBox "Queue's Format name is: " + qinfo.FormatName

End Sub
```

See Also

Create, **FormatName**, **Label**, **MSMQQueueInfo** **PathName**, **Priority**, **Refresh**, **Update**

Body

MSMQMessage

The **Body** property specifies the contents of the message.

Quick Info

Type:	**Variant**
Run time:	read/write

Syntax

object.**Body**

Syntax Element	Description
object	Message (**MSMQMessage**) object that defines message.

Settings

Any simple Variant type: string, array of bytes, numeric type, currency, date, or persistent ActiveX object.

Remarks

The sending application does not indicate the type of information (string, array of bytes, numeric types, currency, date, or ActiveX object) that is stored in the message body. MSMQ determines the body type from the true type of the Variant assigned to the Body property.

The receiving application, however, should determine what type of information is in the message body. To do this, it can inspect the message body using a standard Visual Basic **If..Then..Else** function with the following conditional functions: **TypeOf** and **TypeName**. The **TypeName** function can be used to find out if the message in the queue is a string, array of bytes, numeric type, currency, or date. If it is not one of these, **TypeOf** can be used to see which **OLE** interface the object supports. For an example on how MSMQ reads messages from a queue, see Reading Messages in a Queue.

For information on the type of objects that can be sent, see MSMQ ActiveX Support.

For an example of how MSMQ sends the messages to a queue, see Sending Messages to a Queue.

Example

This example creates a queue, opens the queue for sending messages, sets the body of a message, then sends the message to the queue.

To try this example using Microsoft® Visual Basic® (version 5.0), paste the code into the Code window of a form, and then run the example and click the form.

```
Dim qinfo As MSMQQueueInfo
Dim q As MSMQQueue
Dim msg As New MSMQMessage

Private Sub Form_Click()
    '**************
    ' Create queue.
    '**************
  Set qinfo = New MSMQQueueInfo
  qinfo.PathName = ".\SendTest"
  qinfo.Label = "Test Queue"
  qinfo.Create
```

```
  '**************
  ' Open queue.
  '**************
Set q = qinfo.Open(MQ_SEND_ACCESS, MQ_DENY_NONE)
  '**************
  ' Send Message.
  '**************
msg.Label = "Test Message"
msg.Body = "This is a test message with a string Body."
msg.Send q


 q.Close

End Sub
```

See Also

Close, **Create**, **FormatName**, **Label**, **MSMQMessage**, **MSMQQueue**, **MSMQQueueInfo**, **Open**, **PathName**, **Send**

BodyLength

MSMQMessage

Read-only. The **BodyLength** property indicates the length of the message Body in bytes.

Quick Info

Type:	**Long**
Run time:	read-only

Syntax

object.**BodyLength**

Syntax Element	Description
object	Message (**MSMQMessage**) object that represents the message.

Return Values

Length (in bytes) of the message.

Remarks

BodyLength can be used by the sending application or receiving application.

Sending applications can use **BodyLength** to find the size of the message body as soon as **Body** is set, including before and after the message is sent.

Receiving applications can use **BodyLength** to find the size of a message before or after retrieving the message body.

Class

MSMQMessage

Read-only. The **Class** property indicates message type. A message can be a normal MSMQ message, a positive or negative (arrival and read) acknowledgment message, or a report message. This property is set by MSMQ.

Quick Info

Type:	**Long**
Run time:	read-only

Syntax

object.**Class**

Syntax Element	Description
object	Message (**MSMQMessage**) object that represents the message.

Return Values

MSMQ sets the **Class** property to one of the following values:

Normal messages (all messages created by your application):

MQMSG_CLASS_NORMAL
: A normal MSMQ message.

Positive acknowledgment messages (generated by MSMQ):

MQMSG_CLASS_ACK_REACH_QUEUE
: The original message reached its destination queue.

MQMSG_CLASS_ACK_RECEIVE
: The original message was retrieved by the receiving application.

Negative arrival acknowledgment messages (generated by MSMQ):

MQMSG_CLASS_NACK_ACCESS_DENIED
: The sending application does not have access rights to the destination queue.

MQMSG_CLASS_NACK_BAD_DST_Q
: The destination queue is not available to the sending application.

MQMSG_CLASS_NACK_BAD_ENCRYPTION
: The destination Queue Manager could not decrypt a private (encrypted) message (see **PrivLevel**).

MQMSG_CLASS_NACK_BAD_SIGNATURE
: MSMQ could not authenticate the original message. The original message's digital signature is not valid.

MQMSG_CLASS_NACK_COULD_NOT_ENCRYPT
: The source Queue Manager could not encrypt a private message (see **PrivLevel**).

MQMSG_CLASS_NACK_HOP_COUNT_EXCEEDED
: The original message's hop count is exceeded.

MQMSG_CLASS_NACK_Q_EXCEED_QUOTA
: The original message's destination queue is full.

MQMSG_CLASS_NACK_REACH_QUEUE_TIMEOUT
: Either the time-to-reach-queue or time-to-be-received timer expired before the original message could reach the destination queue.

MQMSG_CLASS_NACK_PURGED
: The message was purged before reaching the destination queue.

MQMSG_CLASS_NACK_NOT_TRANSACTIONAL_Q
A transaction message was sent to a non-transaction queue.

MQMSG_CLASS_NACK_NOT_TRANSACTIONAL_MSG
A non-transaction message was sent to a transaction queue.

Negative read acknowledgment messages (generated by MSMQ):

MQMSG_CLASS_NACK_Q_DELETED
The queue was deleted before the message could be read from the queue.

MQMSG_CLASS_NACK_Q_PURGED
The queue was purged and the message no longer exists.

MQMSG_CLASS_NACK_RECEIVE_TIMEOUT
The original message was not removed from the queue before its time-to-be-received timer expired.

Report messages (generated by MSMQ):

MQMSG_CLASS_REPORT
Sent each time the message enters or leaves an MSMQ server.

Remarks

Acknowledgment messages are generated by MSMQ whenever the sending application requests them. MSMQ returns the appropriate acknowledgment message to the administration queue that is specified by the sending application. For information on administration queues, see Administration Queues. For a complete discussion of requesting acknowledgment messages, Sending Messages that Request Acknowledgments.

Report messages are generated by MSMQ whenever a report queue is defined at the source Queue Manager. For information on report queues, see Report Queues.

When reading messages in an administration queue or dead letter queue, retrieve **Class** to find out why the message was sent to the queue.

Example

This example uses an administration queue to see if a message reaches its destination queue. It sends a message and then looks at the **Class** property of the acknowledgment message (returned by MSMQ) to see if the original message reached its destination. The destination and administration queues are created if they don't exist.

To try this example using Microsoft Visual Basic (version 5.0), paste the code into the code window of a form, and then run the example and click the form.

```
Dim query As New MSMQQuery
Dim qinfos As MSMQQueueInfos
Dim qinfoAdmin As MSMQQueueInfo
Dim qinfoDest As MSMQQueueInfo
Dim q As MSMQQueue
Dim msgSent As New MSMQMessage
Dim msgAdmin As MSMQMessage

Private Sub Form_Click()
    '**********************************
    ' Locate administration queue
    '(create one if one doesn't exist).
    '**********************************
    Set qinfos = query.LookupQueue(Label:="Administration Queue")
    qinfos.Reset
    Set qinfoAdmin = qinfos.Next
    If qinfoAdmin Is Nothing Then
```

```
        Set qinfoAdmin = New MSMQQueueInfo
        qinfoAdmin.PathName = ".\AdminQueue"
        qinfoAdmin.Label = "Administration Queue"
        qinfoAdmin.Create
    End If

    '**********************************
    ' Locate destination queue
    '(create one if one doesn't exist).
    '**********************************
    Set qinfos = query.LookupQueue(Label:="Destination Queue")
    qinfos.Reset
    Set qinfoDest = qinfos.Next
    If qinfoDest Is Nothing Then
        Set qinfoDest = New MSMQQueueInfo
        qinfoDest.PathName = ".\DestQueue"
        qinfoDest.Label = "Destination Queue"
        qinfoDest.Create
    End If

    '************************
    ' Open destination queue.
    '************************
    Set q = qinfoDest.Open(MQ_SEND_ACCESS, MQ_DENY_NONE)

    '***************
    ' Send Message.
    '***************
    msgSent.Label = "Test Message"
    msgSent.Body = "This message tests acknowledgment messages."
    msgSent.Ack = MQMSG_ACKNOWLEDGMENT_FULL_REACH_QUEUE
    Set msgSent.AdminQueueInfo = qinfoAdmin
    msgSent.Send q

    MsgBox "The message was sent. Check the MSMQ Explorer to see the messages in the
queue."
    q.Close

    '*******************************
    ' Read Acknowledgment message in the
    ' administration queue.
    '*******************************
    Set q = qinfoAdmin.Open(MQ_RECEIVE_ACCESS, MQ_DENY_NONE)
    Set msgAdmin = q.Receive

    If msgAdmin.Class = MQMSG_CLASS_ACK_REACH_QUEUE Then
        MsgBox "The message reached the queue."
    Else
        MsgBox " The message did not reach the queue."
    End If

End Sub
```

See Also

Ack, **AdminQueueInfo**, **Body**, **Close**, **Create**, **Label**, **LookupQueue**, **MSMQMessage**, **MSMQQueue**, **MSMQQueueInfo**, **MSMQQueueInfos**, **MSMQQuery**, **Next**, **Open**, **PathName**, **Receive**, **Reset**, **Send**

CorrelationId

MSMQMessage

The **CorrelationId** property identifies the message using a 20-byte, correlation identifier.

Quick Info

Type:	Variant (array of bytes).
Run time:	read/write

Syntax

object.**CorrelationId**

Syntax Element	Description
object	Message (**MSMQMessage**) object that defines the message.

Settings

A 20-byte application-defined correlation identifier.

Remarks

CorrelationId provides an application-defined identifier that the receiving application can use to sort messages.

When sending response messages to the sending application, **CorrelationId** can be set to the message identifier (**Id**) of the message that is in the queue. This provides an easy mechanism that the sending application can use to match the response message with the message that was sent.

When MSMQ generates an acknowledgment or report message, it uses the **CorrelationId** property to specify the message identifier of the original message. The application can then look at the **CorrelationId** property to find the message identifier of the original message.

Note MSMQ Connector applications must also set the correlation identifier of the acknowledgment and report messages to the message identifier of the original message.

CreateTime

MSMQQueueInfo

Read-only. The **CreateTime** property indicates when the public queue was created.

Quick Info

Type:	**Date Variant**
Run time:	read-only

Syntax

object.**CreateTime**

Syntax Element	Description
object	Queue information (**MSMQQueueInfo**) object that defines the queue.

Return Values

Date when queue was created.

Remarks

To read this property, the application must first call the **MSMQQueueInfo** object's **Refresh** method. Although MSMQ updates MQIS when the queue is created, **CreateTime** is not updated until **Refresh** is called.

The returned value for this property can be manipulated using standard Microsoft® Visual Basic® date and time functions such as **Date$**, and **Time$**. For descriptions of Visual Basic functions, see the Visual Basic documentation.

When **CreateTime** is displayed, Visual Basic will automatically convert the parameter's value to the local system time and system date.

The **CreateTime** property can also be used when making a query. See the **MSMQQuery** object's **LookupQueue** method for details on running a query.

Example

This example uses the CreateTime and RelCreateTime parameters of LookupQueue to locate all the public queues. To locate the queues, MSMQ compares the date specified by the CreateTime parameter with the date of each queue's CreateTime property.

To try this example using Microsoft Visual Basic (version 5.0), paste the code into the Code window of a form, and then run the example and click the form.

```
Dim query As New MSMQQuery
Dim qinfos As MSMQQueueInfos
Dim qinfoDest As MSMQQueueInfo
Dim queueCount As Integer

Private Sub Form_Click()

    '**********************
    ' Locate public queues
    '**********************
    Set qinfos = query.LookupQueue(CreateTime:=Now, RelCreateTime:=REL_LT)
    qinfos.Reset
```

```
    '****************************************
    ' Display create time of all public queues.
    '****************************************
    Set qinfoDest = qinfos.Next
    queueCount = 0                  'Counter for number of queues found.

    While Not qinfoDest Is Nothing
        MsgBox "This public queue (" + qinfoDest.Label + ") was created on: " +
CStr(qinfoDest.CreateTime)
        queueCount = queueCount + 1
        Set qinfoDest = qinfos.Next
    Wend

    MsgBox "The total public queues found were: " + CStr(queueCount)

End Sub
```

See Also

Label, **LookupQueue**, **MSMQQuery**, **MSMQQueueInfo**, **MSMQQueueInfos**, **Next**, **Reset**

Delivery

MSMQMessage

The **Delivery** property specifies how MSMQ delivers the message.

Quick Info

Type:	**Long**
Run time:	read/write

Syntax

object.**Delivery**

Syntax Element	Description
object	Message (**MSMQMessage**) object that represents the message.

Settings

The **Delivery** property can have one of the following values:

MQMSG_DELIVERY_RECOVERABLE
: In every hop along its route, the message is forwarded to the next hop or stored locally in a backup file until delivered. This guarantees delivery even in the case of a machine crash.

MQMSG_DELIVERY_EXPRESS
: The default. The message stays in memory until it can be delivered. (In-memory message store and forward.)

Remarks

When the message's delivery mechanism is set to MQMSG_DELIVERY_EXPRESS, the message has faster throughput. When set to MQMSG_DELIVERY_RECOVERABLE, throughput may be slower, however, MSMQ guarantees that the message will be delivered, even if a computer crashes while the message is en-route to the queue.

MSMQ always sets the delivery mechanism of transactional messages to MQMSG_DELIVERY_RECOVERABLE. For information on transactions, see MSMQ Transactions.

Example

This example first creates and opens a queue for sending messages, then sets the delivery mechanism for a message and sends it off to the queue.

To try this example using Microsoft® Visual Basic® (version 5.0), paste the code into the code window of a form, and then run the example and click the form.

```
Dim qinfo As MSMQQueueInfo
Dim q As MSMQQueue
Dim msg As New MSMQMessage

Private Sub Form_Click()
    '*************************
    ' Create queue (no error
    ' handling if queue exists).
    '*************************
  Set qinfo = New MSMQQueueInfo
  qinfo.PathName = ".\DeliveryTest"
  qinfo.Label = "Test Queue"
```

```
   qinfo.Create
    '**************
    ' Open queue.
    '**************
   Set q = qinfo.Open(MQ_SEND_ACCESS, MQ_DENY_NONE)
    '**************
    ' Send Messages.
    '**************
   msg.Label = "Test Message"
   msg.Body = "This is a test message with a string Body."
   msg.Delivery = MQMSG_DELIVERY_RECOVERABLE

   msg.Send q

   MsgBox "The message was sent. Use the MSMQ Explorer to see the message in the
queue."

   q.Close

End Sub
```

See Also

Body, **Close**, **Create**, **Label**, **MSMQMessage**, **MSMQQueue**, **MSMQQueueInfo**, **Open**, **PathName**, **Send**

DestinationQueueInfo

MSMQMessage

The **DestinationQueueInfo** property specifies the destination queue for the message.

Quick Info

Type:	**MSMQQueueInfo**
Run time:	read-only

Syntax

set object2 = object1.**DestinationQueueInfo**

Syntax Element	Description
object1	Message (**MSMQMessage**) object that represents the message.
object2	Queue information (**MSMQQueueInfo**) object that represents the destination queue.

Settings

MSMQQueueInfo object.

Remarks

The **DestinationQueueInfo** property is set by MSMQ when a message is sent. It is typically used when reading messages from a machine journal, dead letter queue, or response messages from a response queue. **DestinationQueueInfo** provides the destination queue of the original message.

To provide the destination queue of a message in a response message, set the correlation identifier (**CorrelationId**) of the response message to the **DestinationQueueInfo** property of the original message. The application reading the response message can then look at the correlation identifier to determine the origin of the response message. For an example of this, see Sending Messages that Request a Response.

EncryptAlgorithm

MSMQMessage

The **EncryptAlgorithm** property specifies the encryption algorithm used by MSMQ to encrypt private messages.

Quick Info

Type:	**Long**
Run time:	read-write

Syntax

object.**EncrypAlgorithm**

Syntax Element	Description
object	Message (**MSMQMessage**) object that represents the message.

Settings

The **EncryptAlgorithm** property can have any one of the following values:

MQMSG_RC2
Default.

MQMSG_RC4

Message encryption is based on public-key encryption using the Microsoft® Cryptography API with an underlying RSA provider.

Remarks

EncryptAlgorithm is used with the **MSMQMessage** object's **PrivLevel** property to send private messages.

For a discussion on private messages, see Private Messages.

For a complete example of sending a private message (including setting the privacy level of a queue), see Sending Private Messages.

FormatName

MSMQQueueInfo

The **FormatName** property specifies the format name of the queue. This property must be set before the queue is opened.

When creating a queue, if a format name is not provided by the application, MSMQ generates one from the queue's MSMQ pathname (**PathName**). After the queue is created, you can use the name generated by MSMQ, or specify a different one.

Quick Info

Type:	**String**
Run time:	read/write

Syntax

object.**FormatName**

Syntax Element	Description
object	The queue (**MSMQQueueInfo**) object in question.

Settings

String. Possible strings are:

PUBLIC=QueueGUID
DIRECT=Protocol:MachineAddress\QueueName
DIRECT=OS:MachineName\QueueName
PRIVATE=MachineGUID\QueueNumber

Remarks

The **FormatName** property must be specified to open the queue. MSMQ uses the queue's format name to identify which queue to open and how to access the opened queue. A queue's format name cannot be changed while the queue is open.

For information on how MSMQ uses format names, see Format Name.

Example

This example creates a private queue on the local machine, then opens the queue with the format name provided by MSMQ. To try this example using Microsoft Visual Basic (version 5.0), paste the code into the Code window of a form, and then run the example and click the form.

```
Dim qinfo As MSMQQueueInfo
Dim q As MSMQQueue

Private Sub Form_Click()

  Set qinfo = New MSMQQueueInfo
  qinfo.PathName = ".\PRIVATE$\FormatNameTest"
  qinfo.Label = "Test Queue"
  qinfo.Create

  MsgBox "Queue format name is: " + qinfo.FormatName

  Set q = qinfo.Open(MQ_SEND_ACCESS, MQ_DENY_NONE)
```

```
    On Error GoTo ErrorHandler

End Sub
```

See Also

Create, **IsOpen**, **Label**, **MSMQQueue**, **MSMQQueueInfo**, **Open**, **PathName**

Handle

MSMQQueue

Read-only. The **Handle** property indicates the handle of the opened queue.

Quick Info

Type:	**Long**
Run time:	read-only

Syntax

object.**Handle**

Syntax Element	Description
object	Queue (**MSMQQueue**) object that represents the open instance of the queue.

Return Values

Queue handle.

Remarks

This handle refers to this instance of the queue. The value of **Handle** changes each time a queue is opened.

After the queue is closed, MSMQ sets **Handle** to INVALID_HANDLE_VALUE.

Handle can be used to call MSMQ API functions directly. For example, when using Microsoft® Visual Basic®, MSMQ functions can be called directly using the Declare Function facility.

Example

This example opens a queue for sending messages, then uses the value of **Handle** to test if the queue is open. To try this example using Microsoft Visual Basic (version 5.0), paste the code into the Code window of a form, and then run the example and click the form.

```
Dim qinfo As MSMQQueueInfo
Dim q As MSMQQueue

Private Sub Form_Click()

  Set qinfo = New MSMQQueueInfo
  qinfo.PathName = ".\Handle"
  qinfo.Label = "Test Queue"
  qinfo.Create

  Set q = qinfo.Open(MQ_SEND_ACCESS, MQ_DENY_NONE)

  If q.Handle = INVALID_HANDLE_VALUE Then
     MsgBox "The queue is closed"
  Else
     MsgBox "The queue is open. Handle is: " + CStr(q.Handle)
  End If

End Sub
```

See Also

Create, **Label**, **MSMQueue**, **MSMQQueueInfo**, **Open**, **PathName**

HashAlgorithm

MSMQMessage

The **HashAlgorithm** property specifies the hash algorithm used by MSMQ when authenticating messages.

Quick Info

Type:	**Long**
Run time:	read-write

Syntax

object.**HashAlgorithm**

Syntax Element	Description
object	Message (**MSMQMessage**) object that represents the message.

Settings

This property can be set to any of the values defined by the ALG_ID data type in wincrypt.h (the default is CALG_MD5) or in the MSMQ_CALG enumeration.

Remarks

The MSMQ run-time code uses the hashing algorithm when creating a digital signature and when authenticating the message.

For information on what MSMQ does to authenticate messages, see How MSMQ Authenticates Messages.

Id

MSMQMessage

Read-only. The **Id** property identifies the message using an MSMQ-generated message identifier.

Quick Info

Type:	Variant containing array of bytes.
Run time:	read-only

Syntax

object.**Id**

Syntax Element	Description
object	Message (**MSMQMessage**) object from queue.

Return Values

20-byte message identifier (array of bytes).

Remarks

MSMQ generates a 20-byte message identifier and attaches it to the message when the message is sent. The identifier is an array of bytes that can be read by either the sending or receiving application.

MSMQ generates message identifiers for all messages, including acknowledgment messages generated by MSMQ and MSMQ Connector applications. When an acknowledgment message is created, the identifier of the original message can be found in the acknowledgment message's **CorrelationId** property.

Example

This example locates a destination queue (creating one if one does not exist), sends two messages to the queue, then reads the message and displays their message identifier.

To try this example using Microsoft Visual Basic (version 5.0), paste the code into the Code window of a form, and then run the example and click the form.

```
Dim query As New MSMQQuery
Dim qinfos As MSMQQueueInfos
Dim qinfoDest As MSMQQueueInfo
Dim q As MSMQQueue
Dim msgSent1 As New MSMQMessage
Dim msgSent2 As New MSMQMessage
Dim msgRead As MSMQMessage
Dim strID As String                 'String representation of ID.

Private Sub Form_Click()


    '**********************************
    ' Locate destination queue
    '(create one if one doesn't exist).
    '**********************************

    Set qinfos = query.LookupQueue(Label:="Destination Queue")
    qinfos.Reset
    Set qinfoDest = qinfos.Next
```

```
    If qinfoDest Is Nothing Then
        Set qinfoDest = New MSMQQueueInfo
        qinfoDest.PathName = ".\DestQueue"
        qinfoDest.Label = "Destination Queue"
        qinfoDest.Create
    End If

    '************************
    ' Open destination queue.
    '************************
    Set q = qinfoDest.Open(MQ_SEND_ACCESS, MQ_DENY_NONE)

    '**************
    ' Send Messages.
    '**************
    msgSent1.Label = "Test Message"
    msgSent1.Body = "This message tests the message identifier."
    msgSent1.Send q
    For counter = LBound(msgSent.Id) To UBound(msgSent.Id)
            strID = strID & Hex(msgSent.Id(Counter))
    Next counter
    MsgBox "Message (" + StrID + ") was sent to the queue."
    strID = ""

    msgSent2.Label = "Test Message"
    msgSent2.Body = "This message tests the message identifier."
    msgSent2.Send q
    For counter = LBound(msgSent2.Id) To UBound(msgSent2.Id)
            strID = strID & Hex(msgSent2.Id(Counter))
    Next counter

    MsgBox "Message (" + strID + ") was sent to the queue."
    strID = ""
    q.Close

    '************************************
    ' Read the message in the destination
    ' queue and display its identifier.
    '************************************

    Set q = qinfoDest.Open(MQ_RECEIVE_ACCESS, MQ_DENY_NONE)
    Set msgRead = q.Receive (ReceiveTimeout:=0)

    While Not msgRead Is Nothing
        For Counter = LBound(msgRead.Id) To UBound(msgRead.Id)
            strID = strID & Hex(msgRead.Id(Counter))
        Next Counter
        MsgBox "The message " + strID + " was removed from the queue."
        strID = ""
        Set msgRead = q.Receive
   Wend

End Sub
```

See Also

Body, **Close**, **Create**, **Label**, **MSMQMessage**, **MSMQQuery**, **MSMQQueue**, **MSMQQueueInfo**, **MSMQQueueInfos**, **Next**, **Open**, **PathName**, **Receive**, **Reset**, **Send**

IsAuthenticated

MSMQMessage

Read-only. The **IsAuthenticated** property indicates that the message was authenticated by MSMQ.

Quick Info

Type:	**Boolean**
Run time:	read-only

Syntax

object.**IsAuthenticated**

Syntax Element	Description
object	Message (**MSMQMessage**) object that defines the message.

Returned Value

TRUE
The message is authenticated.

FALSE
The message is not authenticated.

Remarks

The receiving application can use **IsAuthenticated** to test which messages have been authenticated if the target queue does not force authentication. By default, the target queue can accept authenticated and non-authenticated messages. To change this default condition the queue's **Authenticate** property must be set.

IsOpen

MSMQQueue

Read-only. The **IsOpen** property indicates whether or not the queue is open.

Quick Info

Type:	**Boolean**
Run time:	read-only

Syntax

object.**IsOpen**

Syntax Element	Description
object	Queue (**MSMQQueue**) object that represents an instance of the queue.

Return Values

TRUE
The queue is open.

FALSE
The queue is not open.

Remarks

IsOpen is TRUE if and only if **Handle** <> INVALID_HANDLE_VALUE.

See Also

Create, **Label**, **MSMQQueue**, **MSMQQueueInfo**, **Open**, **PathName**

IsTransactional

MSMQQueueInfo

Read-only. The **IsTransactional** property indicates whether or not the queue supports transactions.

Quick Info

Type:	**Boolean**
Run time:	read-only

Syntax

object.**IsTransactional**

Syntax Element	Description
object	Queue information (**MSMQQueueInfo**) object that defines the queue.

Return Values

TRUE
The queue is only used in transactions.

FALSE
The queue is not used in transactions.

Remarks

To specify a queue as a transaction queue, see **Create**.

If a queue is transactional, it can only accept messages that are sent as part of a transaction (see the **MSMQMessage** object's **Send** method). In a similar way, messages read from a transactional queue must also be done as part of a transaction (see the **MSMQQueue** object's **Peek**, **PeekNext**, **PeekCurrent**, **Receive**, or **ReceiveCurrent** methods).

For information on transactions, see MSMQ Transactions.

Example

This example creates a transactional private queue on the local machine, then uses the queue's **IsTransactional** property to display the appropriate message. To try this example using Microsoft Visual Basic (version 5.0), paste the code into the Code window of a form, and then run the example and click the form.

```
Dim qinfo As MSMQQueueInfo

Private Sub Form_Click()

  Set qinfo = New MSMQQueueInfo
  qinfo.PathName = ".\PRIVATE$\isTransactionTest"
  qinfo.Label = "Test Queue"
  qinfo.Create IsTransactional:=True     'Creates transactional queue.

  qinfo.Refresh
  MsgBox "The queue's format name is: " + qinfo.FormatName

  If qinfo.IsTransactional Then
     MsgBox "The queue is a transactional queue."
     Else
     MsgBox "The queue is not a transactional queue."
```

```
    End If

End Sub
```

See Also

Create, **FormatName**, **Label**, **MSMQMessage**, **MSMQQueueInfo**, **PathName**, **Peek**, **PeekNext**, **Receive**, **Refresh**, **Send**

IsWorldReadable

MSMQQueueInfo

Read-only. The **IsWorldReadable** property indicates if everyone can read the messages in the queue.

Quick Info

Type:	**BOOLEAN**
Run time:	read-only

Syntax

object.**IsWorldReadable**

Syntax Element	Description
object	Queue information (**MSMQQueueInfo**) object that defines the queue.

Return Values

TRUE
Everyone can read messages in the queue and its queue journal.

FALSE
Default. Only the owner of the queue can read the messages in the queue.

Remarks

MSMQ initially sets this property based on the **Create** method's optional *IsWorldReadable* parameter. After the queue is created, MSMQ resets **IsWorldReadable** whenever the queue's security is changed to allow or disallow read access to everyone.

This property has no effect on sending messages to the queue.

For information on access control, see Access Control.

Journal

MSMQQueueInfo **MSMQMessage**

The **Journal** property is used for queues (**MSMQQueueInfo** object) and messages (**MSMQMessage** object).

For a queue, **Journal** specifies whether or not messages retrieved from the queue are stored in a queue journal.

For a message, **Journal** specifies whether a copy of the message is sent to a machine journal when the message is sent, to a dead letter queue if the message could not be sent, or neither.

Quick Info

Type:	**Long**
Run time:	read/write

Syntax

object.**Journal**

Syntax Element	Description
object	Queue information (**MSMQQueueInfo**) object used to define a queue, or message (**MSMQMessage**) object used to define a message.

Settings

For queues, set **Journal** to one of the following values:

MQ_JOURNAL
: When a message is removed from the queue, it is stored in the queue journal.

MQ_JOURNAL_NONE
: The default. Messages are not stored in a journal queue when they are removed from the queue.

For messages, set **Journal** to one or more of the following values:

MQMSG_DEADLETTER
: If the message time-to-be-received or time-to-reach-queue setting expires, keep the message in the dead letter queue on the machine where time expired.

MQMSG_JOURNAL
: If the message is transmitted (from the originating machine to the next hop), keep it in the machine journal on the originating machine.

MQMSG_JOURNAL_NONE
: The default. The message is not kept in the originating machine's machine journal.

Remarks

Journal does not create a queue. Journal, machine, and dead letter queues are all system queues generated by MSMQ. For more information about types of queues, see Journal Queues and Dead Letter Queues. For an example of reading messages from a journal queue or dead letter queue, see Reading Messages In a Queue.

Queue Journal

To specify a journal queue, set **Journal** to MQ_JOURNAL and call the **MSMQQueueInfo** object's **Create** method.

To reset **Journal**, set **Journal** to a new value and, if the queue is open, call the **MSMQQueueInfo** object's **Update** method. If the queue is not open do not call **Update**, the queue's properties are

updated automatically when the queue is opened.

To find out if a queue is using a journal queue, call the **MSMQQueueInfo** object's **Refresh** method.

The size of the queue's journal queue can be set using the **JournalQuota** property.

Machine Journal

MSMQ automatically sends transactional messages to the transaction dead letter queue (DEADXACT) on the source machine if the message is not delivered. For information on transactions, see MSMQ Transactions.

Example: Specifying a queue journal

This example creates a private queue on the local machine, attaching a journal queue to the created queue. To try this example using Microsoft® Visual Basic® (version 5.0), paste the code into the Code window of a form, and then run the example and click the form.

```
Dim qinfo As MSMQQueueInfo

Private Sub Form_Click()

  Set qinfo = New MSMQQueueInfo
  qinfo.PathName = ".\PRIVATE$\JournalTest"
  qinfo.Label = "Test Queue"
  qinfo.Journal = MQ_JOURNAL
  qinfo.Create

  MsgBox "Queue's Format name is: " + qinfo.FormatName

End Sub
```

Example: Specifying a machine journal

This example first creates and opens a queue for sending messages, then sets the delivery mechanism for a message and sends it off to the queue.

To try this example using Microsoft Visual Basic (version 5.0), paste the code into the Code window of a form, and then run the example and click the form.

```
Dim msgMessage as New MSMQMessage

If qFriendQueue.IsOpen Then
    msgMessage.Journal = MQMSG_JOURNAL       'Specify machine journal.
    msgMessage.Body = Chr(KeyAscii)          'Fills message Body.
    msgMessage.Label = "myMessage"           'Sets message label.
    msgMessage.Send qFriendQueue             'Sends message.
End If
```

See Also

Body, **Create**, **FormatName**, **JournalQuota**, **Label**, **MSMQQueueInfo**, **PathName**, **Refresh**, **Send**, **Update**

JournalQuota

MSMQQueueInfo

Optional. The **JournalQuota** property specifies the maximum size (in kilobytes) of the queue journal.

Quick Info

Type:	**Long**
Run time:	read/write

Syntax

object.**Journal**

Syntax Element	Description
object	Queue information (**MSMQQueueInfo**) object that defines the queue.

Settings

Maximum size (in kilobytes) of the queue journal (the default is INFINITE).

Remarks

JournalQuota is used along with **Journal** to tell MSMQ to start storing a copy of the messages retrieved from the queue. For information on accessing queue journals, see Journal Queues.

To set the size of the journal queue, set **JournalQuota** and call the **MSMQQueueInfo** object's **Create** method.

To reset the size of a journal queue after the queue is created, set **JournalQuota** to a new value and, if the queue is open, call the **MSMQQueueInfo** object's **Update** method. If the queue is not open do not call **Update**, the queue's properties are updated automatically when the queue is opened.

To find the size of a journal queue, call the **MSMQQueueInfo** object's **Refresh** method.

Example: Specifying a queue journal

This example creates a private queue on the local machine, attaching a journal queue to the created queue whose size is 7K. To try this example using Microsoft Visual Basic (version 5.0), paste the code into the Code window of a form, and then run the example and click the form.

```
Dim qinfo As MSMQQueueInfo

Private Sub Form_Click()

  Set qinfo = New MSMQQueueInfo
  qinfo.PathName = ".\PRIVATE$\JournalQuotaTest"
  qinfo.Label = "Test Queue"
  qinfo.Journal = MQ_JOURNAL
  qinfo.JournalQuota = 7
  qinfo.Create

  MsgBox "Queue's Format name is: " + qinfo.FormatName

End Sub
```

See Also

Create, **FormatName**, **Journal**, **Label**, **MSMQQueueInfo**, **Refresh**, **PathName** **Update**

Label

MSMQQueueInfo **MSMQMessage**

Optional. The **Label** property specifies a description of the queue or message.

Quick Info

Type:	**String**
Run time:	read/write

Syntax

object.**Label**

Syntax Element	Description
object	Either the queue information (**MSMQQueueInfo**) object that defines the queue, or the message (**MSMQMessage**) object that defines the message.

Settings

Queue label

Application-defined string (default is ""). The maximum length of the string is MQ_MAX_Q_LABEL_LEN (124 Unicode characters).

Message label

Application-defined string describing the message. The maximum length of a message label is 250 Unicode characters (including end-of-line character).

Remarks

Queue label

For public queues, the queue's label can be used as the search criteria for a query. By setting the label of several queues to the same string, the application can later run a query on the queue label and return all the queues with the same label. (A query can also be used to retrieve the label of a public queue.) For information on running a query, see Locating a Public Queue.

To specify the label of a queue, set **Label** and call the **MSMQQueueInfo** object's **Create** method.

To reset the label of a queue after the queue is created, set **Label** to a new label and, if the queue is open, call the **MSMQQueueInfo** object's **Update** method. If the queue is not open do not call **Update**, the queue's properties are updated automatically when the queue is opened.

To find the label of a queue, call the **MSMQQueueInfo** object's **Refresh** method.

Message label

Message labels can be used by administration tools for display purposes. For example, a printing application could put the source application and document name in the label of each message it sends to the printer queue.

For an example of how MSMQ sends the messages to a queue, see Sending Messages To a Queue.

Example: Setting a queue label

This example creates a private queue on the local machine, setting the queue's label to "Test Queue". To try this example using Microsoft Visual Basic (version 5.0), paste the code into the Code window of a form, and then run the example and click the form.

```
Dim qinfo As MSMQQueueInfo
```

```
Private Sub Form_Click()

   Set qinfo = New MSMQQueueInfo
   qinfo.PathName = ".\PRIVATE$\myqueue"
   qinfo.Label = "Test Queue"
   qinfo.Create

   MsgBox "Queue's Format name is: " + qinfo.FormatName

End Sub
```

Example: Setting a message label

This example creates a queue, opens the queue for sending messages, sets the label of a message, then sends the message to the queue.

To try this example using Microsoft® Visual Basic® (version 5.0), paste the code into the Code window of a form, and then run the example and click the form.

```
Dim qinfo As MSMQQueueInfo
Dim q As MSMQQueue
Dim msg As New MSMQMessage

Private Sub Form_Click()
    '**************************
    ' Create queue (no error
    ' handling if queue exists.
    '**************************
   Set qinfo = New MSMQQueueInfo
   qinfo.PathName = ".\LabelTest"
   qinfo.Label = "Test Queue"
   qinfo.Create
    '***************
    ' Open queue.
    '***************
   Set q = qinfo.Open(MQ_SEND_ACCESS, MQ_DENY_NONE)
    '***************
    ' Send Message.
    '***************
   msg.Label = "Test Message"
   msg.Body = "This is a test message with a string Body."
   msg.Send q


 q.Close

End Sub
```

See Also

Body, **Close**, **Create**, **FormatName**, **MSMQMessage**, **MSMQQueue**, **MSMQQueueInfo**, **Open**, **PathName**

MaxTimeToReachQueue

MSMQMessage

The **MaxTimeToReachQueue** property specifies a time limit (in seconds) for the message to reach the queue.

Quick Info

Type:	**Long**
Run time:	read/write

Syntax

object.**MaxTimeToReachQueue**

Syntax Element	Description
object	Message (**MSMQMessage**) object that represents the message.

Settings

Integer value (the default is LONG_LIVED).

Remarks

MaxTimeToReachQueue sets the message's *time-to-reach-queue* timer. For a discussion of message timers, see Message Timers. If the *time-to-reach-queue* timer expires before the message reaches its destination, MSMQ discards the message, sending it to the dead letter queue if the message's **Journal** property is set to MQMSG_DEADLETTER.

MSMQ can also send a return negative acknowledgment messages back to the sending application if the message does not arrive and the message's **Ack** property is set accordingly.

The default value LONG_LIVED is an enterprise-wide setting that can be adjusted by the MSMQ Administrator. Typically, LONG_LIVED is set to 90 days. Although this timer can be set to INFINITE, MSMQ automatically uses the LONG_LIVED value in its place.

Once a message arrives at the queue, **MaxTimeToReachQueue** can be used to find out how much time remains in the time-to-reach-queue timer. A value of 0 indicates the timer has expired.

MSMQ uses two message timers: time-to-reach-queue and time-to-be-received. If the time-to-be-received timer is set to a value less than the time-to-reach-queue timer, the time-to-be-received timer takes precedence over the time-to-reach-queue timer.

No matter what value **MaxTimeToReachQueue** is set to (even if set to 0), MSMQ always gives each message one chance to reach its destination if the queue is waiting for the message. If the queue is local, the message always reaches the queue.

MSMQ automatically uses the *time-to-reach-queue* timer of the first message when several messages are sent in a transaction. For information on transactions, see MSMQ Transactions.

When MSMQ creates an acknowledgment message, it always sets the message's *time-to-reach-queue* timer to LONG_LIVED.

Example

This example first creates and opens a queue for sending messages, sets the time-to-reach-queue timer for a message and sends it off to the queue, then reads the message in the queue and displays the time remaining in the timer.

To try this example using Microsoft Visual Basic (version 5.0), paste the code into the code window of a form, and then run the example and click the form.

```
Dim qinfo As MSMQQueueInfo
Dim q As MSMQQueue
Dim msgSent As New MSMQMessage
Dim msgReceived As MSMQMessage

Private Sub Form_Click()
    '****************************
    ' Create queue (no error
    ' handling if queue exists).
    '****************************
  Set qinfo = New MSMQQueueInfo
  qinfo.PathName = ".\TimerTest"
  qinfo.Label = "Test Queue"
  qinfo.Create
   '***************
   ' Open queue.
   '***************
  Set q = qinfo.Open(MQ_SEND_ACCESS, MQ_DENY_NONE)
   '***************
   ' Send Message.
   '***************
  msgSent.Label = "Test Message"
  msgSent.Body = "This message tests MaxTimeToReachQueue."
  msgSent.MaxTimeToReachQueue = 120
  msgSent.Send q

  MsgBox "The message was sent. Check the MSMQ Explorer to see the messages in the
queue."
  q.Close

  MsgBox "Click OK to continue"

    '***************
    ' Read Message.
    '***************
  Set q = qinfo.Open(MQ_PEEK_ACCESS, MQ_DENY_NONE)
  Set msgReceived = q.Peek
  MsgBox "MaxTimeToReachQueue = " + CStr(msgReceived.MaxTimeToReachQueue)

End Sub
```

See Also

Body, **Close**, **Create**, **Label**, **MSMQMessage**, **MSMQQueueInfo**, **MSMQQueue**, **Open**, **PathName**, **Send**

MaxTimeToReceive

MSMQMessage

The **MaxTimeToReceive** property specifies a time limit (in seconds) for the message to be retrieved from the target queue. This includes the time spent getting to the destination queue plus the time spent waiting in the queue before it is retrieved by an application.

Quick Info

Type:	**Long**
Run time:	read/write

Syntax

object.**MaxTimeToReceive**

Syntax Element	Description
object	Message (**MSMQMessage**) object that defines the message.

Settings

Integer value (the default is INFINITE).

Remarks

MaxTimeToReceive sets the message's *time-to-be-received* timer. For a discussion of message timers, see Message Timers. If the *time-to-be-received* timer expires before the message is removed from the queue, MSMQ discards the message, sending it to the dead letter queue if the message's **Journal** property is set to MQMSG_DEADLETTER.

MSMQ can also send a negative acknowledgment message back to the sending application if the message's **Ack** property is set accordingly and the message is not retrieved before the timer expires.

Once a message arrives at the queue, **MaxTimeToReceive** can be used to find out how much time remains in the *time-to-be-received* timer.

MSMQ uses two message timers: *time-to-reach-queue* and *time-to-be-received*. If the *time-to-be-received* timer is set to a value less than *the time-to-reach-queue* timer, the *time-to-be-received* timer takes precedence over the time-to-reach-queue timer.

MSMQ automatically uses the *time-to-be-received* timer of the first message when several messages are sent in a transaction. For information on transactions, see MSMQ Transactions.

When MSMQ creates an acknowledgment message, it always sets the message's *time-to-be-received* timer to INFINITE.

Example

This example first creates and opens a queue for sending messages, sets the time-to-receive timer for a message and sends it off to the queue, then reads the message in the queue and displays the time remaining in the timer.

To try this example using Microsoft Visual Basic (version 5.0), paste the code into the code window of a form, and then run the example and click the form.

```
Dim qinfo As MSMQQueueInfo
Dim q As MSMQQueue
Dim msgSent As New MSMQMessage
Dim msgReceived As MSMQMessage
```

```
Private Sub Form_Click()
    '****************************
    ' Create queue (no error
    ' handling if queue exists).
    '****************************
   Set qinfo = New MSMQQueueInfo
   qinfo.PathName = ".\TimerTest"
   qinfo.Label = "Test Queue"
   qinfo.Create
    '***************
    ' Open queue.
    '***************
   Set q = qinfo.Open(MQ_SEND_ACCESS, MQ_DENY_NONE)
    '***************
    ' Send Message.
    '***************
   msgSent.Label = "Test Message"
   msgSent.Body = "This message tests MaxTimeToReceive."
   msgSent.MaxTimeToReceive = 120
   msgSent.Send q

   MsgBox "The message was sent. Check the MSMQ Explorer to see the messages in the
queue."
   q.Close

   MsgBox "Click OK to continue"

    '***************
    ' Read Message.
    '***************
   Set q = qinfo.Open(MQ_PEEK_ACCESS, MQ_DENY_NONE)
   Set msgReceived = q.Peek
   MsgBox "MaxTimeToReceive = " + CStr(msgReceived.MaxTimeToReceive)

End Sub
```

See Also

Body, **Close**, **Create**, **Label**, **MSMQMessage**, **MSMQQueueInfo**, **MSMQQueue**, **Open**, **PathName**, **Send**

ModifyTime

MSMQQueueInfo

Read-only. The **ModifyTime** property indicates when the public queue's properties in the information store were last updated.

Quick Info

Type:	**Date Variant**
Run time:	read-only

Syntax

object.**ModifyTime**

Syntax Element	Description
object	Queue information (**MSMQQueueInfo**) object that defines the queue.

Return Values

Date when the queue properties were last updated (includes when the queue was created and the last time **Update** was called).

Remarks

To read this property, the application must first call the **MSMQQueueInfo** object's **Refresh** method. Although MSMQ updates MQIS when the queue is created and when **Update** is called, the **ModifyTime** property is not updated until **Refresh** is called.

The returned value for this property can be manipulated using standard Microsoft® Visual Basic® date and time functions such as **Date$**, and **Time$**. For descriptions of Visual Basic functions, see the Visual Basic documentation.

When **ModifyTime** is displayed, Visual Basic will automatically convert the parameter's value to the local system time and system date.

The **ModifyTime** property can be used when making a query (see example below).

Example

This example uses the ModifyTime and RelModifyTime parameters of LookupQueue to locate all the public queues that have been modified in the last 10 minutes. To locate the queues, MSMQ compares the date specified by the ModifyTime parameter with the date of each queue's ModifyTime property.

To try this example using Microsoft Visual Basic (version 5.0), paste the code into the Code window of a form, and then run the example and click the form. Run the example once, then wait ten minutes and run the example again with a new **PathName**. The queue created by the first pass will not be found on the second pass.

```
Dim query As New MSMQQuery
Dim qinfos As MSMQQueueInfos
Dim qinfoNew As MSMQQueueInfo
Dim qinfoDest As MSMQQueueInfo

Private Sub Form_Click()


  Set qinfoNew = New MSMQQueueInfo
  qinfoNew.PathName = ".\ModifyTest1"
```

```
    qinfoNew.Label = "Test Queue"
    qinfoNew.Create


    '**********************
    ' Locate public queues
    '**********************
    dateLast = DateAdd("n", -10, Now)
    Set qinfos = query.LookupQueue(ModifyTime:=dateLast, RelModifyTime:=REL_GT)
    qinfos.Reset

    '**********************************
    ' Display public queues modified in
    ' last 10 minutes.
    '**********************************
    Set qinfoDest = qinfos.Next
    cQueue = 0                  'Counter for number of queues found.

    While Not qinfoDest Is Nothing
        MsgBox "The properties of this queue (" + qinfoDest.FormatName + ") were
modified in the last ten minutes."
        cQueue = cQueue + 1
        Set qinfoDest = qinfos.Next
    Wend

    MsgBox "The total public queues found were: " + CStr(cQueue)

End Sub
```

See Also

Create, **FormatName**, **Label**, **LookupQueue**, **MSMQQuery**, **MSMQQueueInfo**, **MSMQQueueInfos**, **Next**, **PathName**, **Reset**

PathName

MSMQQueueInfo

Required. The **PathName** property specifies the MSMQ pathname of the queue. The MSMQ pathname specifies the name of the computer where the queue's message are stored, if the queue is public or private, and the name of the queue.

Quick Info

Type:	**String**
Run time:	read/write

Syntax

object.**PathName**

Syntax Element	Description
object	Queue information (**MSMQQueueInfo**) object that defines the queue.

Settings

String. The maximum length for the complete MSMQ pathname is MQ_MAX_Q_NAME_LEN (124 Unicode characters).

Remarks

The **PathName** property is the only property required when calling the MQQueueInfo object's **Create** method. An MQ_ERROR_PROPERTY_NOTALLOWED error is returned if any attempt is made to set this property after the queue is created.

For public queues, **PathName** includes the name of the computer where the queue's messages are stored, followed by the name of the queue (the MSMQ pathname of public queues is stored in MQIS name space). For private queues, add \PRIVATE$\ between the machine name of the local computer and the queue name (private queues can only be registered on the local computer). For a description of public and private queues, see Message Queues.

Here are three examples of MSMQ pathnames. The first two examples indicate two public queues (one on a local computer and the other on a remote computer), and the third example indicates a private queue.

```
"myMachine\myPublicQueue"
"otherMachine\otherPublicQueue"
"myMachine\Private$\myPrivateQueue"
```

As a shortcut, you can substitute a period "." for the local machine. So *myPublicQueue* and *myPrivateQueue* could be specified on the local machine as:

```
".\myPublicQueue"
".\Private$\myPrivateQueue"
```

Public queues are registered in MQIS, and private queues are registered on the local computer. Both types of queues exist until deleted explicitly.

Private queues are only created on the local computer. It is the application's responsibility to ensure that all queue names on the local computer are unique. If a queue name already exists when **Create** is called, MSMQ returns an MQ_ERROR_QUEUE_EXISTS error to the application.

To find out the MSMQ pathname of a queue, call the **MSMQQueueInfo** object's **Refresh** method.

Foreign public queues (queues located outside the MSMQ enterprise) are created in the same way as an MSMQ public queue. For foreign queues, the **PathName** property specifies the name of the foreign computer as it is defined in MQIS. For information on foreign computers, see MSMQ Connector Server.

Example

This example creates a private queue on the local machine, setting the queue's label to “Test Queue". To try this example using Microsoft Visual Basic (version 5.0), paste the code into the Code window of a form, and then run the example and click the form.

```
Dim qinfo As MSMQQueueInfo

Private Sub Form_Click()

  Set qinfo = New MSMQQueueInfo
  qinfo.PathName = ".\PRIVATE$\myqueue"
  qinfo.Label = "Test Queue"
  qinfo.Create

  MsgBox "Queue's Format name is: " + qinfo.FormatName

End Sub
```

See Also

Create, **FormatName**, **Label**, **MSMQQueueInfo**

Priority

MSMQMessage

The **Priority** property specifies the message's priority. A low number indicates a low priority.

Quick Info

Type:	**Long**
Run time:	read/write

Syntax

object.**Priority**

Syntax Element	Description
object	Message (**MSMQMessage**) object that represents the message.

Settings

Set **Priority** to an integer value between 7 and 0 (the default is 3).

Remarks

Message priority affects how MSMQ handles the message while it is in route, as well as where the message is placed in the queue. Higher priority messages are given preference during routing, and inserted towards the front of the queue. Messages with the same priority are placed in the queue according to their arrival time.

MSMQ automatically sets the priority level of transactional messages to 0: **Priority** is ignored by the transaction. For information on transactions, see MSMQ Transactions.

If the message is sent to a public queue, a second priority (the queue's **BasePriority** property) is added to **Priority** for routing purposes. However, the queue's base priority has no effect on how messages are placed in the queue.

Example

This example first creates and opens a queue for sending messages, then sets the priority level of two message and sends them off to the queue.

To try this example using Microsoft® Visual Basic® (version 5.0), paste the code into the Code window of a form, and then run the example and click the form.

```
Dim qinfo As MSMQQueueInfo
Dim q As MSMQQueue
Dim msg1 As New MSMQMessage
Dim msg2 As New MSMQMessage

Private Sub Form_Click()
    '**************************
    ' Create queue (no error
    ' handling if queue exists.
    '**************************
  Set qinfo = New MSMQQueueInfo
  qinfo.PathName = ".\PriorityTest"
  qinfo.Label = "Test Queue"
  qinfo.Create
    '**************
    ' Open queue.
```

```
    '**************
   Set q = qinfo.Open(MQ_SEND_ACCESS, MQ_DENY_NONE)
    '**************
    ' Send Messages.
    '**************
   msg1.Label = "Test Message1"
   msg1.Body = "This is a test message with a string Body."
   msg1.Priority = 0

   msg2.Label = "Test Message2"
   msg2.Body = "This is a test message with a string Body."
   msg2.Priority = 7

   msg1.Send q
   msg2.Send q

   MsgBox "Both messages were sent. Check the MSMQ Explorer to see the messages in
the queue."

 q.Close

End Sub
```

See Also

Body, **Close**, **Create**, **Label**, **MSMQMessage**, **MSMQQueue**, **MSMQQueueInfo**, **Open**, **PathName**, **Send**

PrivLevel

MSMQQueueInfo **MSMQMessage**

Optional. The **PrivLevel** property specifies the privacy level of a queue or message.

For queues, this property specifies whether or not the queue accepts private (encrypted) messages, non-private messages, or both.

For messages, this property specifies whether or not the message is private (encrypted).

Quick Info

Type:	**Long**
Run time:	read/write

Syntax

object.**PrivLevel**

Syntax Element	Description
object	For queues, the queue information (**MSMQQueueInfo**) object that defines the queue. For messages, the message (**MSMQMessage**) object that defines the message.

Settings

For queues, set **PrivLevel** to one of the following values:

MQ_PRIV_LEVEL_NONE
: The queue accepts only non-private (clear) messages.

MQ_PRIV_LEVEL_BODY
: The queue accepts only private (encrypted) messages.

MQ_PRIV_LEVEL_OPTIONAL
: The default. The queue does not force privacy. It accepts private (encrypted) messages and non-private (clear) messages.

For messages, set **PrivLevel** to one of the following values:

MQMSG_PRIV_LEVEL_NONE
: The default. The message is a non-private (clear) message.

MQMSG_PRIV_LEVEL_BODY
: The message is private (encrypted) message.

Remarks

Queue Privacy Level
: The application can set the privacy level of queues and messages. If the privacy level of the message does not correspond to the privacy level of the queue, the message is rejected by the queue, and, if the sending application requested a negative acknowledgment message when it sent the message, MQMSG_CLASS_BAD_ENCRYPTION is returned to the sending application to indicate the message was rejected.

  To specify the privacy level when creating the queue, set **PrivLevel** and call the **MSMQQueueInfo** object's **Create** method.

  To reset the privacy level of a queue after the queue is created, set **PrivLevel** to a new level and, if

the queue is open, call the **MSMQQueueInfo** object's **Update** method. If the queue is not open do not call **Update**, the queue's properties are updated automatically when the queue is opened.

To find out the privacy level of a queue, call the **MSMQQueueInfo** object's **Refresh** method.

Message Privacy Level

MSMQ can send private messages throughout the MSMQ enterprise. When sending a private message the source Queue manager encrypts the body of the message and the target queue manager decrypts the message body. For a discussion of private messages, see Private Messages.

When encrypting and decrypting messages, MSMQ uses the algorithm specified in **EncryptAlgorithm**.

For a complete example of sending a private message (including setting the privacy level of a queue), see Sending Private Messages.

Example: Setting the privacy level of a queue

This example creates a private queue on the local machine, setting the queue's privacy level to MQ_PRIV_LEVEL_OPTIONAL. To try this example using Microsoft Visual Basic (version 5.0), paste the code into the Code window of a form, and then run the example and click the form.

```
Dim qinfo As MSMQQueueInfo

Private Sub Form_Click()

  Set qinfo = New MSMQQueueInfo
  qinfo.PathName = ".\PRIVATE$\PrivacyLevelTest"
  qinfo.Label = "Test Queue"
  qinfo.PrivLevel = MQ_PRIV_LEVEL_OPTIONAL
  qinfo.Create

  MsgBox "Queue's Format name is: " + qinfo.FormatName

End Sub
```

Example: Setting the privacy level of a message

This example opens a queue that can only accept private messages, then sends a private message to the queue.

To try this example using Microsoft Visual Basic (version 5.0), paste the code into the Code window of a form, and then run the example and click the form.

```
Dim qinfo As MSMQQueueInfo
Dim q As MSMQQueue
Dim msg As New MSMQMessage

Private Sub Form_Click()
  '**********************
  ' Create queue
  '**********************
  Set qinfo = New MSMQQueueInfo
  qinfo.PathName = ".\PrivacyTest"
  qinfo.Label = "Test Queue"
  qinfo.PrivLevel = MQ_PRIV_LEVEL_BODY
  qinfo.Create
  '**********************
  ' Open queue.
  '**********************
```

```
    Set q = qinfo.Open(MQ_SEND_ACCESS, MQ_DENY_NONE)
    '**********************
    ' Send message.
    '**********************
    msg.Label = "Test Message"
    msg.Body = "This is a private message."
    msg.PrivLevel = MQMSG_PRIV_LEVEL_BODY

    msg.Send q

    q.Close

End Sub
```

See Also

Body, **Create**, **Label**, **MSMQMessage**, **MSMQQueue**, **MSMQQueueInfo**, **Open**, **PathName**, **Refresh**, **Send**, **Update**

QueueGuid

MSMQQueueInfo

Read-only. The **QueueGuid** property identifies the public queue associated with the **MSMQQueueInfo** object. The queue identifier is created by MSMQ when the queue is created.

Quick Info

Type:	**GUID**
Run time:	read-only

Syntax

object.**QueueGuid**

Syntax Element	Description
object	Queue information (**MSMQQueueInfo**) object that defines the queue.

Return Values

GUID of public queue.

Remarks

To find out the queue identifier of a queue, the application must first call **Refresh**. Although MSMQ creates the queue identifier when the queue is created, it does not update **quidQueue** until **Refresh** is called.

The **QueueGuid** property identifies the queue defined by the **MSMQQueueInfo** object. It does not identify an open instance of the queue.

Example

This example creates a public queue and then uses **Refresh** to display the queue's identifier. To try this example using Microsoft Visual Basic (version 5.0), paste the code into the Code window of a form that has a single text box, and then run the example and click the form.

```
Dim qinfo As MSMQQueueInfo

Private Sub Form_Click()

  Set qinfo = New MSMQQueueInfo
  qinfo.PathName = ".\guidQueueTest"
  qinfo.Label = "Test Queue"
  qinfo.Create

  qinfo.Refresh             'Required to update QueueGuid
  Text1.Text = "quidQueue = " + CStr(qinfo.QueueGuid)

End Sub
```

See Also

Create, **Label**, **MSMQQueueInfo**, **PathName**, **Refresh**

QueueInfo

MSMQQueue

Read-only. The **QueueInfo** property retrieves the initial settings of the **MSMQQueueInfo** object used to open the queue.

Quick Info

Type:	**MSMQQueueInfo**
Run time:	read-only

Syntax

*Set object2=object1.***QueueInfo**

Syntax Element	Description
object1	Queue (**MSMQQueue**) object that represents the open instance of the queue.
object2	Queue information (**MSMQQueueInfo**) object that defined the queue.

Return Values

MSMQQueueInfo object.

Remarks

The **QueueInfo** property is used to determine the original settings used to create the queue. For example, it can be used to determine the original label of a queue when it has been changed.

Example

This example opens a queue for sending messages, changes the label of the queue, then retrieves the original label using **QueueInfo**. To try this example using Microsoft® Visual Basic® (version 5.0), paste the code into the Code window of a form, and then run the example and click the form.

```
Dim qinfo As MSMQQueueInfo
Dim q As MSMQQueue

Private Sub Form_Click()

  Set qinfo = New MSMQQueueInfo
  qinfo.PathName = ".\queueinfo6"
  qinfo.Label = "Test Queue"
  qinfo.Create

  Set q = qinfo.Open(MQ_SEND_ACCESS, MQ_DENY_NONE)
  MsgBox "Label is: " + qinfo.Label

  qinfo.Label = "New Label"
  MsgBox "Label is: " + qinfo.Label

  q.Close

  Set qinfo = q.QueueInfo
  MsgBox "Label is: " + qinfo.Label

End Sub
```

See Also

Close, **Create**, **QueueGuid**, **Open**, **Label**, **PathName**

Quota

MSMQQueueInfo

Optional. The **Quota** property specifies the maximum size (in kilobytes) of the queue.

Quick Info

Type:	**Long**
Run time:	read/write

Syntax

object.**Quota**

Syntax Element	Description
object	Queue information (**MSMQQueueInfo**) object that defines the queue.

Settings

Long integer (default is INFINITE). Maximum size is based on the memory available in the computer where the queue's messages are stored.

Remarks

Quota is typically set when the queue is created.

To set the quota of a queue, set **Quota** and call the **MSMQQueueInfo** object's **Create** method.

To reset the quota of a queue after the queue is created, set **Quota** to a new level and, if the queue is open, call the **MSMQQueueInfo** object's **Update** method. If the queue is not open do not call **Update**, the queue's properties are updated automatically when the queue is opened.

To find the quota of a queue, call the **MSMQQueueInfo** object's **Refresh** method.

Example

This example creates a private queue on the local machine, setting the queue's quota to 10K. To try this example using Microsoft Visual Basic (version 5.0), paste the code into the Code window of a form, and then run the example and click the form.

```
Dim qinfo As MSMQQueueInfo

Private Sub Form_Click()

  Set qinfo = New MSMQQueueInfo
  qinfo.PathName = ".\myqueue"
  qinfo.Label = "Test Queue"
  qinfo.Quota = 10
  qinfo.Create

  MsgBox "Queue's Format name is: " + qinfo.FormatName

End Sub
```

See Also

Create, **FormatName**, **Label**, **MSMQQueueInfo**, **PathName**, **Update**

ResponseQueueInfo

MSMQMessage

The **ResponseQueueInfo** property specifies a queue for receiving response messages from the target application.

Quick Info

Type:	**MSMQQueueInfo**
Run time:	read/write

Syntax

set object1.**ResponseQueueInfo =** *object2*

Syntax Element	Description
object1	Message (**MSMQMessage**) object that represents the message.
object2	Queue information (**MSMQQueueInfo**) object that represents the response queue.

Settings

MSMQQueueInfo object.

Remarks

ResponseQueueInfo is used to send the format name of another queue to the receiving application. Typically, this is done so that the receiving application can send response messages back to the sending application. For information on response queues, see Response Queues.

Note The **MSMQQueueInfo** object of a private queue (which would be inaccessible otherwise) can also be sent using **queuinfoResponse**.

Messages returned to the queue are application defined. The application must define what is in the messages, as well as what is to be done when a message is received.

For a complete example of sending a message that requests a response plus sending the response, see Sending Messages that Request a Response.

Example

This example locates a response and destination queue (creating them if needed), sends a message to the destination queue, then retrieves the message and sends a response message back to the response queue. To coordinate between the two messages, the correlation identifier of the response message (**CorrelationId**) is set to the message identifier of the original message.

To try this example using Microsoft Visual Basic (version 5.0), paste the code into the code window of a form, and then run the example and click the form.

```
Dim query As New MSMQQuery
Dim qinfos As MSMQQueueInfos
Dim qinfoResp, As MSMQQueueInfo
Dim qinfoDest As MSMQQueueInfo
Dim qRead As New MSMQQueue
Dim qResp As MSMQQueue
Dim msgSent As New MSMQMessage
Dim msgRead As MSMQMessage
Dim msgResp As New MSMQMessage
```

```
Private Sub Form_Click()

    '**********************************
    ' Locate response queue (create one
    ' if one doesn't exist).
    '**********************************
    Set qinfos = query.LookupQueue(Label:="Response Queue")
    qinfos.Reset
    Set qinfoResp = qinfos.Next
    If qinfoResp Is Nothing Then
        Set qinfoResp = New MSMQQueueInfo
        qinfoResp.PathName = ".\RespQueue"
        qinfoResp.Label = "Response Queue"
        qinfoResp.Create
    End If
    '**********************************
    ' Locate destination queue
    '(create one if one doesn't exist).
    '**********************************
    Set qinfos = query.LookupQueue(Label:="Destination Queue")
    qinfos.Reset
    Set qinfoDest = qinfos.Next
    If qinfoDest Is Nothing Then
        Set qinfoDest = New MSMQQueueInfo
        qinfoDest.PathName = ".\DestQueue"
        qinfoDest.Label = "Destination Queue"
        qinfoDest.Create
    End If

    '*************************
    ' Open destination queue.
    '*************************
    Set q = qinfoDest.Open(MQ_SEND_ACCESS, MQ_DENY_NONE)

    '***************
    ' Send Message.
    '***************
    msgSent.Label = "Test Message"
    msgSent.Body = "This message tests the response queue."
    Set msgSent.ResponseQueueInfo = qinfoResp
    msgSent.Send q

    MsgBox "The message was sent to the following queue: " + qinfoDest.QueueGuid + ".
Check the MSMQ Explorer to see the message in the queue."
    q.Close

    '************************************
    ' Read the message in the destination
    ' queue and send response message if
    ' one is requested.
    '************************************
    Set q = qinfoDest.Open(MQ_RECEIVE_ACCESS, MQ_DENY_NONE)
    Set msgRead = q.Receive
```

```
    Set qResp = msgRead.ResponseQueueInfo.Open(MQ_SEND_ACCESS, MQ_DENY_NONE)
    msgResp.Label = "Response Message"
    msgResp.Body = "This is a response message"
    msgResp.CorrelationId = msgRead.Id
    msgResp.Send qResp
    MsgBox "The response message was sent to the following queue: " +
msgRead.ResponseQueueInfo.QueueGuid


End Sub
```

See Also

Body, **Close**, **Create**, **CorrelationId**, **Id**, **Label**, **LookupQueue**, **MSMQMessage**, **MSMQQuery**, **MSMQQueue**, **MSMQQueueInfo**, **MSMQQueueInfos**, **Next**, **Open**, **PathName**, **Receive**, **Reset**, **Send**

SenderCertificate

MSMQMessage

The **SenderCertificate** property provides an array of bytes that represents the security certificate. The security certificate is used to authenticate messages.

Quick Info

Type:	**Variant**
Run time:	read/write

Syntax

object.**SenderCertificate**

Syntax Element	Description
object	Message (**MSMQMessage**) object that defines the message.

Settings

Security certificate (internal or external).

Remarks

The sending application can specify an external certificate obtained from a certificate authority, or use the internal certificate provided by MSMQ.

When using external certificates, the receiving application can use all the information in the certificate to verify who sent the message. When using an internal certificate, this property is not useful to the receiving application.

There are two ways to specify the security information provided by a certificate. If the sending application is only going to use a certificate once, it should provide the complete certificate using **SenderCertificate**. If the sending application is going to use the same certificate over and over, it should call **AttachCurrentSecurityContext**.**AttachCurrentSecurityContext** retrieves and caches the needed information using a single call, then automatically passes the information along with the message when it is sent.

For information on using an external certificate, see Authenticating Messages Using an External Certificate.

For information on using an internal certificate, see Authenticating Messages Using an Internal Certificate.

SenderId

MSMQMessage

The **SenderId** property is an array of bytes that represent the identifier of the sending application. MSMQ sets this property when the message is sent.

Quick Info

Type:	**Variant**
Run time:	read-only

Syntax

object.**SenderId**

Syntax Element	Description
object	Message (**MSMQMessage**) object that defines the message.

Return Values

An array of bytes generated by MSMQ.

Remarks

When a message is sent, MSMQ attaches **SenderId** to the message when an MSMQ sender identifier is found for the user. The **SenderId** property is primarily used by MSMQ security to identify who sent the message.

A second property, **SenderIdType**, is also attached to the message when the sender identifier is found. This second property indicates what type of identifier was found (currently, the only type of sender identifier available to MSMQ is an SID.

The receiving applications can look at **SenderId** to verify who sent a message.

SenderIDType

MSMQMessage

The **SenderIDType** property specifies the type of sender identifier found by MSMQ. Currently, the only type of sender identifier available to MSMQ is an SID.

Quick Info

Type:	**Long**
Run time:	read-write

Syntax

object.**SenderIDType**

Syntax Element	Description
object	Message (**MSMQMessage**) object that defines the message.

Settings

MSMQ sets **SenderIDType** to one of the following:

MQMSG_SENDERID_TYPE_NONE
SenderID is not attached to the message.

MQMSG_SENDERID_TYPE_SID
The default. The **SenderID** property contains an SID for the user sending the message.

Remarks

If the sending application does not want MSMQ to send a sender identifier with a message, it can specify MQMSG_SENDERID_TYPE_NONE when sending the message. This suppresses the message's **SenderID** property.

The receiving application can use **SenderIDType** to determine what type of sender identifier was attached to the message.

An SID of a local user (a user not logged into a Windows NT domain) is only valid locally. Even if a local user specifies an SID, it is not sent with the message.

SentTime

MSMQMessage

Read-only. The **SentTime** property indicates when a message was sent.

Quick Info

Type:	**Date Variant**
Run time:	read-only

Syntax

object.**SentTime**

Syntax Element	Description
object	Message (**MSMQMessage**) object that defines the message.

Return Values

Date message was sent.

Remarks

The returned value for this property can be manipulated using standard Microsoft® Visual Basic® date and time functions such as **Date$**, and **Time$**. For descriptions of Visual Basic functions, see the Visual Basic documentation.

When **SentTime** is displayed, Visual Basic will automatically convert the parameter's value to the local system time and system date.

Example

This example locates a destination queue (creating one if one does not exist), sends a message to the queue, then reads the message and displays when the message was sent.

To try this example using Microsoft Visual Basic (version 5.0), paste the code into the Code window of a form, and then run the example and click the form.

```
Dim query As New MSMQQuery
Dim qinfos As MSMQQueueInfos
Dim qinfoDest As MSMQQueueInfo
Dim qDest As MSMQQueue
Dim msgSent As New MSMQMessage
Dim msgRead As MSMQMessage


Private Sub Form_Click()


    '**********************************
    ' Locate destination queue
    '(create one if one doesn't exist).
    '**********************************
    Set qinfos = query.LookupQueue(Label:="Destination Queue")
    qinfos.Reset
    Set qinfoDest = qinfos.Next
    If qinfoDest Is Nothing Then
        Set qinfoDest = New MSMQQueueInfo
        qinfoDest.PathName = ".\DestQueue"
```

```
    qinfoDest.Label = "Destination Queue"
    qinfoDest.Create
End If

'**************
' Send Message.
'**************
Set qDest = qinfoDest.Open(MQ_SEND_ACCESS, MQ_DENY_NONE)
msgSent.Label = "Test Message"
msgSent.Body = "This message tests the message timers."
msgSent.Send qDest
qDest.Close

'****************************************
' Remove the message from destination queue
' and display when the message was sent.
'****************************************
Set qDest = qinfoDest.Open(MQ_RECEIVE_ACCESS, MQ_DENY_NONE)
Set msgRead = qDest.Receive

MsgBox "The message was sent at: " + CStr(msgRead.SentTime)

End Sub
```

See Also

Body, **Close**, **Create**, **Label**, **LookupQueue**, **MSMQMessage**, **MSMQQuery**, **MSMQQueue**, **MSMQQueueInfo**, **MSMQQueueInfos**, **Next**, **Open**, **PathName**, **Receive**, **Send**

ServiceTypeGuid

MSMQQueueInfo

Optional. The **ServiceTypeGuid** property specifies the type of service provided by the queue.

Quick Info

Type:	**GUID**
Run time:	read/write

Syntax

object.**ServiceTypeGuid**

Syntax Element	Description
object	Queue information (**MSMQQueueInfo**) object that defines queue.

Settings

GUID. Pre-defined or application generated.

Remarks

The queue's service type can be used to identify the queue.

It is recommended that the service type of the queue be specified when the queue is created. In most cases, the service type of the queue can be defined by the application; however, some queues used by MSMQ require a specific MSMQ-defined service type. For example, input queues used by the MSMQ MAPI Transport Provider have a specific MSMQ-defined service type.

Note To generate a GUID, run the UUIDGEN.EXE program provided by MSDN. (For information about UUIDGEN.EXE, see the Microsoft Platform SDK.)

The queue's service type can also be used to locate public queues registered in MQIS (see **LookupQueue**).

To specify the queue's service type, set **ServiceTypeGuid** and call the MSMQQueueInfo object's **Create** method.

To reset the service type of a queue after the queue is created, set **Authenticate** to a new GUID and, if the queue is open, call the **MSMQQueueInfo** object's **Update** method. If the queue is not open do not call **Update**, the queue's properties are updated automatically when the queue is opened.

To find the service type of a queue, call the **MSMQQueueInfo** object's **Refresh** method.

Example

This example creates a private queue on the local machine, setting the queue's service type to an application-defined GUID. To try this example using Microsoft® Visual Basic® (version 5.0), paste the code into the Code window of a form, and then run the example and click the form.

```
Dim qinfo As MSMQQueueInfo

Private Sub Form_Click()

  Set qinfo = New MSMQQueueInfo
  qinfo.PathName = ".\PRIVATE$\myqueue"
  qinfo.Label = "Test Queue"
  qinfo.ServiceTypeGuid = "{bed40680-b773-11d0-8b96-00aa0062c8e9}"
```

```
    qinfo.Create

    MsgBox "Queue's Format name is: " + qinfo.FormatName

End Sub
```

See Also

Create, **FormatName**, **Label**, **LookupQueue**, **PathName**

ShareMode

MSMQQueue

Read-only. The **ShareMode** property indicates the share mode of the queue.

Quick Info

Type:	**Long**
Run time:	read-only

Syntax

object.**ShareMode**

Syntax Element	Description
object	Queue (**MSMQQueue**) object that represents the open instance of the queue.

Return Values

The **ShareMode** property returns one of the following values:

MQ_DENY_NONE
: The queue is available to everyone for sending, peeking, or retrieving messages. This is always returned if the queue was opened with *Access* set to MQ_PEEK_ACCESS or MQ_SEND_ACCESS.

MQ_DENY_RECEIVE_SHARE
: Messages can only be retrieved by this process. This value is only returned if the queue was opened with *Access* set to MQ_RECEIVE_ACCESS.

Remarks

The **ShareMode** property returns the share mode of the queue when it was last opened, regardless if the queue is currently opened or closed.

When **ShareMode** returns MQ_DENY_NONE, several users can be using the queue at the same time.

Example

This example opens a queue for sending messages, then uses the value of **ShareMode** to test who can use the queue (with what share mode). To try this example using Microsoft Visual Basic (version 5.0), paste the code into the Code window of a form, and then run the example and click the form.

```
Dim qinfo As MSMQQueueInfo
Dim q As MSMQQueue

Private Sub Form_Click()

  Set qinfo = New MSMQQueueInfo
  qinfo.PathName = ".\lShareModeTest"
  qinfo.Label = "Test Queue"
  qinfo.Create

  Set q = qinfo.Open(MQ_SEND_ACCESS, MQ_DENY_NONE)

  Select Case q.ShareMode
     Case MQ_DENY_NONE
          MsgBox "The queue is open for multiple users."
     Case MQ_DENY_RECEIVE_SHARE
          MsgBox "The queue is only open for this process."
```

```
    Case Else
        MsgBox "Not a valid return value!"
  End Select

  q.Close

End Sub
```

See Also

Close, **Create**, **Label**, **MSMQQueue**, **MSMQQueueInfo**, **Open**, **PathName**

SourceMachineGuid

MSMQMessage

Read-only. The **SourceMachineGuid** specifies the source machine used to send the message.

Quick Info

Type:	**GUID**
Run time:	read-only

Syntax

object.**SourceMachineGuid**

Syntax Element	Description
object	Message (**MSMQMessage**) object that defines the message.

Return Values

GUID of the source machine.

Trace

MSMQMessage

The **Trace** property specifies where report messages are sent when tracing a message.

Quick Info

Type:	**Long**
Run time:	read/write

Syntax

object.**Trace**

Syntax Element	Description
object	Message (**MSMQMessage**) object that defines the message.

Settings

The **Trace** property can be set to one of the following values:

MQMSG_SEND_ROUTE_TO_REPORT_QUEUE
: Each hop made by the original message generates a report that is recorded in a report message. The report elements include source Queue Manager, message identifier, target, time, and next hop. The report message is sent to the report queue specified by the source Queue Manager.

MQMSG_TRACE_NONE
: The default. No tracing for this message.

Remarks

If **Trace** is specified but the report queue is not defined by the MSMQ Administrator for the message's source Queue Manager, this property is ignored.

For a description of report queues and messages, see Report Queues and Report Messages.

Transaction

MSMQTransaction

Read-only. The **Transaction** property represents the underlying transaction object used by this transaction.

Quick Info

Type:	Long
Run time:	read-only

Syntax

*object.***Transaction**

Syntax Element	Description
object	The **MSMQTransaction** object that represents the transaction.

Return Values

Transaction object.

ActiveX Events

MSMQ uses a very limited number of events: **Arrived** and **ArrivedError**. Both events are fired by the **MSMQEvent** component in response to a message or error arriving at a queue. The **MSMQEvent** object is triggered whenever a message or error arrives at a queue and the queue object's **EnableNotification** method has been called.

Arrived

MSMQEvent

The **Arrived** event is fired when **EnableNotification** has been called and some message is found in the queue.

Syntax

object_Arrived(byval Queue as Object)

Syntax Element	Description
object	Instance of event (**MSMQEvent**) object used for the event handler.
Queue	Queue (**MSMQQueue**) object that represents an open instance of the queue where the message arrived.

Remarks

The **Arrived** event is fired by an instance of the **MSMQEvent** object. Every queue that has called **EnableNotification** triggers an instance of the **MSMQEvent** object when a message arrives in the queue (this includes existing messages in the queue when **EnableNotification** is first called).

There is no association between an **Arrived** event and a specific message. The arrived event only means that some message arrived in the queue.

Although an **Arrived** event is fired for each message, there is no guarantee that the message that triggered the event will still be there when the event handler tries to use the message. Queues are dynamic, and another application may have already removed the message that triggered the **Arrived** event.

EnableNotification must be explicitly reset after each **Arrived** event is fired.

Example

This example assumes that queues q1 and q2 are open and that they are not being shared by another application. **EnableNotification** is called, and then the label of the first message in the queue (if messages are already in the queue), or the first message sent to the queue, is displayed.

```
Dim WithEvents qevents as MSMQEvent
Dim q1 as MSMQQueue
Dim q2 as MSMQQueue

Sub Enable
    'Assuming q1 and q2 are open.
    q1.EnableNotification
    q2.EnableNotification
End Sub

Sub event_Arrived (byval q as Object)
    'Generic event handler
    Dim qArrive as MSMQQueue
    Set qArrive = q                    'Cast Object reference to MSMQQueue

    'This example assumes the message has not been removed
    'by another application.
    msgbox “Message in following queue: “ + qArrive.QueueInfo.Label
    qArrive.EnableNotification Event          'Reenables notification.
```

```
End Sub
```

See Also

EnableNotification, **MSMQQueue**

ArrivedError

MSMQEvent

The **ArrivedError** event is fired when **EnableNotification** has been called and an error is generated.

Syntax

object_ArrivedError(byval Queue as Object, byval ErrorCode as Long)

Syntax Element	Description
object	Instance of event (**MSMQEvent**) object used for the event handler.
Queue	Queue (**MSMQQueue**) object that represents an open instance of the queue where the message arrived.
ErrorCode	Error code returned by **MQReceiveMessage**.

Remarks

An **ArrivedError** event can be triggered by the message's timeout timer expiring or by an error occurring while the queue is receiving the message. The message's timeout timer can be used when the application only wants to wait a specific amount of time to receive its messages.

This event applies only to messages failing to arrive at the queue; it is not associated with any errors generated while the application is trying to read messages in the queue.

MSMQ Mail ActiveX Components

MSMQ provides ActiveX components that support composing and parsing the body of MSMQ mail messages, which are used to communicate with e-mail based applications through the MSMQ mail services.

Note The MSMQ Mail SDK must be installed to use any of the MSMQ Mail ActiveX components.

For information on MSMQ mail services (MAPI Transport Provider and Exchange Connector), see MSMQ Mail Services.

MSMQ Mail Objects

The MSMQ Mail objects include:

- **MSMQMailEMail**
- **MSMQMailFormData**
- **MSMQMailFormField**
- **MSMQMailFormFieldList**
- **MSMQMailRecipient**
- **MSMQMailRecipientList**
- **MSMQMailTextMessageData**
- **MSMQMailTnefData**
- **MSMQMailDeliveryReportData**
- **MSMQMailNonDeliveryReportData**

MSMQMailEMail

Properties **Methods**

The **MSMQMailEMail** object represents an e-mail message. The e-mail message can be a form with several fields, a text message with a single text body, a MAPI TNEF message, or a delivery or non-delivery report.

The following properties define the e-mail message.

Property	What it does
ContentType	Specifies the type of e-mail: form, text message, MAPI TNEF message, delivery report, or non-delivery report.
DeliveryReportData	Defines a delivery report (**MSMQMailDeliveryReportData** object).
DestinationQueueLabels	Provides the label of each destination queue needed to send the MSMQ mail message (read-only).
FormData	Defines an e-mail form (**MSMQMailFormData** object).
NonDeliveryReportData	Defines a non-delivery report (**MSMQMailNonDeliveryReportData** object).
Recipients	Specifies who receives the e-mail (**MSMQMailRecipientList** object).
RequestDeliveryReport	Specifies whether a delivery report should be returned when the message is delivered to a recipient.
RequestNonDeliveryReport	Specifies whether a non-delivery report should be returned when the message is not delivered to a recipient.
Sender	Specifies who sent the e-mail(**MSMQMailRecipient** object).
Subject	Specifies the subject of the e-mail.
SubmissionTime	Indicates when message was sent.
TextMessageData	Defines a single-body text message (**MSMQMailTextMessageData** object).
TnefData	Defines a MAPI TNEF message (**MSMQMailTnefData** object).

The **MSMQMailEMail** object's methods provide a means to compose and parse an e-mail message.

MSMQMailFormData

Properties

The **MSMQMailFormData** object represents an e-mail form.

The following properties define the form.

Property	What it does
FormFields	Defines the fields of the form (**MSMQMailFormFieldList** object).
Name	Specifies the name of the form.

The **MSMQMailFormData** object has no methods.

MSMQMailFormFieldList

Properties **Methods**

The **MSMQMailFormFieldList** object represents a list of all the fields of a specific e-mail form.

It references a collection of **MSMQMailFormField** objects.

The following properties define a field list.

Property	What it does
Count	Number of fields in a form (read-only).
Item	Defines a field in the list (read-only).

The **MSMQMailFormFieldList** object has methods for adding and removing items (**MSMQMailFormField** objects) from the list.

MSMQMailFormField

Properties

The **MSMQMailFormField** object represents a field of an e-mail form.

Every **MSMQMailFormField** object is an item of an **MSMQMailFormFieldList** object.

The following properties define the field.

Property	What it does
Name	Specifies the name of the field.
Value	Specifies the value of the field.

The **MSMQMailFormField** object has no methods.

MSMQMailRecipientList

Properties **Methods**

The **MSMQMailRecipientList** object represents a list of e-mail recipients.

It contains references to one of more **MSMQMailRecipient** object.

The following properties define a recipient list.

Property	What it does
Count	Number of recipients that will receive the e-mail (read-only).
Item	Defines a recipient in the list (read-only).

The **MSMQMailRecipientList** object has methods for adding and removing items (**MSMQMailRecipient** objects) from the list.

MSMQMailRecipient

Properties

The **MSMQMailRecipient** object represents an e-mail recipient.

It can be included in a list of recipients (**MSMQMailRecipientList** object) to indicate who is receiving the mail, or it can be used to indicate who sent the mail (**formdata**).

The following properties define a recipient.

Property	What it does
Address	Specifies the address of the recipient.
Name	Specifies the name of the recipient.
RecipientType	Specifies how the e-mail is sent to the recipient.
NonDeliveryReason	Optional. Used in non-delivery reports to specify why the recipient did not receive the e-mail.
DeliveryTime	Optional. Used in delivery reports to specify when the message arrived

The **MSMQMailRecipient** object has no methods.

MSMQMailTextMessageData

Properties

The **MSMQMailTextMessageData** object represents an e-mail text message.

This object uses a single property (**Text**) to define the message.

The **MSMQMailTextMessageData** object uses no methods.

MSMQMailTnefData

Properties

The **MSMQMailTnefData** object represents an e-mail in TNEF format.

TNEF is a MAPI internal format that encapsulates the MAPI properties, and is used by the MSMQ Mail services (the MSMQ Exchange Connector and the MSMQ MAPI Transport) to send mail to recipients who are defined as rich-text recipients.

This object uses a single property (**Data**) to define the TNEF data.

The **MSMQMailTnefData** object uses no methods.

MSMQMailDeliveryReportData

Properties

The **MSMQMailDeliveryReportData** object represents a delivery report e-mail.

The following properties define the delivery report.

Property	What it does
DeliveredRecipients	Defines the recipients to whom the original mail was delivered. (**MSMQMailRecipientList** object). The information for each recipient in this list contains the optional recipient property **DeliveryTime** which specifies the delivery time of the original mail to this recipient.
OriginalSubject	Specifies the subject of the original mail.
OriginalSubmissionTime	Specifies the submission time of the original mail.

The **MSMQMailDeliveryReportData** object has no methods.

MSMQMailNonDeliveryReportData

Properties

The **MSMQMailNonDeliveryReportData** object represents a non-delivery report e-mail.

The following properties define the non-delivery report.

Property	What it does
NonDeliveredRecipients	Defines the recipients to whom the original mail was not delivered. (**MSMQMailRecipientList** object). The information for each recipient in this list contains the optional recipient property **NonDeliveryReason** which specifies the reason why the original e-mail was not delivered to this recipient.
OriginalEMail	Specifies the original e-mail. (**MSMQMailEMail** object)

The **MSMQMailNonDeliveryReportData** object has no methods.

MSMQ Mail ActiveX Methods

The following topics describe the methods associated with the MSMQ Mail ActiveX components provided by MSMQ Mail SDK.

Note The MSMQ Mail SDK must be installed to use any of the MSMQ Mail ActiveX components.

The methods of the MSMQ Mail objects include:

MSMQMailEMail

ComposeBody

ParseBody

MSMQMailFormFieldList

Add

Remove

MSMQMailRecipientList

Add

Remove

Add

MSMQMailFormFieldList **MSMQMailRecipientList**

The **Add** method adds a recipient to the recipient list (**MSMQMailRecipientList**)or a field to a form field list (**MSMQMailFormFieldList**).

Syntax for Adding Recipient

object1.object2.**Add** *Name*, *Address*, *Type*[, *DeliveryTime*][, *NonDeliveryReason*][, *Key*]

Syntax Element	Description
object1	E-mail message (**MSMQMailEMail**) object that defines the e-mail message.
object2	Recipient list object (**MSMQMailRecipientList**) that represents the list of recipients.
Name	String representation of recipient.
Address	String representation of the e-mail recipient's address. Use one of the following formats (see Remarks for examples). When the e-mail is being sent to another MSMQ application, the recipient's address is the label (**Label**) of the application input queue *Adress:="MSMQQueueLabel"* When the e-mail is being sent to an Exchange user via the MSMQ Exchange Connector, the address includes the e-mail alias of the Exchange user plus the label of the Exchange Connector's *server input queue*. *Address:="UserName@ServerInputQueueLabel"* When e-mail is being sent to the MAPI Transport Provider, the address is the label of the MAPI user input queue. *Adress:="UserInputQueueLabel"*
Type	The *Type* parameter can have any one of the following values: MSMQMAIL_RECIPIENT_TO: Default. The recipient is the primary recipient of the e-mail. MSMQMAIL_RECIPIENT_CC: The e-mail is copied to the recipient. MSMQMAIL_RECIPIENT_BCC: The e-mail is blind copied to the recipient.
DeliveryTime	Optional. *DeliveryTime* is only used when adding a recipient to the **DeliveredRecipients** property of the **MSMQMailDeliveryReportData** object. This property specifies when the original e-mail was delivered to the recipient.
NonDeliveryReason	Optional. *NonDeliveryReason* is only used when adding a recipient to the **NonDeliveredRecipients** property of the **MSMQMailNonDeliveryReportData** object. This property specifies the reason why the

	original e-mail was not delivered to the recipient.
Key	Optional. Key (Variant type) used when removing the recipient from the recipient list (see *IndexKey* parameter of **Remove**).

Syntax for Adding Field

object1.object2.object3.**Add** *Name*, *Value*[, *Key*]

Syntax Element	Description
object1	E-mail message (**MSMQMailEMail**) object that defines the e-mail message.
object2	Form field data (MSMQMailFormData) object that defines the form.
object3	Form field list (**MSMQMailFormFieldList**) object that represents the list of fields in the form.
Name	String representation of the name of the field. This name should correspond to the name of the field's *Reference Name* as specified in the Exchange Form Designer. The *Reference Name* can be found on the *General* page of the *Field Properties* dialog of the Exchange Form Designer. If the correct *Reference Name* is not used, Microsoft Exchange may not display the field correctly.
Value	String, Integer, Boolean, Double, or Currency value of field.
Key	Optional. Key used when removing the field from form field list (see *IndexKey* parameter of **Remove**).

Remarks

Recipient List

The format of *Address* varies depending on the MSMQ Mail Service used or if the destination is another MSMQ apllication.

Form Field List

Fields can be one of the following types: String, Integer, Boolean, Double, or Currency value of field.

Recipient List Example

This example defines an e-mail message with two recipients, then displays the name and address of each recipient. The addresses of the recipients indicate that the mail will be sent to the Exchange Connector (first recipient) and the MAPI transport Provider (second recipient).

Note For examples of setting the *NonDeliveryReason* and *DeliveryTime* parameters, see the examples in **NonDeliveryReason** and **DeliveryTime** properties respectively.

To try this example using Microsoft® Visual Basic® (version 5.0), paste the code into the Code window of a form, and then run the example and click the form.

```
Private Sub Form_Click()

Dim email As New MSMQMailEMail

'Set e-mail type to text message
 email.ContentType= MSMQMAIL_EMAIL_TEXTMESSAGE
```

```
'**********************************
'* Add Exchange Connector recipient
'* as primary recipient.
'**********************************
email.Recipients.Add "Exchange_User", "ExchangeUser@ServerInputQueueLabel",
MSMQMAIL_RECIPIENT_TO

'**********************
'* Add MAPI recipient
'* as CC recipient.
'**********************
email.Recipients.Add "MAPI_User", "MAPIUserInputQueueLabel", MSMQMAIL_RECIPIENT_CC


'Set who sent the e-mail.
email.Sender.Name = "Our name"
email.Sender.Address = "Our queue label"

'Set the subject of the e-mail.
email.Subject = "Test mail."

'Set the Body of the e-mail.
email.TextMessageData.Text = "This is the Body of the message."


'**********************
'* Display Recipients.
'**********************

Dim recipient As MSMQMailRecipient

For Each recipient In email.Recipients
    MsgBox "Mail was sent to " + recipient.Name + " at " + recipient.Address
Next recipient


End Sub
```

Field List Example

This example defines an e-mail message with three fields (string Boolean, and Date), then displays the value given to each field.

To try this example using Microsoft Visual Basic (version 5.0), paste the code into the Code window of a form, and then run the example and click the form.

```
Dim email As New MSMQMailEMail

Private Sub Form_Click()

    '**********************
    '*   Define e-mail
    '**********************

    'Set e-mail type to form message.
```

```
    email.ContentType = MSMQMAIL_EMAIL_FORM

    'Add primary recipient.
    email.Recipients.Add "Exchange_User", "ExchangeUser@ServerInputQueueLabel",
MSMQMAIL_RECIPIENT_TO

    'Set who sent the e-mail.
    email.Sender.Name = "Our name"
    email.Sender.Address = "Our queue label"

    'Set the subject of the mail.
    email.Subject = "Test form."

    '**********************
    '* Define the form.
    '**********************

    'Set the form name.
    email.formdata.Name = "Test Form"

    'Set form field list.
      email.formdata.FormFields.Add "StringField", "Test Field"
      email.formdata.FormFields.Add "BooleanField", True
      email.formdata.FormFields.Add "DateField", “DateString”


    '**********************
    '* Display fields.
    '**********************

    Dim formfield As MSMQMailFormField

    For Each formfield In email.formdata.FormFields
      MsgBox "Form: " + formfield.Name + " = " +      CStr(formfield.Value)

    Next formfield

End Sub
```

See Also

Address, **ContentType**, **FormData**, **FormFields**, **MSMQMailEMail**, **MSMQMailRecipient**, **Name**, **Recipients**, **Sender**, **Subject**, **Text**, **TextMessageData**

ComposeBody

MSMQMailEMail

The **ComposeBody** method creates the body of an MSMQ mail message based on an e-mail message.

Syntax

varBody=object.**ComposeBody**

Syntax Element	Description
object	E-mail message (**MSMQMailEMail**) object that defines the e-mail message.
varBody	Variant type (array of bytes) Body of an MSMQ mail message.

Return Values

Byte-array message body.

Remarks

The returned byte-array can be used as the body of an MSMQ mail message.

The following table defines which properties of the **MSMQMailEMail** object are required to compose a message body for each type of e-mail. Required properties are marked with an X (default values can be used). When composing an e-mail object, all required properties must have valid values, otherwise an error condition is raised by **ComposeBody**.

MSMQMailEMail	Text Message	Form	TNEF	Delivery Report	Non-Delivery Report
ContentType	X	X	X	X	X
DeliveryReportData				X	
DestinationQueueLabels					
FormData		X			
NonDeliveryReportData					X
Recipients	X	X	X	X	X
RequestDeliveryReport	X (False)	X (False)			
RequestNonDeliveryReport	X (False)	X (False)			
Sender	X	X			
Subject	X (Empty string)	X (Empty string)			
SubmissionTime	X (Creation time of e-mail object)	X (Creation time of e-mail object)		X (Creation time of e-mail object)	X (Creation time of e-mail object)
TestMessageData	X				
TnefData			X		

Example

This example defines an e-mail message, composes the body of an MSMQ message (formatted in MSMQ mail format), then prints out the body of the message.

To try this example using Microsoft Visual Basic (version 5.0), paste the code into the Code window of a form, and then run the example and click the form.

```
Dim email As New MSMQMailEMail
Dim msg As New MSMQMessage

Private Sub Form_Click()

    '**********************
    '*   Define the e-mail
    '**********************

    'Set e-mail type as text message
    email.ContentType = MSMQMAIL_EMAIL_TEXTMESSAGE

    'Set who receives the e-mail.
    email.Recipients.Add "RecipientName", "RecipientQueueLabel",
MSMQMAIL_RECIPIENT_TO

    'Set who sent the e-mail.
    email.Sender.Name = "Our name"
    email.Sender.Address = "Our queue label"

    'Set the subject of the e-mail.
    email.Subject = "Test mail."

    'Set the Body of the e-mail.
    email.TextMessageData.Text = "This is the Body of the e-mail message."

    'Compose the MSMQ mail message Body.
    msg.Body = email.ComposeBody


    '*********************************
    '*   Display the MSMQ message Body.
    '*********************************

    'Display the mail message Body.
    Dim lTmp As Long
    For lTmp = LBound(msg.Body) To UBound(msg.Body)
        Debug.Print Chr$(msg.Body(lTmp));
    Next lTmp


End Sub
```

See Also

Add, **Address**, **Body**, **ContentType**, **MSMQMailEMail**, **MSMQMessage**, **Name**, **Recipients**, **Sender**, **Subject**, **Text**, **TextMessageData**

ParseBody

MSMQMailEMail

The **ParseBody** method sets the properties of an e-mail message object based on the body of an MSMQ mail message.

Syntax

object.**ParseBody** *varBody*

Syntax Element	Description
object	E-mail message (**MSMQMailEMail**) object that defines the e-mail message.
varBody	Variant type (array of bytes) Body of an MSMQ mail message.

Remarks

The body of the MSMQ mail message is formatted in MSMQ Mail format.

Example

This example defines an e-mail message object, composes the body of a mail message, then parses the message body into a second e-mail object and displays its subject.

To try this example using Microsoft Visual Basic (version 5.0), paste the code into the Code window of a form, and then run the example and click the form.

```
Dim email1 As New MSMQMailEMail
Dim email2 As New MSMQMailEMail
Dim msg As New MSMQMessage

Private Sub Form_Click()

    '**********************
    '*   Define e-mail
    '**********************

    'Set e-mail type as form.
    email1.ContentType = MSMQMAIL_EMAIL_FORM

    'Set who receives the e-mail.
    email1.Recipients.Add "RecipientName", "RecipientQueueLabel",
MSMQMAIL_RECIPIENT_TO

    'Set who sent the e-mail.
    email1.Sender.Name = "Our name"
    email1.Sender.Address = "Our queue label"

    'Set the subject of the e-mail.
    email1.Subject = "Test mail."

    '**********************
    ' Define the form.
    '**********************

    'Set the form name.
```

```
    email1.formdata.Name = "Test Form"

    'Set single field of form
    email1.formdata.FormFields.Add "Field1", "Field1 text."


    '******************************
    'Compose the mail message Body.
    '******************************
    msg.Body = email1.ComposeBody


    '**********************
    'Parse message Body as
    'new e-mail object.
    '**********************

    email2.ParseBody msg.Body

    '****************************************
    'Display the subject of new e-mail object.
    '****************************************

    MsgBox "The subject of the new mail is: " + email2.Subject


End Sub
```

See Also

Add, **Address**, **Body**, **ComposeBody**, **ContentType**, **FormData**, **FormFields**, **MSMQMailEMail**, **MSMQMessage**, **Name**, **Recipients**, **Sender**, **Subject**

Remove

MSMQMailFormFieldList **MSMQMailRecipientList**

The **Remove** method removes a specific recipient from the recipient list (**MSMQMailRecipientList**)or a field from the form field list (**MSMQMailFormFieldList**).

Syntax for Removing a Recipient

object1.object2.**Remove** *IndexKey*

Syntax Element	Description
object1	E-mail message (**MSMQMailEMail**) object that defines the e-mail message.
object2	Recipient (**MSMQMailRecipientList**) list object.
IndexKey	Index of the **Item** array, or the key specified when the recipient was added to the list (see *Key* parameter of **Add**).

Syntax for Removing a Field

object1.object2.object3.**Remove** *IndexKey*

Syntax Element	Description
object1	E-mail message (**MSMQMailEMail**) object that defines the e-mail message.
object2	Form field data (MSMQMailFormData) object that defines the form.
object3	Form field (**MSMQMailFormFieldList**) list object.
IndexKey	Index of the **Item** array, or the key specifies when the recipient or field was added to the list (see *Key* parameter of **Add**).

Example of Removing a Recipient

This example creates an e-mail object with a primary, copy, and blind copy recipient, setting the index key on the blind-copy recipient. It then displays the names of the recipients, removes the blind-copy recipient using its index key, then displays the names of the recipients remaining in the form.

To try this example using Microsoft Visual Basic (version 5.0), paste the code into the Code window of a form, and then run the example and click the form.

```
Dim email As New MSMQMailEMail

Private Sub Form_Click()

    'Set e-mail type to text message
    email.ContentType = MSMQMAIL_EMAIL_TEXTMESSAGE

    '**********************************
    '* Add primary recipient.
    '**********************************
    email.Recipients.Add "Exchange_User", "ExchangeUser@ServerInputQueueLabel",
MSMQMAIL_RECIPIENT_TO
```

```
    '*********************
    '* Add CC recipient.
    '*********************
    email.Recipients.Add "MAPI_User1", "MAPIUserInputQueueLabel",
MSMQMAIL_RECIPIENT_CC

    '*********************
    '* Add BC recipient.
    '*********************
    email.Recipients.Add "MAPI_User2", "MAPIUserInputQueueLabel",
MSMQMAIL_RECIPIENT_BCC, "BC"


    'Set who sent the e-mail.
    email.Sender.Name = "Our name"
    email.Sender.Address = "Our queue label"

    'Set the subject of the e-mail.
    email.Subject = "Test mail."

    'Set the Body of the e-mail.
    email.TextMessageData.Text = "This is the Body of the message."


    '*******************************
    '* Remove blind-copy Recipients.
    '*******************************

    Dim recipient As MSMQMailRecipient

    Debug.Print "**Old Recipient List**"
    For Each recipient In email.Recipients
        Debug.Print recipient.Name
    Next recipient

    email.Recipients.Remove "BC"

    Debug.Print "**New Recipient List**"
    For Each recipient In email.Recipients
            Debug.Print recipient.Name
    Next recipient

End Sub
```

Example of Removing a Field

This example creates an e-mail form with three fields, setting the index key of the date field. It then displays the names and values of all fields, removes the date field using its index key, then displays the names of the fields remaining in the form.

To try this example using Microsoft Visual Basic (version 5.0), paste the code into the Code window of a form, and then run the example and click the form.

```
Dim email As New MSMQMailEMail
```

```
Private Sub Form_Click()


    '**********************
    '*   Define e-mail
    '**********************

    'Set e-mail type to form message.
    email.ContentType = MSMQMAIL_EMAIL_FORM

    'Add primary recipient.
    email.Recipients.Add "Exchange_User", "ExchangeUser@ServerInputQueueLabel",
MSMQMAIL_RECIPIENT_TO

    'Set who sent the e-mail.
    email.Sender.Name = "Our name"
    email.Sender.Address = "Our queue label"

    'Set subject of mail.
    email.Subject = "Test form."

   'Set form name.
   email.formdata.Name = "Test Form"

 'Set form field list.
    email.formdata.FormFields.Add "StringField", "Test Field"
    email.formdata.FormFields.Add "BooleanField", True
    email.formdata.FormFields.Add "DateField", “DateString”, "SentDate"

    '***********************
    '* Remove Date field.
    '***********************

    Dim formfield As MSMQMailFormField

    Debug.Print "**Old Field List**"
    For Each formfield In email.formdata.FormFields
        Debug.Print formfield.Name + ": " + CStr(formfield.Value)
    Next formfield

    email.formdata.FormFields.Remove "SentDate"

    Debug.Print "**New Field List**"
    For Each formfield In email.formdata.FormFields
        Debug.Print formfield.Name + ": " + CStr(formfield.Value)
    Next formfield

End Sub
```

See Also

Add, **Address**, **ContentType**, **FormData**, **FormFields**, **MSMQMailEMail**, **MSMQMailFormField**, **MSMQMailRecipient**, **Name**, **Recipients**, **Sender**, **Subject**, **Text**, **TextMessageData**, **Value**

MSMQ Mail ActiveX Properties

The following topics describe the properties associated with the MSMQ Mail ActiveX components.

Note The MSMQ Mail SDK must be installed to use any of the MSMQ Mail ActiveX components.

MSMQMailEMail

ContentType

DeliveryReportData

DestinationQueueLabels

FormData

NonDeliveryReportData

Recipients

RequestDeliveryReport

RequestNonDeliveryReport

Sender

Subject

SubmissionTime

TextMessageData

TnefData

MSMQMailFormData

FormFields

Name

MSMQMailFormField

Name

Value

MSMQMailFormFieldList

Count

Item

MSMQMailRecipient

Address

DeliveryTime

Name

NonDeliveryReason

RecipientType

MSMQMailRecipientList

Count

Item

MSMQMailTextMessageData

Text

MSMQMailTnefData

Data

MSMQMailDeliveryReportData

DeliveredRecipients

OriginalSubject

OriginalSubmissionTime

MSMQMailNonDeliveryReportData

NonDeliveredRecipients

OriginalEMail

Address

MSMQMailRecipient

The **Address** property specifies the e-mail address of the recipient.

Quick Info

Type:	**String**
Run time:	Read-write

Syntax

object1,object2.**Address**

Syntax Element	Description
object1	An e-mail (**MSMQMailEMail**) object that defines an e-mail message.
object2	Recipient (**MSMQMailRecipient**) object the represents the e-mail recipient.

Settings

String representation of the e-mail recipient's address. Use one of the following formats:

"UserAlias@ServerInputQueueLabel"	Exchange Connector
"MAPIUserInputQueueLabel"	MAPI Transport Provider
"ApplicationInputQueueLabel"	MSMQ Application

Remarks

Typically, **Address** is not referenced explicitly. In most cases, it is set when adding a recipient to the recipient list.

When the e-mail is being sent to an MSMQ application, the recipient's address is the label (**Label**) or the application input queue.

When the e-mail is being sent to an e-mail user, the recipient's address varies depending on which MSMQ Mail service is being used.

- For applications using the MSMQ Exchange Connector, the address includes the e-mail alias of the MS Exchange user plus the label of the MSMQ Exchange server input queue.
- For applications using the MSMQ MAPI Transport Provider, the address includes the e-mail alias, plus the label of the MAPI Transport Provider's User Input Queue.

Example

This example defines an e-mail form, adding three recipients to the e-mail's recipient list. The e-mail object is composed into a mail message, then each recipient is displayed.

To try this example using Microsoft Visual Basic (version 5.0), paste the code into the Code window of a form, enter valid user address for each recipient, then run the example and click the form.

```
Dim email As New MSMQMailEMail
Dim msg As New MSMQMessage

Private Sub Form_Click()

    '**********************
    '*   Define e-mail
    '**********************
```

```
    'Set e-mail type to form message.
    email.ContentType = MSMQMAIL_EMAIL_FORM

    'Add Recipients.
    email.Recipients.Add "Exchange_User", "UserAlias@ServerInputQueueLabel",
MSMQMAIL_RECIPIENT_TO
    email.Recipients.Add "MAPI_User", "MAPIUserInputQueueLabel",
MSMQMAIL_RECIPIENT_CC
    email.Recipients.Add "MSMQApplication", "ApplicationInputQueueLabel",
MSMQMAIL_RECIPIENT_BCC

    'Set who sent the e-mail.
    email.Sender.Name = "Our name"
    email.Sender.Address = "Our queue label"

    'Set subject of mail.
    email.Subject = "Test mail"

    'Set name of form
    email.FormData.Name = "Test form"

    'Set form field list.
    email.FormData.FormFields.Add "StringField", "Test Field"

    'Compose message Body
    msg.Body = email.ComposeBody


    '**********************
    '* Display Recipients.
    '**********************

    Dim recipient As MSMQMailRecipient

    For Each recipient In email.Recipients
        MsgBox "Recipient: " + recipient.Name + " at " + recipient.Address
    Next recipient

End Sub
```

See Also

Add, **ComposeBody**, **ContentType**, **FormData**, **FormFields**, **MSMQMailEMail**, **Name**, **Recipients**, **Sender**, **Subject**

ContentType

MSMQMailEMail

The **ContentType** property specifies the type of e-mail object (**MSMQMailEMail**).

Quick Info

Type:	**Long**
Run time:	Read-write

Syntax for E-mail Type

object.**ContentType**

Syntax Element	Description
object	An e-mail (**MSMQMailEMail**) object that defines an e-mail message.

Settings

When specifying the type of e-mail message, **ContentType** can have any one of the following values:

MSMQMAIL_EMAIL_TEXTMESSAGE
The e-mail message consist of a text message (see **TextMessageData**).

MSMQMAIL_EMAIL_FORM
The e-mail message is a form (see **FormData**).

MSMQMAIL_EMAIL_TNEF
The e-mail message is a TNEF message (see **TnefData**).

MSMQMAIL_EMAIL_DELIVERY_REPORT
The e-mail message is a delivery report (see **DeliveryReportData**).

MSMQMAIL_EMAIL_NON_DELIVERY_REPORT
The e-mail message is a non-delivery report (see **NonDeliveryReportData**).

Remarks

This property must be set when defining an e-mail object.

Example

This example prints out the form specific information of a e-mail form or the text-message information of an email text message.

To try this example using Microsoft® Visual Basic® (version 5.0), paste the code into the Code window of a form, and then run the example and click the form.

```
Dim email As New MSMQMailEMail
Dim msg As New MSMQMessage

Private Sub Form_Click()

    '**********************
    '*   Define form e-mail
    '**********************
    'Set e-mail type to form message.
    email.ContentType = MSMQMAIL_EMAIL_FORM

    'Add primary recipient.
```

```
    email.Recipients.Add "Exchange_User", "ExchangeUser@ServerInputQueueLabel",
MSMQMAIL_RECIPIENT_TO

    'Set who sent the e-mail.
    email.Sender.Name = "Our name"
    email.Sender.Address = "Our queue label"

    'Set subject of mail.
    email.Subject = "Test form."

    'Set form name.
    email.FormData.Name = "Test Form"

    'Set form field list.
    email.FormData.FormFields.Add "Name", "Test Field"

  '************************
  '* Compose message bodies.
  '************************
  msg.Body = email.ComposeBody

  MsgBox "The e-mail form object was created."

End Sub
```

See Also

Add, **Address**, **ComposeBody**, **FormData**, **MSMQMailEMail**, **MSMQMessage**, **Name**, **Recipients**, **Subject**

Count

MSMQMailFormFieldList **MSMQMailRecipientList**

Read-only. The **Count** property indicates the number of recipients in the recipient list or the number of fields in the field list.

Quick Info

Type:	**Long**
Run time:	Read-only

Syntax for Recipient List

object1.object2.**Count**

Syntax Element	Description
object1	E-mail message (**MSMQMailEMail**) object that defines the e-mail message.
object2	The recipient list (**MSMQMailRecipientList**) object that defines the list of recipients.

Syntax for Field List

object1.object2.object3.**Count**

Syntax Element	Description
object1	E-mail message (**MSMQMailEMail**) object that defines the e-mail message.
object2	Form field data (MSMQMailFormData) object that defines the form.
object3	The field list (**MSMQMailFormFieldList**) object that defines the fields on a form.

Return Values

Number of items in list.

Remarks

Count is updated each time the **Add** or **Remove** method of either list object (**MSMQMailRecipientList** and **MSMQMailFormFieldList**) is called.

Example of Recipient List

This example defines an e-mail object, adds three recipients to the recipient list, then displays the number of recipients followed by the name of each recipient.

To try this example using Microsoft® Visual Basic® (version 5.0), paste the code into the Code window of a form, and then run the example and click the form.

```
Dim email As New MSMQMailEMail

Private Sub Form_Click()

    'Set e-mail type to text message
    email.ContentType = MSMQMAIL_EMAIL_TEXTMESSAGE

    '**********************************
    '* Add Recipients.
```

```
    '*********************************
    email.Recipients.Add "Exchange_User", "ExchangeUser@ServerInputQueueLabel",
MSMQMAIL_RECIPIENT_TO
    email.Recipients.Add "MAPI_User1", "MAPIUserInputQueueLabel",
MSMQMAIL_RECIPIENT_CC
    email.Recipients.Add "MAPI_User2", "MAPIUserInputQueueLabel",
MSMQMAIL_RECIPIENT_BCC, "BC"


    'Set who sent the e-mail.
    email.Sender.Name = "Our name"
    email.Sender.Address = "Our queue label"

    'Set the subject of the e-mail.
    email.Subject = "Test mail."

    'Set the Body of the e-mail.
    email.TextMessageData.Text = "This is the Body of the message."


    '*******************************
    '* Count recipient.
    '*******************************

    Dim recipient As MSMQMailRecipient

    Debug.Print "**Number of Recipient in List**"
    Debug.Print CStr(email.Recipients.Count)

    Debug.Print "**Recipients**"
    For Each recipient In email.Recipients
        Debug.Print recipient.Name
    Next recipient

End Sub
```

Example of Form Field List

This example defines an e-mail object as a form, adds three fields to the form, then displays the number of fields followed by each fields name.

To try this example using Microsoft Visual Basic (version 5.0), paste the code into the Code window of a form, and then run the example and click the form.

```
Dim email As New MSMQMailEMail

Private Sub Form_Click()

    '**********************
    '*   Define e-mail
    '**********************

    'Set e-mail type to form message.
    email.ContentType = MSMQMAIL_EMAIL_FORM

    'Add primary recipient.
```

```
    email.Recipients.Add "Exchange_User", "ExchangeUser@ServerInputQueueLabel",
MSMQMAIL_RECIPIENT_TO

    'Set who sent the e-mail.
    email.Sender.Name = "Our name"
    email.Sender.Address = "Our queue label"

    'Set subject of the mail.
    email.Subject = "Test form."

   'Set form name.
   email.formdata.Name = "Test Form"

   'Set form field list.
    email.formdata.FormFields.Add "StringField", "Test Field"
    email.formdata.FormFields.Add "BooleanField", True
    email.formdata.FormFields.Add "DateField", “DateString”, "SentDate"


    '*******************************
    '* Count fields.
    '*******************************

    Dim formfield As MSMQMailFormField

    Debug.Print "**Number of Fields in Form**"
    Debug.Print CStr(email.formdata.FormFields.Count)

    Debug.Print "**Fields**"
    For Each formfield In email.formdata.FormFields
        Debug.Print formfield.Name
    Next formfield

End Sub
```

See Also

Add, **Address**, **ContentType**, **FormData**, **FormFields**, **MSMQMailEMail**, **MSMQMailFormField**, **MSMQMailRecipient**, **Name**, **Recipients**, **Sender**, **Subject**, **Text**, **TextMessageData**

Data

MSMQMailTnefData

The **Data** property specifies the data of a TNEF e-mail.

Quick Info

Type:	Variant (variant type is an array of bytes)
Run time:	Read-write

Syntax

object1.object2.**Data**

Syntax Element	Description
object1	E-mail (**MSMQMailEMail**) object that defines the e-mail message.
object2	TNEF data (**MSMQMailTnefData**) object that represents the text message.

Settings

A variant which contains the TNEF data of the e-mail. The data in the variant should be an array of bytes representing the TNEF data.

TNEF is a MAPI internal format that encapsulates the MAPI properties, and is used by the MSMQ Mail services (the MSMQ Exchnage Connector and the MSMQ MAPI Transport) to send mail to recipients who are defined as rich-text recipients. These recipients have selected the check box labeled “Send to this recipient in Microsoft Exchange rich text format” in their Exchange/MAPI address.

Example

This example defines an e-mail object as a TNEF message, setting the TNEF message body to a predefined variant value that should be filled by MAPI or taken from another TNEF message. The email object is then used to compose the body of a mail message, and a message box displays the size of the TNEF data (the data itself is binary and is not supposed to be readable).

To try this example using Microsoft Visual Basic (version 5.0), paste the code into the Code window of a form, then run the example and click the form.

```
Dim email As New MSMQMailEMail
Dim msg As New MSMQMessage

Private Sub Form_Click()

    '************************
    '*   Define e-mail object
    '************************

    'Set e-mail type to TNEF message
    email.ContentType = MSMQMAIL_EMAIL_TNEF

    'Add Recipients
    email.Recipients.Add "Connector Recipient Name",
"ExchangeUser@ServerInputQueueLabel", MSMQMAIL_RECIPIENT_TO
    email.Recipients.Add "MAPI Recipient Name", "MAPIUserInputQueueLabel",
MSMQMAIL_RECIPIENT_CC

'Set the TNEF information of the e-mail.
```

```
    email.TnefData.Data = tnefdata

    '**********************
    '* Compose message Body
    '**********************
    msg.Body = email.ComposeBody

    '****************************
    '* Display size of TNEF data.
    '****************************
    MsgBox "The size of the TNEF data is:" + (UBound(email.TnefData.Data) -
LBound(email.TnefData.Data) + 1)

End Sub
```

See Also

Add, **Address**, **Body**, **ComposeBody**, **ContentType**, **MSMQMailEMail**, **MSMQMessage**, **Name**, **Recipients**, **Sender**, **Subject**, **TextMessageData**

DeliveredRecipients

MSMQMailDeliveryReportData

The **DeliveredRecipients** property specifies the recipients that received a previously sent e-mail.

Quick Info

Type:	**MSMQMailRecipientList**
Run time:	Read-write

Syntax

object1.object2.**DeliveredRecipients**

Syntax Element	Description
Object1	E-mail (**MSMQMailEMail**) object that defines the delivery report.
Object2	Delivery report data (**MSMQMailDeliveryReportData**) object that represents the delivery report information.

Settings

MSMQMailRecipientList object.

Remarks

Each recipient in the delivered recipient list is represented by an **MSMQMailRecipient** object. Each recipient object includes the name and address of the recipient, the input queue of the recipient, how the message was sent to recipient, plus when the message was delivered.

The **DeliveryTime** property of the **MSMQMailRecipient** object specifies when the original e-mail was delivered to the recipient.

Example

This example defines a delivery report, adding two delivered recipients to the delivered recipient list of the report. The e-mail object is composed into a mail message, then each delivered recipient is displayed.

To try this example using Microsoft® Visual Basic® (version 5.0), paste the code into the Code window of a form, enter valid user address for each recipient, then run the example and click the form.

```
Dim email As New MSMQMailEMail
Dim msg As New MSMQMessage

Private Sub Form_Click()

    '**********************
    '*   Define e-mail
    '**********************

    'Set e-mail type to delivery report.
    email.ContentType = MSMQMAIL_EMAIL_DELIVERY_REPORT

    'Add the Recipient of the report (usually the original e-mail sender).
    email.Recipients.Add "Exchange_User", "UserAlias@ServerInputQueueLabel",
MSMQMAIL_RECIPIENT_TO

    'Set subject of original mail.
```

```
    email.DeliveryReportData.OriginalSubject = "Original subject "

    'Set submission time of original mail.
    email.DeliveryReportData.OriginalSubmissionTIme = CDate("5/20/94 10:16:07 PM")

     'Add two delivered recipients.
    email.DeliveryReportData.DeliveredRecipients.Add "Exchange_User2",
"UserAlias2@ServerInputQueueLabel", MSMQMAIL_RECIPIENT_TO, DeliveryTime:=
CDate("5/20/94 10:17:00 PM")
email.DeliveryReportData.DeliveredRecipients.Add "Exchange_User3",
"UserAlias3@ServerInputQueueLabel", MSMQMAIL_RECIPIENT_CC, DeliveryTime:=
CDate("5/20/94 11:01:00 PM")

    'Compose message Body
    msg.Body = email.ComposeBody


    '**********************
    '* Display Delivered Recipients.
    '**********************

    Dim recipient As MSMQMailRecipient

    For Each recipient In email.DeliveryReportData.DeliveredRecipients
        MsgBox "Delivered To Recipient: " + recipient.Name + " at " + recipient.Address + “
on “ + recipient.DeliveryTime
    Next recipient

End Sub
```

See Also

Add, **Address**, **ComposeBody**, **ContentType**, **FormData**, **FormFields**, **MSMQMailEMail**, **Name**, **Sender**, **Subject**, **DeliveryReportData**

DeliveryReportData

MSMQMailEMail

The **DeliveryReportData** defines a delivery report. This report contains a list of the recipients who received the original e-mail, when the original e-mail was submitted, and the subject of the original e-mail.

This property is only meaningful if **ContentType** is set to MSMQMAIL_EMAIL_DELIVERY_REPORT.

Quick Info

Type:	**MSMQMailDeliveryReportData**
Run time:	Read-write

Syntax

object.**DeliveryReportData**

Syntax Element	Description
object	E-mail message (**MSMQMailEMail**) object that defines the e-mail message.

Settings

MSMQMailDeliveryReportData object.

Remarks

When defining delivery report, set **ContentType** to MSMQMAIL_EMAIL_DELIVERY_REPORT whenever **DeliveryReportData** is set.

After parsing an MSMQ Mail message, verify that, **ContentType** is set to MSMQMAIL_EMAIL_DELIVERY_REPORT , before looking at **DeliveryReportData**. This property is empty if **ContentType** indicates another e-mail type.

DeliveryTime

MSMQMailRecipient

The **DeliveryTime** property specifies the time when the original e-mail was delivered to the recipient (**MSMQMailRecipient**).

Quick Info

Type:	**Date**
Run time:	Read-write

Syntax

object.**DeliveryTime**

Syntax Element	Description
Object	A recipient (**MSMQMailRecipient**) object that defines an e-mail recipient.

Settings

The time when the original e-mail was delivered to the recipient.

Remarks

DeliveryTime has a valid value only when the recipient is a member of the **DeliveredRecipients** property of a delivery report e-mail.

Example

This example defines a delivery report, adding two delivered recipients to the delivered recipient list of the report. The e-mail object is composed into a mail message, then each delivered recipient is displayed.

To try this example using Microsoft® Visual Basic® (version 5.0), paste the code into the Code window of a form, enter valid user address for each recipient, then run the example and click the form.

```
Dim email As New MSMQMailEMail
Dim msg As New MSMQMessage

Private Sub Form_Click()

    '**********************
    '*   Define e-mail
    '**********************

    'Set e-mail type to delivery report.
    email.ContentType = MSMQMAIL_EMAIL_DELIVERY_REPORT

    'Add the Recipient of the report (usually the original e-mail sender).
    email.Recipients.Add "Exchange_User", "UserAlias@ServerInputQueueLabel",
MSMQMAIL_RECIPIENT_TO

    'Set subject of original mail.
    email.DeliveryReportData.OriginalSubject = "Original subject "

    'Set submission time of original mail.
    email.DeliveryReportData.OriginalSubmissionTime = CDate("5/20/94 10:16:07 PM")
```

```
    'Add two delivered recipients.
    email.DeliveryReportData.DeliveredRecipients.Add "Exchange_User2",
"UserAlias2@ServerInputQueueLabel", MSMQMAIL_RECIPIENT_TO, DeliveryTime:=
CDate("5/20/94 10:17:00 PM")
email.DeliveryReportData.DeliveredRecipients.Add "Exchange_User3",
"UserAlias3@ServerInputQueueLabel", MSMQMAIL_RECIPIENT_CC, DeliveryTime:=
CDate("5/20/94 11:01:00 PM")

    'Compose message Body
    msg.Body = email.ComposeBody


    '**********************
    '* Display Delivered Recipients.
    '**********************

    Dim recipient As MSMQMailRecipient

    For Each recipient In email.DeliveryReportData.DeliveredRecipients
        MsgBox "Delivered To Recipient: " + recipient.Name + " at " + recipient.Address + “
on “ + recipient.DeliveryTime
    Next recipient

End Sub
```

See Also

Add, **Body**, **ComposeBody**, **ContentType**, **FormData**, **FormFields**, **MSMQMailEMail**, **MSMQMAILFormField**, **Name**, **ParseBody**, **Recipients**, **Subject**, **Text**, **TextMessageData**, **Value**

DestinationQueueLabels

MSMQMailEMail

The **DestinationQueueLabels** property provides the label of each destination queue needed to send the MSMQ mail message. These labels are taken from the recipient addresses defined by the e-mail object.

Quick Info

Type:	Collection of Strings
Run time:	Read-only

Syntax

object.**DestinationQueueLabels**

Syntax Element	Description
object	An e-mail (**MSMQMailEMail**) object that defines an e-mail message.

Returned Value

String representation of each queue label needed to send message.

Remarks

Typically, a query is used to locate the destination queues of an e-mail. When running a query, make sure the queue type is added to the search criteria as well as the destination queue label. All application input queues, MAPI user input queues, and Exchange service input queues use the following MSMQ-defined queue type: MSMQMAIL_SERVICE_MAIL.

When sending mail to several Exchange user via the MSMQ Exchange Connector, only one label is stored in the destination queue collection. Only one label is needed because the Exchange connector has only one input queue.

Example

This example sends mail to each destination queue.

To try this example using Microsoft Visual Basic (version 5.0), paste the code into the Code window of a form, enter valid user address for each recipient, then run the example and click the form.

```
Dim email As New MSMQMailEMail
Dim msg As New MSMQMessage


Private Sub Form_Click()

    '**********************
    '*   Define e-mail
    '**********************

    'Set e-mail type to text message.
    email.ContentType = MSMQMAIL_EMAIL_TEXTMESSAGE

    'Add recipient.
    email.Recipients.Add "User1", "User1Address", MSMQMAIL_RECIPIENT_TO
    email.Recipients.Add "User2", "User2Address", MSMQMAIL_RECIPIENT_CC

    'Set who sent the e-mail.
```

```
    email.Sender.Name = "Our name"
    email.Sender.Address = "Our queue label"

    'Set subject of mail.
    email.Subject = "Test form."

    'Set message text
    email.TextMessageData.Text = "This is a test message."


    '******************
    '* Compose message
    '******************
    msg.Body = email.ComposeBody

    '******************
    '* Send Message
    '******************

    Dim qLabel As Variant

    For Each qLabel In email.DestinationQueueLabels
        Dim query As MSMQQuery
        Dim qinfos As MSMQQueueInfos
        Dim qinfo As MSMQQueueInfo

        Set query = New MSMQQuery
        Set qinfos = query.LookupQueue(Label:=qLabel,
ServiceTypeGuid:=MSMQMAIL_SERVICE_MAIL)

        qinfos.Reset
        Set qinfo = qinfos.Next

        If Not (qinfo Is Nothing) Then
            Dim qdestination As MSMQQueue
            Set qdestination = qinfo.Open(MQ_SEND_ACCESS, MQ_DENY_NONE)
            msg.Send qdestination
        End If

    Next qLabel

End Sub
```

See Also

Add, **Address**, **Body**, **ComposeBody**, **ContentType**, **LookupQueue**, **MSMQMailEMail**, **MSMQMessage**, **MSMQQuery**, **MSMQQueue**, **MSMQQueueInfo**, **MSMQQueueInfos**, **Name**, **Open**, **Recipients**, **Send**, **Sender**, **Subject**, **Text**, **TextMessageData**

FormData

MSMQMailEMail

The **FormData** property defines the form. It specifies the name of the form, and lists all the fields in the form.

This property is only meaningful if **ContentType** is set to MSMQMAIL_EMAIL_FORM.

Quick Info

Type:	**MSMQMailFormData**
Run time:	Read-write

Syntax

object.**FormData**

Syntax Element	Description
object	E-mail message (**MSMQMailEMail**) object that defines the e-mail message.

Settings

MSMQMailFormData object.

Remarks

When defining an e-mail object that represent a form, set **ContentType** to MSMQMAIL_EMAIL_FORM whenever **FormData** is set.

After parsing an MSMQ Mail message, verify that, **ContentType** is set to MSMQMAIL_EMAIL_FORM, before looking at **FormData**. This property is empty if **ContentType** indicates another e-mail type.

Example

This example defines an e-mail form with three fields, composes the body of an MSMQ mail message, then displays a name of the form plus the number of fields in the form.

To try this example using Microsoft Visual Basic (version 5.0), paste the code into the Code window of a form, and then run the example and click the form.

```
Option Explicit
Dim email As New MSMQMailEMail
Dim msg As New MSMQMessage

Private Sub Form_Click()

    '**********************
    '*   Define e-mail
    '**********************

    'Set e-mail type as form.
    email.ContentType = MSMQMAIL_EMAIL_FORM

    'Set who receives the e-mail.
    email.Recipients.Add "RecipientName", "RecipientQueueLabel",
MSMQMAIL_RECIPIENT_TO

    'Set who sent the e-mail.
    email.Sender.Name = "Our name"
```

```
    email.Sender.Address = "Our queue label"

    'Set the subject of the e-mail.
    email.Subject = "Test mail."

    '**********************
    '* Define the form.
    '**********************

    'Set the form name.
    email.FormData.Name = "Test Form"

    'Set fields of form
    email.FormData.FormFields.Add "Field1", "Field1 text."
    email.FormData.FormFields.Add "Field2", True
    email.FormData.FormFields.Add "Field3", Date

    '******************************
    'Compose the mail message Body.
    '******************************
    msg.Body = email.ComposeBody

    MsgBox "Defined the form " + email.FormData.Name + " with " +
CStr(email.FormData.FormFields.Count) + " fields."

End Sub
```

See Also

Add, **Address**, **ComposeBody**, **ContentType**, **FormFields**, **MSMQMailEMail**, **MSMQMessage**, **Name**, **Recipients**, **Sender**, **Subject**

FormFields

MSMQMailFormData

The **FormFields** property specifies a list of the fields in the e-mail form.

Quick Info

Type:	**MSMQMailFormFieldList**
Run time:	Read-write

Syntax

*object1.object2.***FormFields**

Syntax Element	Description
object1	E-mail message (**MSMQMailEMail**) object that defines the e-mail message.
object2	The form data (**MSMQMailFormData**) object that defines the form.

Settings

MSMQMailFormFieldList object.

Remarks

The **MSMQMailFormFieldList** object specifies all the fields in the form

Example

This example defines an e-mail form with three fields, composes the body of an MSMQ mail message, then displays a name of the form plus the number of fields in the form.

To try this example using Microsoft Visual Basic (version 5.0), paste the code into the Code window of a form, and then run the example and click the form.

```
Option Explicit
Dim email As New MSMQMailEMail
Dim msg As New MSMQMessage

Private Sub Form_Click()

    '**********************
    '*   Define e-mail
    '**********************

    'Set e-mail type as form.
    email.ContentType = MSMQMAIL_EMAIL_FORM

    'Set who receives the e-mail.
    email.Recipients.Add "RecipientName", "RecipientQueueLabel",
MSMQMAIL_RECIPIENT_TO

    'Set who sent the e-mail.
    email.Sender.Name = "Our name"
    email.Sender.Address = "Our queue label"

    'Set the subject of the e-mail.
    email.Subject = "Test mail."
```

```
    '**********************
    '* Define the form.
    '**********************

    'Set the form name.
    email.FormData.Name = "Test Form"

    'Set fields of form
    email.FormData.FormFields.Add "Field1", "Field1 text."
    email.FormData.FormFields.Add "Field2", True
    email.FormData.FormFields.Add "Field3", Date

    '******************************
    'Compose the mail message Body.
    '******************************
    msg.Body = email.ComposeBody

    MsgBox "Defined the form " + email.FormData.Name + " with " +
CStr(email.FormData.FormFields.Count) + " fields."

End Sub
```

See Also

Add, **Address**, **ComposeBody**, **ContentType**, **FormFields**, **MSMQMailEMail**, **MSMQMessage**, **Name**, **Recipients**, **Sender**, **Subject**

Item

MSMQMailFormFieldList **MSMQMailRecipientList**

Read-only. The **Item** property provides read access to the recipients of an e-mail's recipient list (**MSMQMailRecipientList**) or the fields of a form's field list (**MSMQMailFormFieldList**).

Quick Info

Type:	**MSMQMailRecipient** or **MSMQMailFormField** object.
Run time:	Read-only

Syntax for Recipient List

set object3=object1.Object2.**Item** (*IndexKey*)

Syntax Element	Description
object1	E-mail message (**MSMQMailEMail**) object that defines the e-mail message.
object2	Recipient list (**MSMQMailRecipientList**) object.
object3	Recipient (**MSMQMailRecipient**) object.
IndexKey	Specifies the position in the recipient list or a specific recipient's key. Numeric indexes range from 1 to **Count**. Keys are added when the recipient is added to the list.

Syntax for Form Field List

set object4=object1.Object2.Object3.**Item** (*IndexKey*)

Syntax Element	Description
object1	E-mail message (**MSMQMailEMail**) object that defines the e-mail message.
object2	Form data (MSMQMailFormData) object that defines the form.
object3	Form field list (**MSMQMailFormFieldList**) object that references the field.
object4	Form field (**MSMQMailFormField**) object that defines the field.
IndexKey	Specifies the position in the field list or a specific field's key. Numeric indexes range from 1 to **Count**. Keys are added when the recipient is added to the list.

Return Values

MSMQMailRecipient or **MSMQMailFormField** object.

Remarks

Item is the default property for the **MSMQMailRecipientList** and **MSMQMailFormFieldList** objects. Thus RecipientList[2] and RecipientList.Item[2] both return the second recipient in the recipient list.

Example of Recipient List

This example adds three recipients, then displays the address of the MAPI recipient using a recipient key.

To try this example using Microsoft Visual Basic (version 5.0), paste the code into the Code window of a form, and then run the example and click the form.

```
Dim email As New MSMQMailEMail
Dim recipient As New MSMQMailRecipient

Private Sub Form_Click()

    '****************
    '* Define eimail
    '****************

    'Set e-mail type to text message
    email.ContentType = MSMQMAIL_EMAIL_TEXTMESSAGE

    'Add Recipients.
    email.Recipients.Add "Exchange_User", "ExchangeUser@ServerInputQueueLabel",
MSMQMAIL_RECIPIENT_TO
    email.Recipients.Add "MAPI_User", "MAPIUserInputQueueLabel",
MSMQMAIL_RECIPIENT_CC, "MAPI"
    email.Recipients.Add "Exchange_User", "ExchangeUser@ServerInputQueueLabel",
MSMQMAIL_RECIPIENT_BCC

    'Set who is sending the e-mail.
    email.Sender.Name = "Our name"
    email.Sender.Address = "Our queue label"

    'Set the subject of the e-mail.
    email.Subject = "Test mail."

    'Set the Body of the e-mail.
    email.TextMessageData.Text = "This is the Body of the message."

    '*********************************
    '* Display Name of MAPI recipient.
    '*********************************

    Set recipient = email.Recipients.Item("MAPI")
    MsgBox "The MAPI recipient's address is: " + recipient.Address

End Sub
```

Example of Form Field List

This example defines a form with three fields, then displays the name of the second field using the field key specified when the field was added.

To try this example using Microsoft Visual Basic (version 5.0), paste the code into the Code window of a form, and then run the example and click the form.

```
Dim email As New MSMQMailEMail
Dim msg As New MSMQMessage
Dim field As New MSMQMailFormField
```

```
Private Sub Form_Click()

    '**********************
    '*   Define e-mail
    '**********************

    'Set e-mail type to form message.
    email.ContentType = MSMQMAIL_EMAIL_FORM

    'Add primary recipient.
    email.Recipients.Add "Exchange_User", "ExchangeUser@ServerInputQueueLabel",
MSMQMAIL_RECIPIENT_TO

   'Set who sent the e-mail.
   email.Sender.Name = "Our name"
   email.Sender.Address = "Our queue label"

   'Set subject of mail.
   email.Subject = "Test form."

   'Set form name.
   email.FormData.Name = "Test Form"

   'Set form field list.
   email.FormData.FormFields.Add "Name", "Test Field"
   email.FormData.FormFields.Add "Employed", True, "Employed"
   email.FormData.FormFields.Add "CurrentDate", "DateString"

   '************************
   '* Compose message Body.
   '************************
   msg.Body = email.ComposeBody



   '*****************************************
   '* Display the value of the "Employed field.
   '*****************************************

    Set field = email.FormData.FormFields.Item("Employed")
    MsgBox "The value of the Employment field is: " + CStr(field.Value)

End Sub
```

See Also

Add, **Address**, **ContentType**, **FormData**, **FormFields**, **MSMQMailEMail**, **MSMQMailFormData**, **MSMQMailRecipient**, **MSMQMessage**, **Name**, **Recipients**, **Sender**, **Subject**, **Text**,**TextMessageData**

Name

MSMQMailFormData **MSMQMailFormField** **MSMQMailRecipient**

The **Name** property identifies the name of a form (**MSMQMailFormData**), the name of a field on a form(**MSMQMailFormField**), or the name of an e-mail recipient(**MSMQMailRecipient**).

Quick Info

Type:	**String**
Run time:	Read-write

Syntax for form name

object1.object2.**Name**

Syntax Element	Description
object1	An e-mail (**MSMQMailEMail**) object that defines an e-mail message.
object2	Form data (**MSMQMailFormData**) object that defines the form.

Syntax for field name

object1.object2.object3.**Item**(*IndexKey*),**Name**

Syntax Element	Description
object1	An e-mail (**MSMQMailEMail**) object that defines an e-mail message.
object2	Form data (**MSMQMailFormData**) object that defines the form.
object3	Form field (**MSMQMailFormField**) object that defines the field.
IndexKey	Specifies the position in the field list or a specific field key. Numeric indexes range from 1 to **Count**. Keys are added when the field is added to the list.

Syntax for sender recipient name

object1.object2.**Name**

Syntax Element	Description
object1	An e-mail (**MSMQMailEMail**) object that defines an e-mail message.
object2	Recipient (**MSMQMailRecipient**) object that defines the e-mail user.

Syntax for recipient name (in recipient list)

object1.object2.**Item**(*IndexKey*).**Name**

Syntax Element	Description
object1	An e-mail (**MSMQMailEMail**) object that defines an e-mail message.

object2	Recipients (**MSMQMailRecipientList**) object that defines the list of e-mail user.
IndexKey	Specifies the position in the recipient list or a specific field key. Numeric indexes range from 1 to **Count**. Keys are added when the recipient is added to the list.

Settings

String that specifies the name of the form, form field, or e-mail recipient.

Remarks

Form Name

When sending a form to an Exchange user, this name should match the name of the field's *Item Type* as specified in the Exchange Form Designer. The *Item Type* can be found on the *General* page of the *Field Properties* dialog of the Exchange Form Designer. If they do not match, Microsoft Exchange may not display the form correctly.

Form Field Name

When used to specify the field of a form, this name should correspond to the name of the field's *Reference Name* as specified in the Exchange Form Designer. The *Reference Name* can be found on the *General* page of the *Field Properties* dialog of the Exchange Form Designer

If the correct *Reference Name* is not used, Microsoft Exchange may not display the field correctly.

E-mail Recipient Name

When used to specify an e-mail user, **Name** is only used for display purposes. It should contain the full name of the user.

Name is explicitly referenced to name the sender recipient. However, it is not referenced explicitly when naming a recipient in the recipient list. In these cases, the name is set when adding the recipient to the recipient list.

Example

This example defines an e-mail message as a form, setting the sender's name to "Our name", the recipient's name to "Exchange user", the form name to "Test form", and field name to "StringField". Then a message body is composed and all names are displayed.

To try this example using Microsoft Visual Basic (version 5.0), paste the code into the Code window of a form, then run the example and click the form.

```
Dim email As New MSMQMailEMail
Dim msg As New MSMQMessage

Private Sub Form_Click()

    '**********************
    '*   Define e-mail
    '**********************

    'Set e-mail type to form message.
    email.ContentType = MSMQMAIL_EMAIL_FORM

    'Set who sent the e-mail.
    email.Sender.Name = "Our name"
    email.Sender.Address = "Our queue label"

    'Add Recipients.
```

```
    email.Recipients.Add "Exchange user", "UserAlias@ServerInputQueueLabel",
MSMQMAIL_RECIPIENT_TO, "Test"

    'Set subject of mail.
    email.Subject = "Test mail"

    '****************
    '*   Define form
    '****************

    'Set name of form
    email.FormData.Name = "Test form"

    'Set form field list.
    email.FormData.FormFields.Add "StringField", "Test Field", "Test"

    'Compose message Body
    msg.Body = email.ComposeBody


    '******************
    '* Display names.
    '******************

    MsgBox "Form name: " + email.FormData.Name
    MsgBox "Field name is: " + email.FormData.FormFields.Item("test").Name
    MsgBox "Sender recipient is: " + email.Sender.Name
    MsgBox "To recipient is: " + email.Recipients.Item("test").Name

End Sub
```

See Also

Add, **Address**, **Body**, **ComposeBody**, **ContentType**, **FormData**, **FormFields**, **Item**, **MSMQMailEMail**, **MSMQMessage**, **Recipients**, **Sender**, **Subject**

NonDeliveredRecipients

MSMQMailNonDeliveryReportData

The **NonDeliveredRecipients** property specifies who did not receive the e-mail.

Quick Info

Type:	**MSMQMailRecipientList**
Run time:	Read-write

Syntax

object1.object2.NonDeliveredRecipients

Syntax Element	Description
Object1	E-mail (**MSMQMailEMail**) object that defines the non-delivered report.
Object2	Non-Delivery report data (**MSMQMailNonDeliveryReportData**) object that represents the non-delivery report information.

Settings

MSMQMailRecipientList object.

Remarks

Each recipient in the non-delivered recipient list is represented by an **MSMQMailRecipient** object. Each recipient object includes the name and address of the recipient, the input queue of the recipient, how the message was sent to recipient, plus the reason why the original message was not delivered.

The **NonDeliveryReason** property of the **MSMQMailRecipient** object specifies why the original e-mail was not delivered to the recipient.

Example

This example defines a non-delivery report, adding two non-delivered recipients to the non-delivered recipient list of the report. The e-mail object is composed into a mail message, then each non-delivered recipient is displayed.

To try this example using Microsoft® Visual Basic® (version 5.0), build an original email object in emailOrig, paste the code into the Code window of a form, enter valid user address for each recipient, then run the example and click the form.

```
Dim email As New MSMQMailEMail
Dim msg As New MSMQMessage

Private Sub Form_Click()

    '**********************
    '*   Define e-mail
    '**********************

    'Set e-mail type to non-delivery report.
    email.ContentType = MSMQMAIL_EMAIL_NON_DELIVERY_REPORT

    'Add the Recipient of the report (usually the original e-mail sender).
    email.Recipients.Add "Exchange_User", "UserAlias@ServerInputQueueLabel",
MSMQMAIL_RECIPIENT_TO
```

```
    'Set original mail.
    Set email.NonDeliveryReportData.OriginalEMail = emailOrig

     'Add two non-delivered recipients.
    email.NonDeliveryReportData.NonDeliveredRecipients.Add "Exchange_User2",
"UserAlias2@ServerInputQueueLabel", MSMQMAIL_RECIPIENT_TO, NonDeliveryReason:=
“Recipient was not available at this address”
email.NonDeliveryReportData.NonDeliveredRecipients.Add "Exchange_User3",
"UserAlias3@ServerInputQueueLabel", MSMQMAIL_RECIPIENT_CC, NonDeliveryReason:=
“Communication failure”

    'Compose message Body
    msg.Body = email.ComposeBody


    '**********************************
    '* Display non-delivered recipients.
    '**********************************

    Dim recipient As MSMQMailRecipient

    For Each recipient In email.NonDeliveryReportData.NonDeliveredRecipients
        MsgBox "Not Delivered To Recipient: " + recipient.Name + " at " +
recipient.Address + “, Reason is:” + recipient.NonDeliveryReason
    Next recipient

End Sub
```

See Also

Add, **Address**, **ComposeBody**, **ContentType**, **FormData**, **FormFields**, **MSMQMailEMail**, **Name**, **Sender**, **Subject**, **NonDeliveryReportData**

NonDeliveryReason

MSMQMailRecipient

The **NonDeliveryReason** property specifies why the original e-mail was not delivered.

Quick Info

Type:	**String**
Run time:	Read-write

Syntax

object.**NonDeliveryReason**

Syntax Element	Description
Object	A recipient (**MSMQMailRecipient**) object that defines an e-mail recipient.

Settings

A free text which describes the reason for not delivering the original e-mail to this recipient.

Remarks

NonDeliveryReason has a valid value only when the recipient is a member of the **NonDeliveredRecipients** property of a non-delivery report e-mail.

Example

This example defines a non-delivery report, adding two non-delivered recipients to the non-delivered recipient list of the report. The e-mail object is composed into a mail message, then each non-delivered recipient is displayed.

To try this example using Microsoft® Visual Basic® (version 5.0), build an original email object in emailOrig, paste the code into the Code window of a form, enter valid user address for each recipient, then run the example and click the form.

```
Dim email As New MSMQMailEMail
Dim msg As New MSMQMessage

Private Sub Form_Click()

    '**********************
    '*   Define e-mail
    '**********************

    'Set e-mail type to non-delivery report.
    email.ContentType = MSMQMAIL_EMAIL_NON_DELIVERY_REPORT

    'Add the Recipient of the report (usually the original e-mail sender).
    email.Recipients.Add "Exchange_User", "UserAlias@ServerInputQueueLabel",
MSMQMAIL_RECIPIENT_TO

    'Set original mail.
    Set email.NonDeliveryReportData.OriginalEMail = emailOrig

     'Add two non-delivered recipients.
```

```
    email.NonDeliveryReportData.NonDeliveredRecipients.Add "Exchange_User2",
"UserAlias2@ServerInputQueueLabel", MSMQMAIL_RECIPIENT_TO, NonDeliveryReason:=
“Recipient was not available at this address”
email.NonDeliveryReportData.NonDeliveredRecipients.Add "Exchange_User3",
"UserAlias3@ServerInputQueueLabel", MSMQMAIL_RECIPIENT_CC, NonDeliveryReason:=
“Communication failure”

    'Compose message Body
    msg.Body = email.ComposeBody


    '**********************
    '* Display Delivered Recipients.
    '**********************

    Dim recipient As MSMQMailRecipient

    For Each recipient In email.NonDeliveryReportData.NonDeliveredRecipients
        MsgBox "Not Delivered To Recipient: " + recipient.Name + " at " +
recipient.Address + “, Reason is:” + recipient.NonDeliveryReason
    Next recipient

End Sub
```

See Also

Add, **Body**, **ComposeBody**, **ContentType**, **FormData**, **FormFields**, **MSMQMailEMail**, **MSMQMAILFormField**, **Name**, **ParseBody**, **Recipients**, **Subject**, **Text**, **TextMessageData**, **Value**

NonDeliveryReportData

MSMQMailEMail

The **NonDeliveryReportData** property defines a non-delivery report. The non-delivery report lists the recipients who did not receive the original e-mail.

This property is only meaningful if **ContentType** is set to MSMQMAIL_EMAIL_NON_DELIVERY_REPORT.

Quick Info

Type:	**MSMQMailNonDeliveryReportData**
Run time:	Read-write

Syntax

object.**NonDeliveryReportData**

Syntax Element	Description
object	E-mail message (**MSMQMailEMail**) object that defines the e-mail message.

Settings

MSMQMailNonDeliveryReportData object.

Remarks

When defining a non-delivery report, set **ContentType** to MSMQMAIL_EMAIL_NON_DELIVERY_REPORT whenever **NonDeliveryReportData** is set.

After parsing an MSMQ Mail message, verify that, **ContentType** is set to MSMQMAIL_EMAIL_NON_DELIVERY_REPORT , before looking at **NonDeliveryReportData**. This property is empty if **ContentType** indicates another e-mail type.

OriginalEmail

MSMQMailNonDeliveryReportData

The **OriginalEmail** property specifies the original e-mail in a non-delivery report e-mail.

Quick Info

Type:	**String**
Run time:	Read-write

Syntax

object1.object2.**OriginalEMail**

Syntax Element	Description
Object1	E-mail (**MSMQMailEMail**) object that defines the e-mail report.
Object2	Non-delivery report data (**MSMQMailNonDeliveryReportData**) object that represents the non-delivery report information.

Settings

The original e-mail which was not delivered.

Remarks

OriginalEMail should be set to the original e-mail.

Example

This example defines a non-delivery report, adding two non-delivered recipients to the non-delivered recipient list of the report. The e-mail object is composed into a mail message, then each non-delivered recipient is displayed.

To try this example using Microsoft® Visual Basic® (version 5.0), build an original email object, paste the code into the Code window of a form, enter valid user address for each recipient, then run the example and click the form.

```
Dim email As New MSMQMailEMail
Dim msg As New MSMQMessage

Private Sub Form_Click()

    '**********************
    '*   Define e-mail
    '**********************

    'Set e-mail type to non-delivery report.
    email.ContentType = MSMQMAIL_EMAIL_NON_DELIVERY_REPORT

    'Add the Recipient of the report (usually the original e-mail sender).
    email.Recipients.Add "Exchange_User", "UserAlias@ServerInputQueueLabel",
MSMQMAIL_RECIPIENT_TO

    'Set original mail.
    Set email.NonDeliveryReportData.OriginalEMail = emailOrig
```

```
    'Add two non-delivered recipients.
    email.NonDeliveryReportData.NonDeliveredRecipients.Add "Exchange_User2",
"UserAlias2@ServerInputQueueLabel", MSMQMAIL_RECIPIENT_TO, NonDeliveryReason:=
“Recipient was not available at this address”
email.NonDeliveryReportData.NonDeliveredRecipients.Add "Exchange_User3",
"UserAlias3@ServerInputQueueLabel", MSMQMAIL_RECIPIENT_CC, NonDeliveryReason:=
“Communication failure”

    'Compose message Body
    msg.Body = email.ComposeBody


    '*******************************
    '* Display non-delivered recipients.
    '*******************************

    Dim recipient As MSMQMailRecipient

    For Each recipient In email.NonDeliveryReportData.NonDeliveredRecipients
        MsgBox "Not Delivered To Recipient: " + recipient.Name + " at " +
recipient.Address + “, Reason is:” + recipient.NonDeliveryReason
    Next recipient

End Sub
```

See Also

Add, **Address**, **Body**, **ComposeBody**, **ContentType**, **MSMQMailEMail**, **MSMQMessage**, **Name**, **Recipients**, **Sender**, **Text**, **TextMessageData**

OriginalSubject

MSMQMailDeliveryReportData

The **OriginalSubject** property specifies the subject of the original e-mail in a delivery report e-mail.

Quick Info

Type:	**String**
Run time:	Read-write

Syntax

Object1.Object2.**OriginalSubject**

Syntax Element	Description
Object1	E-mail (**MSMQMailEMail**) object that defines the e-mail report.
Object2	Delivery report data (**MSMQMailDeliveryReportData**) object that represents the delivery report information.

Settings

String representation of the original e-mail message's subject.

Remarks

OriginalSubject should be set to the subject of the original e-mail.

Example

This example defines a delivery report, adding two delivered recipients to the delivered recipient list of the report. The e-mail object is composed into a mail message, then each delivered recipient is displayed.

To try this example using Microsoft® Visual Basic® (version 5.0), paste the code into the Code window of a form, enter valid user address for each recipient, then run the example and click the form.

```
Dim email As New MSMQMailEMail
Dim msg As New MSMQMessage

Private Sub Form_Click()

    '**********************
    '*   Define e-mail
    '**********************

    'Set e-mail type to delivery report.
    email.ContentType = MSMQMAIL_EMAIL_DELIVERY_REPORT

    'Add the Recipient of the report (usually the original e-mail sender).
    email.Recipients.Add "Exchange_User", "UserAlias@ServerInputQueueLabel",
MSMQMAIL_RECIPIENT_TO

    'Set subject of original mail.
    email.DeliveryReportData.OriginalSubject = "Original subject "

    'Set submission time of original mail.
    email.DeliveryReportData.OriginalSubmissionTIme = CDate("5/20/94 10:16:07 PM")
```

```
    'Add two delivered recipients.
    email.DeliveryReportData.DeliveredRecipients.Add "Exchange_User2",
"UserAlias2@ServerInputQueueLabel", MSMQMAIL_RECIPIENT_TO, DeliveryTime:=
CDate("5/20/94 10:17:00 PM")
email.DeliveryReportData.DeliveredRecipients.Add "Exchange_User3",
"UserAlias3@ServerInputQueueLabel", MSMQMAIL_RECIPIENT_CC, DeliveryTime:=
CDate("5/20/94 11:01:00 PM")

    'Compose message Body
    msg.Body = email.ComposeBody


    '**********************
    '* Display Delivered Recipients.
    '**********************

    Dim recipient As MSMQMailRecipient

    For Each recipient In email.DeliveryReportData.DeliveredRecipients
        MsgBox "Delivered To Recipient: " + recipient.Name + " at " + recipient.Address + “
on “ + recipient.DeliveryTime
    Next recipient

End Sub
```

See Also

Add, **Address**, **Body**, **ComposeBody**, **ContentType**, **MSMQMailEMail**, **MSMQMessage**, **Name**, **Recipients**, **Sender**, **Text**, **TextMessageData**

OriginalSubmissionTime

MSMQMailDeliveryReportData

The **OriginalSubmissionTime** property specifies the submission time of the original e-mail in a delivery report e-mail.

Quick Info

Type:	**Date**
Run time:	Read-write

Syntax

object1.object2.**OriginalSubmissionTime**

Syntax Element	Description
Object1	E-mail (**MSMQMailEMail**) object that defines the e-mail report.
Object2	Delivery report data (**MSMQMailDeliveryReportData**) object that represents the delivery report information.

Settings

Date representation of the original e-mail message's submission time.

Remarks

OriginalSubmissionTime should be set to the **SubmissionTime** property of the original e-mail.

Example

This example defines a delivery report, adding two delivered recipients to the delivered recipient list of the report. The e-mail object is composed into a mail message, then each delivered recipient is displayed.

To try this example using Microsoft® Visual Basic® (version 5.0), paste the code into the Code window of a form, enter valid user address for each recipient, then run the example and click the form.

```
Dim email As New MSMQMailEMail
Dim msg As New MSMQMessage

Private Sub Form_Click()

    '**********************
    '*   Define e-mail
    '**********************

    'Set e-mail type to delivery report.
    email.ContentType = MSMQMAIL_EMAIL_DELIVERY_REPORT

    'Add the Recipient of the report (usually the original e-mail sender).
    email.Recipients.Add "Exchange_User", "UserAlias@ServerInputQueueLabel",
MSMQMAIL_RECIPIENT_TO

    'Set subject of original mail.
    email.DeliveryReportData.OriginalSubject = "Original subject "

    'Set submission time of original mail.
```

```
    email.DeliveryReportData.OriginalSubmissionTime = CDate("5/20/94 10:16:07 PM")

     'Add two delivered recipients.
    email.DeliveryReportData.DeliveredRecipients.Add "Exchange_User2",
"UserAlias2@ServerInputQueueLabel", MSMQMAIL_RECIPIENT_TO, DeliveryTime:=
CDate("5/20/94 10:17:00 PM")
email.DeliveryReportData.DeliveredRecipients.Add "Exchange_User3",
"UserAlias3@ServerInputQueueLabel", MSMQMAIL_RECIPIENT_CC, DeliveryTime:=
CDate("5/20/94 11:01:00 PM")

    'Compose message Body
    msg.Body = email.ComposeBody


    '**********************
    '* Display Delivered Recipients.
    '**********************

    Dim recipient As MSMQMailRecipient

    For Each recipient In email.DeliveryReportData.DeliveredRecipients
        MsgBox "Delivered To Recipient: " + recipient.Name + " at " + recipient.Address + “
on “ + recipient.DeliveryTime
    Next recipient

End Sub
```

See Also

Add, **Address**, **Body**, **ComposeBody**, **ContentType**, **MSMQMailEMail**, **MSMQMessage**, **Name**, **Recipients**, **Sender**, **Text**, **TextMessageData**

Recipients

MSMQMailEMail

The **Recipients** property specifies the intended list of recipients for the e-mail message.

Quick Info

Type:	**MSMQMailRecipientList**
Run time:	Read-write

Syntax

object.**Recipients**

Syntax Element	Description
object	An e-mail (**MSMQMailEMail**) object that defines an e-mail message.

Settings

MSMQMailRecipientList object.

Remarks

Each recipient in the e-mail's recipient list is represented by an **MSMQMailRecipient** object. Each recipient object includes a name and address for the recipient, an input queue of the recipient, plus how the message is sent to recipient.

Example

This example defines an e-mail form, adding three recipients to the e-mail's recipient list. The e-mail object is composed into a mail message, then each recipient is displayed.

To try this example using Microsoft® Visual Basic® (version 5.0), paste the code into the Code window of a form, enter valid user address for each recipient, then run the example and click the form.

```
Dim email As New MSMQMailEMail
Dim msg As New MSMQMessage

Private Sub Form_Click()

    '**********************
    '*   Define e-mail
    '**********************

    'Set e-mail type to form message.
    email.ContentType = MSMQMAIL_EMAIL_FORM

    'Add Recipients.
    email.Recipients.Add "Exchange_User", "UserAlias@ServerInputQueueLabel",
MSMQMAIL_RECIPIENT_TO
    email.Recipients.Add "MAPI_User", "MAPIUserInputQueueLabel",
MSMQMAIL_RECIPIENT_CC
    email.Recipients.Add "MSMQApplication", "ApplicationInputQueueLabel",
MSMQMAIL_RECIPIENT_BCC

    'Set who sent the e-mail.
    email.Sender.Name = "Our name"
    email.Sender.Address = "Our queue label"
```

```
    'Set subject of mail.
    email. = "Test mail"

    'Set name of form
    email.FormData.Name = "Test form"

    'Set form field list.
    email.FormData.FormFields.Add "StringField", "Test Field"

    'Compose message Body
    msg.Body = email.ComposeBody


    '**********************
    '* Display Recipients.
    '**********************

    Dim recipient As MSMQMailRecipient

    For Each recipient In email.Recipients
        MsgBox "Recipient: " + recipient.Name + " at " + recipient.Address
    Next recipient

End Sub
```

See Also

Add, **Address**, **ComposeBody**, **ContentType**, **FormData**, **FormFields**, **MSMQMailEMail**, **Name**, **Sender**, **Subject**

RecipientType

MSMQMailRecipient

The **RecipientType** property specifies how the e-mail is sent (**MSMQMailRecipient**) to the recipient.

Quick Info

Type:	**Long**
Run time:	Read-write

Syntax

object.**RecipientType**

Syntax Element	Description
object	A recipient (**MSMQMailRecipient**) object that defines an e-mail recipient.

Settings

RecipientType can have any one of the following values:

MSMQMailRecipient_TO
Default. The recipient is the primary recipient of the e-mail.

MSMQMailRecipient_CC
The e-mail is copied to the recipient.

MSMQMailRecipient_BCC
The e-mail is blind copied to the recipient.

Remarks

This property is automatically set whenever a recipient is added to the recipient list of an e-mail object (see the *Type* parameter of **Add**). Consequently, explicitly setting this property is seldom required.

Type is ignored if the recipient (**MSMQMailRecipient**) object represents the sender of an e-mail.

Example

This example composes a message body from an e-mail object with three recipients, then parses the message body and displays all the recipients who received a copy of the message.

To try this example using Microsoft Visual Basic (version 5.0), paste the code into the Code window of a form, and then run the example and click the form.

```
Dim email As New MSMQMailEMail
Dim emailReceived As New MSMQMailEMail
Dim recipient As MSMQMailRecipient
Dim msg As New MSMQMessage

Private Sub Form_Click()

    '**********************
    '*   Define e-mail
    '**********************

    'Set e-mail type to form message.
    email.ContentType = MSMQMAIL_EMAIL_FORM

    'Add primary recipient.
```

```
    email.Recipients.Add "Exchange_User1", "ExchangeUser@ServerInputQueueLabel",
MSMQMAIL_RECIPIENT_TO
    email.Recipients.Add "Exchange_User2", "ExchangeUser@ServerInputQueueLabel",
MSMQMAIL_RECIPIENT_CC
    email.Recipients.Add "Exchange_User3", "ExchangeUser@ServerInputQueueLabel",
MSMQMAIL_RECIPIENT_CC

    'Set who sent the e-mail.
    email.Sender.Name = "Our name"
    email.Sender.Address = "Our queue label"

    'Set subject of mail.
    email.Subject = "Test form."

    'Set form name.
    email.FormData.Name = "Test form."

    'Set form field list.
    email.FormData.FormFields.Add "StringField", "Test Field"

    '**************************
    '* Compose and Parse message
    '**************************
    msg.Body = email.ComposeBody
    emailReceived.ParseBody (msg.Body)

    For Each recipient In email.Recipients
        If recipient.RecipientType = MSMQMAIL_RECIPIENT_CC Then
            MsgBox "Mail was copied to: " + recipient.Name
        End If
    Next recipient

End Sub
```

See Also

Add, **Body**, **ComposeBody**, **ContentType**, **FormData**, **FormFields**, **MSMQMailEMail**, **MSMQMAILFormField**, **Name**, **ParseBody**, **Recipients**, **Subject**, **Text**, **TextMessageData**, **Value**

RequestDeliveryReport

MSMQMailEMail

The **RequestDeliveryReport** property specifies whether receiving application should return a delivery report when the e-mail is received.

Quick Info

Type:	**Boolean**
Run time:	Read-write

Syntax

object.**RequestDeliveryReport**

Syntax Element	Description
object	E-mail (**MSMQMailEMail**) object that represents the e-mail message.

Settings

Boolean (default is False).

Remarks

If **RequestDeliveryReport** is set to True, the receiving application should send a delivery report e-mail for the delivered recipients.

The default setting of this property is False.

Example

This example defines an text-message e-mail, setting its **RequestDeliveryReport** to True. The email object is then used to compose the body of a mail message, and a message box displays the e-mail's subject.

To try this example using Microsoft Visual Basic (version 5.0), paste the code into the Code window of a form, then run the example and click the form.

```
Dim email As New MSMQMailEMail
Dim msg As New MSMQMessage

Private Sub Form_Click()

    '************************
    '*   Define e-mail object
    '************************

    'Set e-mail type to text message
    email.ContentType = MSMQMAIL_EMAIL_TEXTMESSAGE

    'Add Recipients
    email.Recipients.Add "Connector Recipient Name",
"ExchangeUser@ServerInputQueueLabel", MSMQMAIL_RECIPIENT_TO
    email.Recipients.Add "MAPI Recipient Name", "MAPIUserInputQueueLabel",
MSMQMAIL_RECIPIENT_CC

    'Set who sent the e-mail.
    email.Sender.Name = "Our name"
    email.Sender.Address = "Our queue label"
```

```
    'Set the subject of the e-mail.
    email.Subject = "Test mail."

    'Request delivery report.
    email.RequestDeliveryReport = True

    'Set the Body of the e-mail.
    email.TextMessageData.Text = "This is the Body of the message."

    '**********************
    '* Compose message Body
    '**********************
    msg.Body = email.ComposeBody

    '******************
    '* Display delivery report flag.
    '******************
    MsgBox "Request a delivery report for the e-mail:" + email.RequestDeliveryReport

End Sub
```

See Also

Add, **Address**, **Body**, **ComposeBody**, **ContentType**, **MSMQMailEMail**, **MSMQMessage**, **Name**, **Recipients**, **Sender**, **Text**, **TextMessageData**

RequestNonDeliveryReport

MSMQMailEMail

The **RequestNonDeliveryReport** property specifies whether a non-delivery report is sent back for the recipients that did not receive the e-mail.

Quick Info

Type:	**Boolean**
Run time:	Read-write

Syntax

object.**RequestNonDeliveryReport**

Syntax Element	Description
object	E-mail (**MSMQMailEMail**) object that represents the e-mail message.

Settings

Boolean (default is False).

Remarks

If **RequestNonDeliveryReport** is set to True, the sender will receive a non-delivery report for the recipients that did not receive the e-mail.

Example

This example defines an text-message e-mail, setting its **RequestNonDeliveryReport** to True. The email object is then used to compose the body of a mail message, and a message box displays the e-mail's non-delivery report request.

To try this example using Microsoft Visual Basic (version 5.0), paste the code into the Code window of a form, then run the example and click the form.

```
Dim email As New MSMQMailEMail
Dim msg As New MSMQMessage

Private Sub Form_Click()

    '************************
    '*   Define e-mail object
    '************************

    'Set e-mail type to text message
    email.ContentType = MSMQMAIL_EMAIL_TEXTMESSAGE

    'Add Recipients
    email.Recipients.Add "Connector Recipient Name",
"ExchangeUser@ServerInputQueueLabel", MSMQMAIL_RECIPIENT_TO
    email.Recipients.Add "MAPI Recipient Name", "MAPIUserInputQueueLabel",
MSMQMAIL_RECIPIENT_CC

    'Set who sent the e-mail.
    email.Sender.Name = "Our name"
    email.Sender.Address = "Our queue label"
```

```
    'Set the subject of the e-mail.
    email.Subject = "Test mail."

    'Request non-delivery report.
    email.RequestNonDeliveryReport = True

    'Set the Body of the e-mail.
    email.TextMessageData.Text = "This is the Body of the message."

    '**********************
    '* Compose message Body
    '**********************
    msg.Body = email.ComposeBody

    '******************
    '* Display delivery report flag.
    '******************
    MsgBox "Request a non-delivery report for the e-mail:" +
email.RequestNonDeliveryReport

End Sub
```

See Also

Add, **Address**, **Body**, **ComposeBody**, **ContentType**, **MSMQMailEMail**, **MSMQMessage**, **Name**, **Recipients**, **Sender**, **Text**, **TextMessageData**

Sender

MSMQMailEMail

The **Sender** property specifies who is sending the e-mail message. It includes the sender's name and address.

Quick Info

Type:	**MSMQMailRecipient**
Run time:	Read-write

Syntax

object.**Sender**

Syntax Element	Description
object	An e-mail (**MSMQMailEMail**) object that defines an e-mail message.

Settings

MSMQMailRecipient object.

Remarks

When setting the **Sender** property, the **RecipientType** property of the **MSMQMailRecipient** object can be ignored. The **RecipientType** property has no meaning when specifying who sent the message.

Sender can be used to create a reply or reply-all e-mail. To do this, add the **Sender** to the recipient list of the reply e-mail.

Example

This example defines an e-mail form, specifying who sent the message. The e-mail object is composed into a mail message, then the sender recipient is displayed.

To try this example using Microsoft Visual Basic (version 5.0), paste the code into the Code window of a form, enter valid user address for each recipient, then run the example and click the form.

```
Dim email As New MSMQMailEMail
Dim msg As New MSMQMessage

Private Sub Form_Click()

    '**********************
    '*   Define e-mail
    '**********************

    'Set e-mail type to form message.
    email.ContentType = MSMQMAIL_EMAIL_FORM

    'Add Recipients.
    email.Recipients.Add "Exchange_User", "UserAlias@ServerInputQueueLabel",
MSMQMAIL_RECIPIENT_TO

    'Set who sent the e-mail.
    email.Sender.Name = "Sender"
    email.Sender.Address = "Sender Application Input Queue"

    'Set subject of mail.
```

```
    email.Subject = "Test mail"

    'Set name of form
    email.FormData.Name = "Test form"

    'Set form field list.
    email.FormData.FormFields.Add "StringField", "Test Field"

    'Compose message Body
    msg.Body = email.ComposeBody


    '**************************
    '* Display recipient Sender.
    '**************************

    MsgBox "Recipient sender: " + email.Sender.Name + " at " + email.Sender.Address

End Sub
```

See Also

Add, **Address**, **ComposeBody**, **ContentType**, **FormData**, **FormFields**, **MSMQMailEMail**, **Name**, **Sender**, **Subject**

Subject

MSMQMailEMail

The **Subject** property specifies the subject of the e-mail.

Quick Info

Type:	**String**
Run time:	Read-write

Syntax

object.**Subject**

Syntax Element	Description
object	E-mail (**MSMQMailEMail**) object that represents the e-mail message.

Settings

String representation of the e-mail message's subject.

Remarks

Subject can be set to any valid string. There are no restrictions on this property.

Example

This example defines an text-message e-mail, setting its subject to “Test mail.” The email object is then used to compose the body of a mail message, and a message box displays the e-mail’s subject.

To try this example using Microsoft Visual Basic (version 5.0), paste the code into the Code window of a form, then run the example and click the form.

```
Dim email As New MSMQMailEMail
Dim msg As New MSMQMessage

Private Sub Form_Click()

    '************************
    '*   Define e-mail object
    '************************

    'Set e-mail type to text message
    email.ContentType = MSMQMAIL_EMAIL_TEXTMESSAGE

    'Add Recipients
    email.Recipients.Add "Connector Recipient Name",
"ExchangeUser@ServerInputQueueLabel", MSMQMAIL_RECIPIENT_TO
    email.Recipients.Add "MAPI Recipient Name", "MAPIUserInputQueueLabel",
MSMQMAIL_RECIPIENT_CC

    'Set who sent the e-mail.
    email.Sender.Name = "Our name"
    email.Sender.Address = "Our queue label"

    'Set the subject of the e-mail.
    email.Subject = "Test mail."
```

```
    'Set the Body of the e-mail.
    email.TextMessageData.Text = "This is the Body of the message."

    '**********************
    '* Compose message Body
    '**********************
    msg.Body = email.ComposeBody

    '******************
    '* Display subject.
    '******************
    MsgBox "Subject of e-mail is:" + email.Subject

End Sub
```

See Also

Add, **Address**, **Body**, **ComposeBody**, **ContentType**, **MSMQMailEMail**, **MSMQMessage**, **Name**, **Recipients**, **Sender**, **Text**, **TextMessageData**

SubmissionTime

MSMQMailEMail

The **SubmissionTime** property specifies when the e-mail object was submitted.

Quick Info

Type:	**Date**
Run time:	Read-write

Syntax

object.**SubmissionTime**

Syntax Element	Description
object	E-mail (**MSMQMailEMail**) object that defines the e-mail message.

Settings

Date e-mail was submitted.

Remarks

Typically, mail is sent at the same time the e-mail object is submitted. However, some applications may need to store the e-mail objects (such as when communication is broken) and send the actual mail at another time.

The returned value for this property can be manipulated using standard Microsoft® Visual Basic® date and time functions such as **Date$**, and **Time$**. For descriptions of Visual Basic functions, see Visual Basic documentation.

When **SubmissionTime** is displayed, Visual Basic will automatically convert the returned value to the local system time and system date.

Example

This example defines an e-mail message as a form, then displays the date and time when the e-mail object was submitted.

To try this example using Microsoft Visual Basic (version 5.0), paste the code into the Code window of a form, and then run the example and click the form.

```
Dim email As New MSMQMailEMail

Private Sub Form_Click()

    '*****************
    '* Define eimail
    '*****************

    'Set e-mail type to text message
    email.ContentType = MSMQMAIL_EMAIL_TEXTMESSAGE

    'Add Recipients.
    email.Recipients.Add "Exchange_User", "ExchangeUser@ServerInputQueueLabel",
MSMQMAIL_RECIPIENT_TO

    'Set who is sending the e-mail.
    email.Sender.Name = "Our name"
```

```
    email.Sender.Address = "Our queue label"

    'Set the subject of the e-mail.
    email.Subject = "Test mail."

    'Set the Body of the e-mail.
    email.TextMessageData.Text = "This is the Body of the message."


    '************************
    '* Display SubmissionTime time.
    '************************

    MsgBox "The e-mail object was submitted at: " + CStr(email.SubmissionTime)

End Sub
```

See Also

Add, **Address**, **ContentType**, **MSMQMailEMail**, **Name**, **Recipients**, **Sender**, **Subject**, **Text**, **TextMessageData**

Text

MSMQMailTextMessageData

The **Text** property specifies the text of a text message e-mail.

Quick Info

Type:	String
Run time:	Read-write

Syntax

object1.object2.**Text**

Syntax Element	Description
object1	E-mail (**MSMQMailEMail**) object that defines the e-mail message.
object2	Text message data (**MSMQMailTextMessageData**) object that represents the text message.

Settings

String representation of message body.

Example

This example defines an e-mail object as a text message, setting the text message body to “This is a text message". The email object is then used to compose the body of a mail message, and a message box displays the text of the mail.

To try this example using Microsoft Visual Basic (version 5.0), paste the code into the Code window of a form, then run the example and click the form.

```
Dim email As New MSMQMailEMail
Dim msg As New MSMQMessage

Private Sub Form_Click()

    '************************
    '*   Define e-mail object
    '************************

    'Set e-mail type to text message
    email.ContentType = MSMQMAIL_EMAIL_TEXTMESSAGE

    'Add Recipients
    email.Recipients.Add "Connector Recipient Name",
"ExchangeUser@ServerInputQueueLabel", MSMQMAIL_RECIPIENT_TO
    email.Recipients.Add "MAPI Recipient Name", "MAPIUserInputQueueLabel",
MSMQMAIL_RECIPIENT_CC

    'Set who sent the e-mail.
    email.Sender.Name = "Our name"
    email.Sender.Address = "Our queue label"

    'Set the subject of the e-mail.
    email.Subject = "Test mail."
```

```
    'Set the Body of the e-mail.
    email.TextMessageData.Text = "This is the Body of the message."

    '**********************
    '* Compose message Body
    '**********************
    msg.Body = email.ComposeBody

    '******************
    '* Display subject.
    '******************
    MsgBox "The text of the message is:" + email.TextMessageData.Text

End Sub
```

See Also

Add, **Address**, **Body**, **ComposeBody**, **ContentType**, **MSMQMailEMail**, **MSMQMessage**, **Name**, **Recipients**, **Sender**, **Subject**, **TextMessageData**

TextMessageData

MSMQMailEMail

The **TextMessageData** property defines a text-message.

This property is only meaningful if **ContentType** is set to MSMQMAIL_EMAIL_TEXTMESSAGE.

Quick Info

Type:	**MSMQMailTextMessageData**
Run time:	Read-write

Syntax

object.**TextMessageData**

Syntax Element	Description
object	E-mail message (**MSMQMailEMail**) object that defines the text message.

Settings

MSMQMailTextMessageData object.

Remarks

When composing the body of an MSMQ mail message that represent a text message, set **ContentType** to MSMQMAIL_EMAIL_TEXTMESSAGE.

When parsing the body of an MSMQ message, verify that **ContentType** is set to MSMQMAIL_EMAIL_TEXTMESSAGE, before looking at **TextMessageData**. This property is empty if **ContentType** indicates another e-mail type.

Example

This example defines an e-mail object as a text message, setting the text message body to “This is a text message". The email object is then used to compose the body of a mail message, and a message box displays the text of the mail.

To try this example using Microsoft Visual Basic (version 5.0), paste the code into the Code window of a form, then run the example and click the form.

```
Dim email As New MSMQMailEMail
Dim msg As New MSMQMessage

Private Sub Form_Click()

    '************************
    '*   Define e-mail object
    '************************

    'Set e-mail type to text message
    email.ContentType = MSMQMAIL_EMAIL_TEXTMESSAGE

    'Add Recipients
    email.Recipients.Add "Connector Recipient Name",
"ExchangeUser@ServerInputQueueLabel", MSMQMAIL_RECIPIENT_TO
    email.Recipients.Add "MAPI Recipient Name", "MAPIUserInputQueueLabel",
MSMQMAIL_RECIPIENT_CC
```

```
    'Set who sent the e-mail.
    email.Sender.Name = "Our name"
    email.Sender.Address = "Our queue label"

    'Set the subject of the e-mail.
    email.Subject = "Test mail."

    'Set the Body of the e-mail.
    email.TextMessageData.Text = "This is the Body of the message."

    '**********************
    '* Compose message Body
    '**********************
    msg.Body = email.ComposeBody

    '******************
    '* Display subject.
    '******************
    MsgBox "The text of the message is:" + email.TextMessageData.Text

End Sub
```

See Also

Add, **Address**, **Body**, **ComposeBody**, **ContentType**, **MSMQMailEMail**, **MSMQMessage**, **Name**, **Recipients**, **Sender**, **Subject**, **Text**

TnefData

The **TnefData** property defines the TNEF data.

This property is only meaningful if **ContentType** is set to MSMQMAIL_EMAIL_TNEF.

Quick Info

Type:	**MSMQMailTnefData**
Run time:	Read-write

Syntax

object.**TnefData**

Syntax Element	Description
object	E-mail message (**MSMQMailEMail**) object that defines the e-mail message.

Settings

MSMQMailTnefData object.

Remarks

When defining TNEF data, set **ContentType** to MSMQMAIL_EMAIL_TNEF whenever **TnefData** is set.

After parsing an MSMQ Mail message, verify that, **ContentType** is set to MSMQMAIL_EMAIL_TNEF before looking at **TnefData**. This property is empty if **ContentType** indicates another e-mail type.

Value

MSMQMailFormField

The **Value** property specifies the value of a field.

Quick Info

Type:	**Variant**
Run time:	Read-write

Syntax

object.**Value**

Syntax Element	Description
object	Field (**MSMQMailFormField**) object that represents the field of the form.

Settings

String, Integer, Boolean, Double, or Currency value of field.

Example

This example defines an e-mail form that has three fields (String, Boolean, and Date). Then composes a message body and displays the values of each field.

To try this example using Microsoft Visual Basic (version 5.0), paste the code into the Code window of a form, then run the example and click the form.

```
Dim email As New MSMQMailEMail
Dim msg As New MSMQMessage

Private Sub Form_Click()

    '**********************
    '*   Define e-mail
    '**********************

    'Set e-mail type to form message.
    email.ContentType = MSMQMAIL_EMAIL_FORM

    'Add primary recipient.
    email.Recipients.Add "Exchange_User", "ExchangeUser@ServerInputQueueLabel",
MSMQMAIL_RECIPIENT_TO

    'Set who sent the e-mail.
    email.Sender.Name = "Our name"
    email.Sender.Address = "Our queue label"

    'Set subject of mail.
    email.Subject = "Test form."

    '***************
    '* Define form
    '***************

    'Set form name
```

```
    emaile.FormData.Name = "Our name"

    'Set form field list.
    email.FormData.FormFields.Add "StringField", "Test String"
    email.FormData.FormFields.Add "BooleanField", True
    email.FormData.FormFields.Add "DateField", "Current Date"

    '************************
    '* Compose mail message.
    '************************

    msq.Body = email.ComposeBody

    '************************
    '* Display Recipients.
    '************************

    Dim formfield As MSMQMailFormField

    For Each formfield In email.FormData.FormFields
        MsgBox "Form: " + formfield.Name + " = " + CStr(formfield.Value)
    Next formfield

End Sub
```

See Also

Add, **Address**, **Body**, **ComposeBody**, **ContentType**, **FormData**, **FormFields**, **MSMQMailEMail**, **MSMQMessage**, **Name**, **Recipients**, **Sender**, **Subject**

MSMQQuery Methods

LookupQueue

MSMQQueueInfos Methods

Next

Reset

MSMQQueueInfo Properties

- Authenticate
- BasePriority
- CreateTime
- FormatName
- IsTransactional
- IsWorldReadable
- Journal
- JournalQuota
- Label
- ModifyTime
- PathName
- PrivLevel
- QueueGuid
- Quota
- ServiceTypeGuid

MSMQQueueInfo Methods

- Create
- Delete
- Open
- Refresh
- Update

MSMQQueue Properties

- Access
- Handle
- IsOpen
- QueueInfo
- ShareMode

MSMQQueue Methods

- Close
- EnableNotification
- Peek
- PeekCurrent
- PeekNext
- Receive
- ReceiveCurrent
- Reset

MSMQEvent Events

- Arrived
- ArrivedError

MSMQMessage Properties

- Ack
- AdminQueueInfo
- AppSpecific
- ArrivedTime
- Authlevel
- Body
- BodyLength
- Class
- CorrelationId
- Delivery
- DestinationQueueInfo
- EncryptAlgorithm
- HashAlgorithm
- Id
- IsAuthenticated
- Journal
- Label
- MaxTimeToReachQueue
- MaxTimeToReceive
- Priority
- PrivLevel
- ResponseQueueInfo
- SenderCertificate
- SenderID
- SenderIDType
- SentTime
- SourceMachineGuid
- Trace

MSMQMessage Methods

- AttachCurrentSecurityContext
- Send

MSMQCoordinatedTransactionDispenser Methods

BeginTransaction

MSMQTransactionDispenser Methods

BeginTransaction

MSMQTransaction Properties

Transaction

MSMQTransaction Methods

- **Abort**
- **Commit**
- **Transaction**

MSMQApplication Methods

MachineIdOfMachineName

MSMQMailEMail Properties

- ContentType
- DestinationQueueLabels
- DeliveryReportData
- FormData
- NonDeliveryReportData
- Recipients
- RequestDeliveryReport
- RequestNonDeliveryReport
- Sender
- Subject
- SubmissionTime
- TextMessageData
- TnefData

MSMQMailEMail Methods

- ComposeBody
- ParseBody

MSMQMailFormData Properties

MSMQMailFormField Properties

Name
Value

MSMQMailFormFieldList Properties

- Count
- Item

MSMQMailFormFieldList Methods

- Add
- Remove

MSMQMailRecipient Properties

- Address
- Name
- RecipientType
- NonDeliveryReason
- DeliveryTime

MSMQMailRecipientList Properties

- Count
- Item

MSMQMailRecipientList Methods

- Add
- Remove

MSMQMailTextMessageData Properties

Text

MSMQMailTnefData Properties

Data

MSMQMailDeliveryReportData Properties

- DeliveredRecipients
- OriginalSubject
- OriginalSubmissionTime

MSMQMailNonDeliveryReportData Properties

A

acknowledgment message
Indicates whether the message reached the queue or was retrieved from the queue. The type of acknowledgment is generated by MSMQ, MSMQ connector applications, or computers outside the MSMQ system.

AddressSpecification
The address specification of a computer. It can be specified using two forms: either as the network address of the target machine (including the network protocol) or as any string that is supported by the underlying operating system to identify the target machine.

As a shortcut, the operating system can be used to indicate that the computer's native protocol should be used.

administration queue
A queue created and maintained by the application. It receives MSMQ-generated acknowledgment messages that indicate if a message reaches a queue, or if the message is retrieved from the queue.

application input queue
An application input queue is created by the MSMQ application that needs to read MSMQ mail formatted messages. When the application creates the queue, the queue's label must be set to the address of the MAPI application and the queue's type must be set to the following MAPI type identifier:

{5EADC0D0-7182-11CF-A8FF-0020AFB8FB50}

When a MAPI form is sent to an MSMQ application, the MSMQ MAPI Transport Provider locates the application's input queue by the queue's label and type. If the MAPI Transport provider cannot find the queue with the correct label and type, the form is rejected and a non-delivery report is generated.

application object
Object whose methods provide global functionality. Unlike most application objects, the MSMQ application object does not start a new instance of MSMQ.

audit
Queue operations can be audited by modifying the system access control list (SACL) of the queue's security descriptor. For a complete description of auditing, see the *Microsoft Message Queue Server Administrator's Guide*).

C

callback function
Used to asynchronously read the messages in a queue. It is an application-defined function that MSMQ calls when a message is available, a timeout occurs, or an error occurs.

certificate authority
Issues external certificates. The certificate authority accepts requests for certificates, confirms that the information provided in the request is accurate, then returns a certificate to the person requesting it.

The requester must provide their public key, and whatever additional information is required by the certificate authority.

CN
See connected network.

computer
Computers are created and maintained by the MSMQ administrator. All existing computers are defined in MQIS.

Their properties can be retrieved using **MQGetMachineProperties**.

connected network
A collection of computers where any two computers can communicate directly. For more information on CNs, see the *Microsoft Message Queue Server Administrator's Guide*.

connector application
Used by a MSMQ connector server to translate between MSMQ message properties and foreign message properties.

Connector applications may also perform security services such as authenticating messages and encrypting/decrypting messages.

connector queue
Queue used by an MSMQ connector server. Messages sent to foreign queues are temporarily stored in a connector queue before they are retrieved by the connector application.

MSMQ connector servers can have several pairs of connector queues. There is a transaction and non-transaction queue for each foreign CN connected to the server.

critical section object
A Win32 object that provides mutually-exclusive synchronization. A critical section object can only be used by one thread at a time.

For more information on critical section objects, see the Platform SDK and winbase.h.

D

dead letter queue
Used to store application-generated messages that cannot be delivered. There are two dead letter queues, one for transaction messages and the other for non-transaction messages.

DEADXACT
Indicates the dead letter queue for transaction messages is requested.

For more information see dead letter queue.

DEADLETTER
Indicates the dead letter queue for non-transaction messages is requested.

For more information see dead letter queue.

delivery report
E-mail that is sent to the originator of a previously sent e-mail (referenced as the original e-mail). It contains a list of the recipients who received the e-mail and when, plus the original e-mail's subject and submission time.

dependent client
MSMQ computers that cannot function without synchronous access to an MSMQ server (PEC, PSC, BSC, or MSMQ routing server).

MSMQ dependent clients require synchronous access to the supporting MSMQ server to perform all standard MSMQ functions, such as creating queues, sending messages, and receiving messages.

digital signature
Used to verify the source of a message and that the message was not modified.

direct format name
Format used to open a queue that is not in your enterprise, or when you want to make sure MSMQ sends messages to the queue in one step.

Direct format names include the address of the computer where the queue is located followed by the local name of the queue (the name specified in the queue's pathname when the queue was created).

E

E-mail message
There are two types of e-mail messages: forms and text messages. Text messages use a single text body to pass information while e-mail forms use one or more fields.

express
Delivery mode that provides faster delivery. The message stays in memory (RAM) until it can be delivered and is not recovered if the computer is rebooted.

external certificate
Used when the receiving application needs information in the certificate to verify who sent a message.

External certificates contain information about the certificate authority, the certified user, the validity period of the certificate, the public key of the certified user, and the certificate authority's signature.

For more information see internal certificate.

external transaction
Transaction called when the transaction must work with more than one resource manager (more than simply sending or retrieving an MSMQ message). In this case, the application must ask MS DTC for a transaction object and reference that object each time it sends a message, retrieves a message, or executes an action of another resource manager.

F

foreign CN
A connected network (CN) that contains computers that do not run MSMQ (foreign computers) and at least one MSMQ connector server.

For information on creating, renaming, or removing a foreign CN, see the *MSMQ Administrator's Guide*.

foreign computer
A computer that does not run MSMQ, but can exchange messages with MSMQ through an MSMQ connector application.

For information on creating, renaming, or removing a foreign computer, see the *MSMQ Administrator's Guide*.

foreign queue
A queue that resides on a computer that does not run MSMQ (a foreign computer).

ForeignCNGUID
Identifier of the foreign CN where the MSMQ connector is located. The CN's identifier is generated by MSMQ when the CN is created.

For information on CNs, see the *Microsoft Message Queue Server Administrator's Guide*.

format name
Used to specify a queue when making calls to several API functions.

The queue's format name is not an MSMQ queue property. It is a unique name for the queue generated by MSMQ when it is created. The format name can also be generated later by the application. MSMQ never stores the format name of a queue for later reference.

H

hive
A discrete body of registry keys, subkeys, and values that is rooted at the top of the registry.

I

independent client

MSMQ independent clients can create and modify queues as well as send and receive messages just as MSMQ servers can. MSMQ independent clients can create queues and store messages on the local computer without synchronous access to an MSMQ server. The primary difference between MSMQ independent clients and MSMQ servers is that independent clients do not have the intermediate store-and-forward capability of MSMQ servers, nor do they store information from the distributed MSMQ database.

In addition to the basic MSMQ files, you can install the Microsoft Message Queue Server SDK on MSMQ independent clients.

You can also install the MSMQ Explorer on MSMQ independent clients running under Windows NT® Workstation or Server. You can use the MSMQ Explorer to administer your MSMQ enterprise remotely from computers running Windows NT Workstation.

internal certificate

Used when the receiving application only needs to verify that the sender identifier attached to a message is valid.

An internal certificate contains a public key written in the form of an X.509 certificate. Internal certificates have no additional sender information that can be used for authentication.

internal transaction

Transaction called where MSMQ is the only resource manager. MSMQ internal transactions cannot be passed to another resource manager, unlike MS DTC external transactions. It is the additional RAM that is needed to coordinate between several resource managers that makes MSMQ internal transaction a better choice than MS DTC external transactions.

J

journal queue
See: machine journal, queue journal.

JOURNAL
Indicates that the journal queue is requested.

For more information see machine journal, or queue journal.

M

machine
See computer.

MachineName
The name of the computer where the queue's messages will be stored. Machine names are not case sensitive, so "mymachine" and "MyMachine" are treated the same way.

To indicate the local computer, you can substitute the string "." for the name of the local machine. For private queues the machine name must be the name of the local machine.

MachineGUID
Computer identifier generated by MSMQ when the computer is added to its CN.

For more information see connected network.

machine journal
Used to store copies of application-generated messages.

For more information see queue journal

mail message
An MSMQ message whose body is formatted in MSMQ mail format. The body of a mail message can be composed or parsed by any MSMQ application.

The MSMQ Exchange Connector and MAPI Transport Provider translate between e-mail and MSMQ mail messages.

message
MSMQ messages are defined by their properties. Included in these properties is the message body, which contains the bulk of the information passed between applications.

message identifier
Used by applications to identify a message. It is also used by MSMQ to indicate the original application message associated with an acknowledgment or report message.

message queue
An application-generated queue that contains application-generated messages. Applications can send messages to these queues or read their messages. They can be *public queues* registered in the MSMQ information store or private queues that are registered on individual computers.

MIME
Multipurpose Internet Mail Extensions (MIME) is a standard that is used to encode Internet mail messages, and is described by several RFCs; the most relevant RFCs are RFC-822, 1521 (MIME), and RFC-1867 (form data).

MQIS
See: MSMQ Information Store.

MS DTC external transaction
External transaction where transaction object is provided by Microsoft® Distributed Transaction Coordinator (MS DTC). Used when more then one resource manager is required.

MSMQ Connector Server

The MSMQ connector server allows MSMQ-based applications communicate with computers that use other messaging systems (foreign computers). MSMQ connector servers use internal connector queues and a connector application to pass messages between the MSMQ and foreign enterprises.

The Level 8 Systems MSMQ message queuing product is an example of an MSMQ connector server.

MSMQ information store

MSMQ information store (MQIS). A Microsoft® SQL Server version 6.5 replicated database that contains information of about your MSMQ enterprise. It includes information about items such as public queues, computers, MSMQ servers, and CNs.

MSMQ internal transaction

Transaction provided by MSMQ. Internal transactions cannot be passed to other resource managers.

MSMQ mail format

The MSMQ mail format is used by applications that send messages to e-mail based applications. The MSMQ mail format is a subset of the standard Multipurpose Internet Mail Extensions (MIME) format.

MTS transaction

Transaction that uses the Microsoft® Transaction Server (MTS) environment. MSMQ implicitly uses the current MTS transaction if one is available.

N

non-delivery report
E-mail that is sent to the originator of a previously sent e-mail (referenced as the original e-mail). It contains the recipients who did not receive the original e-mail, the reason e-mail was not delivered, and the original e-mail content.

non-transaction message
Any message that is sent to a non-transaction queue. Typically, any message that is not part of a transaction.

For more information see non-transaction queue.

non-transaction queue
A queue that only contains non-transaction messages.

For more information see non-transaction message.

P

pathname
Used when creating the queue. It indicates where to store the queue's messages, where to register the queue, and provides a name for the queue.

peek
To look at a message in a queue without removing it from the queue.

For more information see read or retrieve.

private message
Message whose body is encrypted. Applications can set the privacy-level for each message it sends and the encryption algorithm used to encrypt the message.

private format name
Format used to specify queues not registered in MSMQ information store (MQIS). The private format name of the queue includes the string "Private=" followed by the MachineGUID (machine identifier) of the computer where the queue is located and a hexadecimal number that identifies the queue.

The following is the general format of a private format name:

"PRIVATE=MachineGUID\QueueNumber"

private signing key
Signature key used to digitally sign a message. The private signing key is part of the signature key pair and should always be kept private.

For information on the other signature key, see public signing key.

private queue
A queue registered on the local computer (not in MQIS) that typically cannot be located by other applications. Private queues have the advantage of no MQIS overhead (faster to create, no latency, and no replication), and they can be created and deleted when MQIS is not working.

For more information see public queue.

PRIVATE$
Indicates the queue is private and is registered on the local computer. Its absence indicates a public queue that is registered in MQIS.

public format name
Format used to specify a queue registered in MQIS. A queue's public format name contains the string "Public=" followed by the queue identifier generated by MSMQ when the queue was created.

The following is the general format for public format names:

"PUBLIC=QueueQUID"

public signing key
Signature key used to validate the digital signature of a message. The public signing key is part of the signature key pair, and is sent with the message.

For information on the other signature key, see private signing key.

public queue
A queue registered in MQIS that can be located by any MSMQ application. Public queues are persistent and their registration information can be backed up on the MSMQ enterprise, making them good for long-term use.

For more information see private queue.

Q

queue
Objects that hold messages passed among applications, or messages passed between MSMQ and applications. Applications can send messages to queues and read messages from queues.

For more information see message queues, administration queues, dead letter queues, journal queues, response queues, report queues, transaction queues, and foreign queues.

QueueGUID
Queue identifier returned by MSMQ when the queue is created.

QueueName
Application-defined name used to identify queue. This name is specified when the queue is created.

QueueNumber
Eight-digit, hexadecimal number that identifies the private queue. It is generated by MSMQ when the queue is created on the local computer.

To find the queue number of a private queue, use Windows Explorer to locate the queue in the ..\MSMQ\Storage\Lqs folder.

Queue Manager
An MSMQ service responsible for delivering, receiving, authenticating, and routing messages, as well as maintaining the MSMQ information store.

queue journal
Used to store copies of application-generated messages after they are retrieved from the queue.

For more information see machine journal.

R

read
Peeking at or retrieving a message in the queue.

For more information see peek or retrieve.

recoverable
Delivery mode that guarantees message delivery even in the case of a computer crash. In this mode the message is forwarded to the next hop or stored in a local backup file every hop along its route until it is delivered.

report message
Generated each time a message passes through an MSMQ routing server.

For more information see report queue.

report queue
A queue used to track the progress of your messages as they move through your enterprise. Report queues receive MSMQ-generated report messages. Applications can only read the messages in a report queue.

For more information see report message.

response message
Application-generated message that is returned to the sending application's response queue.

For more information see response queue.

response queue
A queue used to return application-generated response messages from the application reading the messages in a queue.

For more information see Response Queues.

retrieve
To read a message in a queue and remove it from the queue.

For more information see peek or read.

rich-text recipient
Users who selected the check box labeled "Send to this recipient in Microsoft Exchange rich text format" in their Exchange/MAPI address.

S

security context information

Information extracted from an external certificate or internal certificate. Security context information is used when the same certificate is used several times. Also, using the correct security context information is very important when impersonating a user.

For more information, see external certificate. and internal certificates.

security descriptor

An opaque structure that consists of a SECURITY_DESCRIPTOR structure and its associated security information.

Security information can include security identifiers (SID), a discretionary access-control list (DACL), and a system access-control list (SACL).

For a complete description of security descriptors, see the Platform SDK.

server input queue

The Exchange server input queue is created by the MSMQ Exchange Connector when the connector starts the first time. When an MSMQ message reaches this queue, it is picked up by the MSMQ Exchange Connector, translated into an e-mail message, then sent on to the Exchange Server where it is distributed to the appropriate Exchange user.

The queue's label is set by the Exchange Connector's Setup program. In addition, the Exchange Connector sets the queue's type to the MAPI type identifier shown in the following example.

MAPI type identifier:

```
{5EADC0D0-7182-11CF-A8FF-0020AFB8FB50}
```

SID

A unique value of variable length used to identify a user or group. The SID is assigned when the user logs on and becomes part of the access token for any process started by the user.

The SID contains a 48-bit identifier authority value, a revision level, and any number of sub-authority values. For a complete description of SIDs, see the Platform SDK.

site controller server

MSMQ server installed using MSMQ Explorer. See *Microsoft Message Queue Server Administrator's Guide* for details on installing a site controller.

T

TNEF
Transport-Neutral Encapsulation format. A MAPI-defined format that encapsulates MAPI message properties inside a single binary stream. In order to preserve MAPI properties, a sending application can encode a MAPI message into a TNEF stream and send the stream to a receiving application. The receiving application can use MAPI to decode the stream and reconstruct the original MAPI message.

TNEF message
Message formatted in Transport-Neutral Encapsulation format (TNEF).

See: TNEF.

transaction message
Message sent as part of a transaction. Transaction messages must be sent to transaction queues

For more information see transaction queue.

transaction queue
A queue that contains transaction messages. Transaction queues can only contain transaction messages, which are messages sent within a transaction.

For more information see transaction message.

transaction status queue
Contains the read receipt acknowledgments returned by connector applications. It is specified by setting the message's PROPID_M_XACT_STATUS_QUEUE property. The transaction status queue must be a transaction queue.

U

user input queue

Queue created when the MSMQ MAPI Transport Provider is started the first time. The transport provider creates a user input queue for each MAPI user. Each queue is created with an enterprise scope, its type property set to the MAPI type identifier shown in the following example, and its label set to a MAPI user's login name.

MAPI type identifier:

```
{5EADC0D0-7182-11CF-A8FF-0020AFB8FB50}
```

An MSMQ mail message that reaches any of these queues is picked up by the MSMQ MAPI Transport Provider, translated into an e-mail message, and then sent on to the inbox of the MAPI application.

X

XA transaction
Transaction that uses an XA-compliant transaction manager. MSMQ implicitly calls the current XA transaction.

XACTONLY
Indicates that the transactional connector queue is requested.

Release Notes

This appendix contains information about the differences between Microsoft® Message Queue Server 1.0 (MSMQ) and previous preliminary releases, as well as late-breaking information that was not incorporated into the Guide or Reference sections of the MSMQ Programmer's Reference.

Changes in ActiveX Components

When developing applications, you must use new names when specifying some of the parameters and properties of the ActiveX components supplied by the MSMQ and MSMQ Mail SDKs.

These new ActiveX objects are not compatible with ActiveX objects that shipped in preliminary releases of MSMQ. All ActiveX binaries must be recompiled against the new MSMQ 1.0 components, and you may also need to edit source code due to object model changes.

In addition to some entirely new names, the naming convention used with the MSMQ 1.0 ActiveX component implementation has removed Hungarian notation prefixes from the old names of all methods, properties, and parameters.

If your applications use the MSMQ ActiveX components included with any of the MSMQ beta releases, you must edit the appropriate ActiveX methods, properties, and parameter names before running your application on MSMQ 1.0. This renaming requires that you edit many, if not all, of your existing programs.

MSMQQuery.LookupQueue

The following name changes were made to the parameters of the **LookupQueue** method of **MSMQQuery**. Former names are in comments.

```
LookupQueue(
            QueueGuid          'strGuidQueue
            ServiceTypeGuid    'strGuidServiceType
            CreateTime         'dateCreateTime
            ModifyTime         'dateModifyTime
            RelServiceType     'relServiceType
            RelLabel           'relLabel
            RelCreateTime      'relCreateTime
            RelModifyTime      'relModifyTime
            )
```

MSMQMessage (Properties)

The following name changes were made to the properties of the **MSMQMessage** object.

MSMQMessage.Class	'lClass
MSMQMessage.PrivLevel	'lPrivLevel
MSMQMessage.AuthLevel	'lAuthLevel
MSMQMessage.IsAuthenticated	'isAuthenticated
MSMQMessage.Delivery	'lDelivery
MSMQMessage.Trace	'lTrace
MSMQMessage.Priority	'lPriority
MSMQMessage.Journal	'lJournal
MSMQMessage.ResponseQueueInfo	'queueinfoResponse
MSMQMessage.AppSpecific	'lAppSpecific
MSMQMessage.SourceMachineGuid	'guidSrcMachine
MSMQMessage.BodyLength	'lenBody
MSMQMessage.Body	'body
MSMQMessage.AdminQueueInfo	'queueinfoAdmin
MSMQMessage.Id	'id
MSMQMessage.CorrelationId	'idCorrelation
MSMQMessage.Ack	'lAck
MSMQMessage.Label	'strLabel
MSMQMessage.MaxTimeToReachQueue	'lMaxTimeToReachQueue
MSMQMessage.MaxTimeToReceive	'lMaxTimeToReceive
MSMQMessage.HashAlgorithm	'lHashAlg
MSMQMessage.EncryptAlgorithm	'lEncryptAlg
MSMQMessage.SentTime	'dateSentTime
MSMQMessage.ArrivedTime	'dateArrivedTime
MSMQMessage.DestinationQueueInfo	'queueinfoDest
MSMQMessage.SenderCertificate	'binSenderCert
MSMQMessage.SenderId	'binSenderId
MSMQMessage.SenderIdType	'lSenderIdType

The lSecurityContext property is no longer available. See MSMQMessage.AttachCurrentSecurityContext.

MSMQMessage.Send

The following name changes were made to the parameters of the **Send** method of **MSMQMessage**. Former names are in comments.

```
Send(
        DestinationQueue        'pqDest
        Transaction             'lTransaction
       )
```

MSMQMessage.AttachCurrentSecurityContext

The **AttachCurrentSecurityContext** method retrieves the security context information from the security certificate specified by **SenderCertificate** and associates it with the current object.

MSMQQueue (Properties)

The following name changes were made to the properties of the **MSMQQueue** object.

```
MSMQMessage.Access        'lAccess
MSMQMessage.ShareMode     'lShareMode
MSMQMessage.QueueInfo     'queueinfo
MSMQMessage.Handle        'lHandle
MSMQMessage.IsOpen        'isOpen
```

MSMQQueue.Receive

The following name changes were made to the parameters of the **Receive** method of **MSMQQueue**. Former names are in comments.

```
Receive(
        Transaction             'lTransaction
        WantDestinationQueue    'wantDestQueue
        WantBody                'wantBody
        ReceiveTimeout          'lReceiveTimeout
       )
```

MSMQQueue.ReceiveCurrent

The following name changes were made to the parameters of the **ReceiveCurrent** method of **MSMQQueue**. Former names are in comments.

```
ReceiveCurrent(
        Transaction          'lTransaction
        WantDestinationQueue 'wantDestQueue
        WantBody             'wantBody
        ReceiveTimeout       'lReceiveTimeout
        )
```

MSMQQueue.Peek

The following name changes were made to the parameters of the **Peek** method of **MSMQQueue**. Former names are in comments.

```
Peek(
        WantDestinationQueue   'wantDestQueue
        WantBody               'wantBody
        ReceiveTimeout         'lReceiveTimeout
       )
```

MSMQQueue.PeekCurrent

The following name changes were made to the parameters of the **PeekCurrent** method of **MSMQQueue**. Former names are in comments.

```
PeekCurrent(
        WantDestinationQueue   'wantDestQueue
        WantBody               'wantBody
        ReceiveTimeout         'lReceiveTimeout
       )
```

MSMQQueue.PeekNext

The following name changes were made to the parameters of the **PeekNext** method of **MSMQQueue**. Old names are in comments.

```
PeekNext(
        WantDestinationQueue   'wantDestQueue
        WantBody               'wantBody
        ReceiveTimeout         'lReceiveTimeout
       )
```

MSMQQueue.EnableNotification

The following name changes were made to the parameters of the **EnableNotification** method of **MSMQQueue**. Former names are in comments.

```
EnableNotification(
                    Event           'pqEvent
                    Cursor          'lCursor
                    ReceiveTimeout  'lReceiveTimeout
                   )
```

MSMQEvent.Arrived

The following name changes were made to the parameters of the **Arrived** method of the **MSMQEvent** object. Former names are in comments.

```
Arrived(
            Queue               'pdispQueue
            Cursor              'lCursor
          )
```

MSMQEvent.ArrivedError

The following name changes were made to the parameters of the **ArrivedError** method of the **MSMQEvent** object. Former names are in comments.

```
ArrivedError(
              Queue          'pdispQueue
              ErrorCode      'lErrorCode
              Cursor         'lCursor
             )
```

MSMQQueueInfo (Properties)

The following name changes were made to the properties of the **MSMQQueueInfo** object.

```
MSMQQueueInfo.QueueGuid              'guidQueue
MSMQQueueInfo.ServiceTypeGuid        'guidServiceType
MSMQQueueInfo.Label                  'strLabel
MSMQQueueInfo.PathName               'strPathName
MSMQQueueInfo.FormatName             'strFormatName
MSMQQueueInfo.IsTransactional        'isTransactional
MSMQQueueInfo.Journal                'lJournal
MSMQQueueInfo.Quota                  'lQuota
MSMQQueueInfo.BasePriority           'lBasePriority
MSMQQueueInfo.CreateTime             'dateCreateTime
MSMQQueueInfo.ModifyTime             'dateModifyTime
MSMQQueueInfo.PrivLevel              'lPrivLevel
MSMQQueueInfo.Authenticate           'lAuthenticate
MSMQQueueInfo.JournalQuota           'lJournalQuota
MSMQQueueInfo.IsWorldReadable        'IsWorldReadable (no change)
MSMQQueueInfo.QueueGuid              'guidQueue
```

MSMQQueueInfo.Create

The following name changes were made to the parameters of the **Create** method of the **MSMQQueueInfo** object. Former names are in comments.

```
Create(
        IsTransactional          'isTransactional
        IsWorldReadable            'IsWorldReadable (no change)
       )
```

MSMQQueueInfo.Open

The following name changes were made to the parameters of the **Open** method of the **MSMQQueueInfo** object. Old names are in comments.

```
Open(
        Access              'lAccess
        ShareMode           'lShareMode
       )
```

MSMQTransaction (Properties)

The following name changes were made to the properties of the **MSMQTransaction** object.

MSMQTransaction.Transaction	'ITransaction

MSMQMailEMail (Properties)

The following name changes were made to the properties of the **MSMQMailEMail** object.

```
MSMQMailEMail.FormData                  'formdata
MSMQMailEMail.SubmissionTime            'dateSent
MSMQMailEMail.ContentType               'lType
MSMQMailEMail.DestinationQueueLabels    'labelsDestination
MSMQMailEMail.Recipients                'recipients
MSMQMailEMail.Sender                    'recipientSender
MSMQMailEMail.TextMessageData           'txtmessagedata
MSMQMailEMail.DeliveryReportData        'New property
MSMQMailEMail.NonDeliveryReportData     'New property
MSMQMailEMail.TnefData                  'New property
MSMQMailEMail.RequestDeliveryReport     'New property
MSMQMailEMail.RequestNonDeliveryReport  'New property
```

MSMQMailEMail.ParseBody

The following name changes were made to the parameters of the **ParseBody** method of the **MSMQMailEmail** object. Former names are in comments.

```
ParseBody(
            Body        'varBody
          )
```

MSMQMailFormData (Properties)

The following name changes were made to the properties of the **MSMQMailFormData** object.

```
MSMQMailFormData.FormFields        'formfields
MSMQMailFormData.Name              'strName
```

MSMQMailFormField (Properties)

The following name changes were made to the properties of the **MSMQMailFormField** object.

MSMQMailFormField.Name 'strName
MSMQMailFormField.Value 'varValue

MSMQMailFormFieldList.Add

The following name changes were made to the parameters of the **Add** method of the **MSMQMailFormFieldList** object. Former names are in comments.

```
Add(
      Name       'strName
      Value      'varValue
      Key        'sKey
     )
```

MSMQMailFormFieldList.Item

The following name changes were made to the parameters of the **Item**method of the **MSMQMailFormFieldList** object. Former names are in comments.

```
Item(
      IndexKey                'vntIndexKey
     )
```

MSMQMailFormFieldList.Remove

The following name changes were made to the parameters of the **Remove** method of the **MSMQMailFormFieldList** object. Former names are in comments.

```
Remove(
        IndexKey      'vntIndexKey
       )
```

MSMQMailRecipient (Properties)

The following name changes were made to the properties of the **MSMQMailRecipient** object.

```
MSMQMailRecipient.Name                'strName
MSMQMailRecipient.Address             'strAddress
MSMQMailRecipient.RecipientType       'lType
MSMQMailRecipient.NonDeliveryReason   'New property
MSMQMailRecipient.DeliveryTime        'New propery
```

MSMQMailRecipientList.Add

The following name changes were made to the parameters of the **Add** method of the **MSMQMailRecipientList** object. Former names are in comments.

```
Add(
      Name                 'strName
      Address              'strAddress
      RecipientType        'lType
      Key                  'sKey
     )
```

MSMQMailRecipientList.Item

The following name changes were made to the parameters of the **Item**method of the **MSMQMailRecipientList** object. Former names are in comments.

```
Item(
      IndexKey          'vntIndexKey
     )
```

MSMQMailTextMessageData (Properties)

The following name changes were made to the properties of the **MSMQMailTextMessageData** object.

MSMQMailTextMessageData.Text	'strText

Upgrading from Beta2

The following modifications were made to the ActiveX object model between Beta2 and Beta2E preliminary releases. These changes are relevant to Beta2 clients who are upgrading directly from Beta2 to MSMQ 1.0.

MSMQQueue

New PeekCurrent method: This new property reads the message at the current cursor position.

lReceiveTimeout is no longer a property: This property is now a parameter for Peek, PeekNext, PeekCurrent, Receive, ReceiveCurrent, and EnableNotification.

New ReceiveCurrent method: This replaces ReceiveNext, which has been removed.

ReceiveCurrent, PeekCurrent, and PeekNext all use the implied cursor. Peek and Receive do not.

EnableNotification has new MQMSG enumeration parameters: MQMSG_FIRST, MQMSG_CURRENT, or MQMSG_NEXT.

- MQMSG_FIRST: This user-defined event handler is invoked whenever there is a message in queue or a timeout occurs. The cursor is not used.
- MQMSG_CURRENT: This user-defined event handler is invoked whenever there is a message at the current cursor position or a timeout occurs.
- MQMSG_NEXT: This user-defined event handler is invoked whenever there is a message at the (new) cursor position or a timeout occurs. The cursor is advanced.

DisableNotification has been removed.

Synchronous read methods Peek, PeekNext, PeekCurrent, Receive, and ReceiveCurrent no longer use the MQ_ERROR_IO_TIMEOUT error. Instead, a NULL message object is returned if a time-out occurs. Use the idiom: not msg is nothing to test for end of queue.

Note Asynchronous read still can use MQ_ERROR_IO_TIMEOUT error (the MSMQEvent.ArrivedError event handler is invoked).

MSMQQuery

LookupQueue takes additional optional restriction parameters:
CreateTime (RelCreateTime) and ModifyTime(RelModifyTime).

Note The default REL_EQ relational operator is not very useful for these two parameters

MSMQMessage

- The body property can now be any intrinsic Variant, including date, currency, numbers (in addition to strings, byte arrays and persistent objects).
- New properties SentTime and ArrivedTime are Variant dates instead of lSentTime and lArrivedTime, which have been removed.
- New AttachCurrentSecurityContext method replaces lSecurityContext property.

MSMQQueueInfo

New properties: CreateTime and ModifyTime are Variant dates. Use these properties instead of lCreateTime, strCreateTime, lModifyTime and strModifyTime, which have been removed.

New Create optional parameter: IsWorldReadable. The default is False. This means the default MSMQ security is used where all users can send to the queue, but only the owner can read from the queue. When True, any user can read from the queue.

New Boolean read-only property: IsWorldReadable. True indicates that everyone can read from the queue. False indicates that only the owner can read messages from the queue.

Note The IsWorldReadable property is not cached, because other applications can dynamically change the state of the queue

MSMQCoordinatedTransactionDispenser

This is simply a renaming of the old MSMQTransactionDispenser. This object dispenses MS DTC transactions.

MSMQTransactionDispenser

A new object that dispenses MSMQ-only transactions.

MSMQTransaction

A new read-only property: Transaction. It exposes the actual ITransaction* magic cookie internally used by the BeginTransaction method from either of the dispenser objects. This property can be used to pass to other components such as SQL Server.

Transactions with Microsoft Transaction Server (MTS)

When sending messages to a transactional queue, the MSMQMessage object's Send method takes an optional MSMQTransaction object parameter. The parameter can be one of the following:

An actual MSMQTransaction object, as obtained from BeginTransaction.

If no parameter is supplied, then any current MTS transactions will be used.

MQ_NO_TRANSACTION: This is an error if the target queue is transactional. Otherwise, it overrides the default MTS transaction.

MQ_XA_TRANSACTION: Uses current default XA transaction, if any.

MQ_MTS_TRANSACTION: Explicitly uses the current MTS transaction.

MQ_SINGLE_MESSAGE: Uses internal MSMQ transaction semantics for the message.

MSMQApplication Object

New object: "application object." A single instance of this class is automatically created by MSMQ. Its methods are all in the global name space, that is, they do not require object qualification.

Its only current method, MachineIdOfMachineName, is used to construct the format name of a computer journal queue so that it can be opened.

MSMQMailTnefData Object

New object that defines a TNEF message. The **MSMQMailTnefData** object has a single **Data** property and no methods.

MSMQMailDeliveryReportData Object

New object that defines an application-generated delivery report: The **MSMQMailDeliveryReportData** object the following properties.

MSMQMailDeliveryReportData.DeliveredRecipients
MSMQMailDeliveryReportData.OriginalSubject
MSMQMailDeliveryReportData.OriginalSubmissionTime

MSMQMailNonDeliveryReoprtData Object

New object that defines a non-delivery report: The **MSMQMailNonDeliveryReportData** object the following properties.

MSMQMailDeliveryReportData.NonDeliveredRecipients
MSMQMailDeliveryReportData.OriginalEMail