Application Messaging with SAS® 9.4, Second Edition
### Chapter 9 / Message Queue Polling

- Overview of Message Queue Polling .................................................. 259
- Define a Queue Manager for WebSphere MQ .................................... 260
- Define a Queue Manager for JMS ..................................................... 261
- Define a Message Queue Polling Server .......................................... 262
- Add the Polling Server to the Object Spawner Definition.................... 264
- Configure Your Java Environment for the Object Spawner and the JMS Access Method ........................................ 265
- Configure Your Programs for Message Queue Polling ....................... 268

### Chapter 10 / The JMS File Access Method

- Dictionary .................................................................................. 273

### Chapter 11 / The ACTIVEMQ File Access Method

- Dictionary .................................................................................. 287
What’s New in Application Messaging with SAS 9.4

Overview

Application Messaging has the following changes:

- new ActiveMQ file access method
- new optional arguments for the CALL CLOSEQUEUE routine

New ACTIVEMQ File Access Method

The new ACTIVEMQ file access method enables you to access an ActiveMQ broker by using the FILENAME, FILE, and INFILE statements.

New Optional Arguments for the CALL CLOSEQUEUE Routine

In SAS 9.4M2, the following two optional arguments have been added for the CALL CLOSEQUEUE routine:

- **RECOGNIZE_MSMQ_DELETE_ERRORS**
  recognizes MSMQ queue deletion errors by checking the return code for the CLOSEQUEUE CALL routine.

- **LOG_MSMQ_DELETE_ERRORS**
  logs SMQ queue deletion errors to the App.Program logger.
Application Messaging Overview

Application messaging architectures provide a platform that supports interoperability among loosely coupled applications over a message passing bus. When the targeted scope of interoperability is broad (for example, spanning multiple application systems and organizational boundaries), application messaging architectures might be required. This is because the likelihood of conformance in the software implementation base (for example, the selected distributed object standard) across the set of participating applications is diminished. In addition, the set of participating applications can exhibit asynchronous, disconnected operation. These applications execute with no direct point-to-point communication session. However, they require guaranteed fulfillment of requests for service or event delivery.

This degree of operational heterogeneity introduces several requirements that are reflected in the application messaging infrastructure. Heterogeneity in the implementation base of the various applications (including perhaps, retrofitted legacy applications) suggests a need for a reasonably nonintrusive integration mechanism. The semantics of application messaging satisfy this need, generally expressing open, close, send, and receive functionality with flexible application-defined message structures. Heterogeneity with respect to the asynchronous, disconnected execution and notification modes of end-point participants introduces requirements for service qualities that include routing, assured just-once delivery, and retained sequencing. The architecture that has emerged within commercial application messaging products to express these quality-of-service properties is store-and-forward queuing.

In a store-and-forward model, messages are sent to named queues, which are in turn hosted at specific destination network addresses. The navigation of messages from their origin occurs through a transmission network that ensures the integrity of message delivery to the destination queue and presentation to the recipient process.
Ever more frequently, the simple design pattern of two identifiable applications that interoperate over a message passing bus is inconsistent with the realities of an event-driven enterprise. Interdependencies across multiple applications with respect to events that occur within an enterprise combined with an ever-changing topology of event supplier and consumer applications are often present. Decision-makers require information pertinent to their domain of responsibility regardless of the reporting applications. Automated business processes require modification in rapid response to changing operational conditions. The ability to satisfy these requirements in a timely manner, and thereby reduce the latencies too common in information interchange, is critical to efficient and effective enterprise performance.

To support such dynamism, extended application messaging infrastructure facilities in the form of message brokers are emerging. Message brokers are being effectively positioned as enterprise application integration and event-management focal points, which function as hub processes that manage the information flow throughout an enterprise. Operationally, message brokers provide rules-based message routing and distribution as well as message transformation and augmentation capabilities that enable the removal of this aspect of implementation logic from participating applications.

A SAS JMS (Java Message Service) file access method enables SAS programs to interface with any Message-Oriented Middleware (MOM) that includes a JMS provider. Many MOM vendors include JMS providers, including some open-source offerings such as Apache ActiveMQ and RabbitMQ.

The SAS ACTIVEMQ file access method enables SAS programs to interface directly with ActiveMQ message brokers through the HTTP protocol.

In addition to the JMS and ACTIVEMQ file access methods, SAS Integration Technologies provides interfaces to three principal commercial messaging platforms, IBM WebSphere MQ (previously named MQSeries), Microsoft MSMQ, and TIBCO Software Rendezvous (including the Certified Message Delivery
Support for these platforms enables SAS software's information delivery capabilities to be leveraged within various enterprise solution scenarios, including application integration, asynchronous and mobile synchronization, and event notification.

Support for client environments is broad. IBM provides WebSphere MQ on a vast array of operating system platforms with programming language support including C/C++, Java, JMS, and Cobol as well as ActiveX control support that enables Visual Basic participation. Likewise, Microsoft provides full language support for MSMQ.

### Supported Platforms for the SAS Messaging Interfaces

<table>
<thead>
<tr>
<th>SAS Messaging Interface</th>
<th>Third-Party Software</th>
<th>Supported Operating Environment</th>
</tr>
</thead>
<tbody>
<tr>
<td>WebSphere MQ</td>
<td>WebSphere MQ version 7.01 and later</td>
<td>Windows: X, UNIX: X, z/OS: (server only)</td>
</tr>
<tr>
<td>Microsoft Message Queuing Services (MSMQ)</td>
<td></td>
<td>Windows: X</td>
</tr>
<tr>
<td>JMS file access method</td>
<td>Any message-oriented middleware JMS provider</td>
<td>Windows: X</td>
</tr>
<tr>
<td>ACTIVEMQ file access method</td>
<td>ActiveMQ broker release 5.7.0 or later</td>
<td>Windows: X, UNIX: X</td>
</tr>
</tbody>
</table>
Configuring WebSphere MQ with the WebSphere MQ Explorer

Configure a Queue Manager

Before you use the WebSphere MQ applications, you must create a queue manager. The queue manager is a system program that is responsible for maintaining the queues and ensuring that the messages in the queues reach their destination. It also performs other functions that are associated with message queuing.

A queue is a named destination that applications use to send and receive messages. A queue name must be unique within a queue manager. Special queue types can be defined, such as transmission queues and dead letter queues.

- A transmission queue is a queue that holds messages that will eventually be sent to a remote queue when a communication channel becomes available. Unless otherwise specified, these messages are transmitted through the default transmission queue.
A dead letter queue is a local queue where messages that cannot be delivered are sent, either by the queue manager or an application. Some method should be in place in production environments to monitor and process messages in this queue.

To configure a queue manager, perform the following steps:

1. From the WebSphere MQ Explorer window, expand the WebSphere MQ node, and then right-click Queue Managers. Select New ➔ Queue Manager from the pop-up menu.

2. Enter the name for your queue manager. The examples in this section use the name MYQMGR. Fill in names for the default transmission queue and dead letter queue. Select Make this the default queue manager.

   Note: All names in WebSphere MQ are case sensitive.

   Click Next to continue.

3. Click Next to accept the default values for the logging options.

4. Verify that Start Queue Manager is selected. Click Next to continue.

5. Make sure that Create listener configured for TCP/IP is selected, and enter 1414 for the port number. This is the default port number for WebSphere MQ. Check with your system administrator to verify that this is the correct port to use. Click Finish to create your queue manager. It might take a minute to create and start the queue manager.

Define Queues

Create one or more local queues for exchanging messages on your queue manager. These are the queues that SAS applications will use to exchange messages with other applications.

To define a queue, perform the following steps:

1. In the WebSphere MQ Explorer, locate your queue manager and expand the menu. Right-click Queues, and then select New ➔ Local Queue from the pop-up menu. The Create Local Queue window appears.

2. In the Queue Name field, enter the name of the local queue that you want to create. This queue name is specified in any application programs that use WebSphere MQ. You might also want to change the Default Persistence value from Not Persistent to Persistent. Setting a value of Persistent enables messages to remain in the queue even if the queue manager is shut down. Click each tab to see the types of values that can be defined.

3. (Optional) If you use high-volume messaging applications like scoring, then select the Extended tab and increase the value of Maximum Queue Depth to 100,000 or more. The value of Maximum Queue Depth represents the maximum number of messages that a queue can hold.
Click **OK** to create the queue. Repeat the process for any additional local queues that you want to create.

You should also create the dead letter queue that is specified in the queue manager definition. If you will be exchanging messages with queues on other queue managers, then create the default transmission queue. For information about configuring channels and transmission queues, see the IBM WebSphere MQ documentation.

At this point, WebSphere MQ has enough information for you to run applications that use message queuing locally within your machine through a single queue manager.

---

### Configuring WebSphere MQ Client Access

**Overview of Configuring Client Access**

IBM provides a lighter client version of WebSphere MQ that can be installed and used separately from the full WebSphere MQ Base or server installation. The client can be installed on the same machine as the server or on a separate machine. The client does not have its own queue manager and must communicate over the network or within a machine to a queue manager that is defined elsewhere.

To configure client access, perform the following steps:

1. “Define a Server Connection Channel” on page 7
2. “Install the WebSphere MQ Client” on page 7
3. “Define the Queue Manager Connection on the Client Machine” on page 8

### Define a Server Connection Channel

You must define a server connection channel on the queue manager that will provide support to the client. A channel is a definition that enables intercommunication between queue managers, or between clients and queue managers.

To define a server connection channel, perform the following steps:

1. In WebSphere MQ Explorer, select a queue manager and then select Advanced.

2. Right-click **Channels**, and then select **NEW ⇒ Server Connection Channel**. The Create Server Connection Channel window appears.

3. Specify the name of the channel and an optional description, and then click **OK** to save the channel.

### Install the WebSphere MQ Client

The WebSphere MQ Client must be installed and configured on the client machine. The WebSphere MQ Client is included as part of the typical installation.
Define the Queue Manager Connection on the Client Machine

You can use the following methods to define the connection from the client to the queue manager:

- Set the MQSERVER environment variable. The following code is an example of how to set this variable on Windows:

  ```
  set MQSERVER=ChannelName/TransportType/ConnectionName
  ```

  Here is an example:

  ```
  set MQSERVER=SERVER.CHANNEL1/TCP/server_address(port)
  ```

  In this example, `server_address` is the TCP/IP host name (either the IP address or complete host name) of the server, and `port` is the number of the TCP/IP port on which the server is listening. The port is defined when you create the queue manager. The default port number is 1414. Here is an example:

  ```
  set MQSERVER=SERVER.CHANNEL1/TCP/1.2.3.4(1414)
  ```

- Create a client channel definition table, and set the MQCHLLIB and MQCHLTAB environment variables to identify the location of the table.

  For more information, see the WebSphere MQ documentation at www.ibm.com.

Use the Configured Values in a SAS DATA Step Application

The queue and queue manager values are required in SAS applications that use the WebSphere MQ functional interface. In the previous examples, the queue manager is named MYQMGR, and the queue is named REQUEST. These values are used as follows in the SAS DATA step application:

```sas
hConn=0;
Name="MYQMGR";
compCode=0;
reason=0;
CALL MQCONN(Name, hConn, compCode, reason);

action = "GEN";
parms="OBJECTNAME";
objname="REQUEST";
call mqod(hod, action, rc, parms, objname);

options="INPUT_SHARED";
call mqopen(hconn, hod, options, hobj, compCode, reason);
```

If a SAS application is running as a WebSphere MQ Client, then you must include the following line of code before making any calls using the WebSphere MQ Functional Interface. This line should be placed at the beginning of the application, before the DATA step, as shown in the following example:
%let MQMODEL=CLIENT;
data _null_;...
run;

Table 2.1 Common WebSphere MQ Application Error Codes

<table>
<thead>
<tr>
<th>Reason Code</th>
<th>Explanation</th>
<th>Suggested Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>2018</td>
<td>A connection handle is invalid.</td>
<td>A connection handle that is created by an MQCONN call must be used within the same DATA step where it was created.</td>
</tr>
<tr>
<td>2035</td>
<td>The user is not authorized to perform the attempted action.</td>
<td>Verify that you are connecting to the correct queue and queue manager. Verify that you are authorized to connect to the queue manager. If error is reported to a client connecting to a queue manager, you might need to set the user ID under the MCA tab in the server connection channel definition properties to a user ID that has permission to access the queue manager on the server machine.</td>
</tr>
<tr>
<td>2058</td>
<td>There is an error in the queue manager name.</td>
<td>Check spelling and case of the queue manager name that is used in the application and is defined in the queue manager.</td>
</tr>
<tr>
<td>2059</td>
<td>The queue manager is not available.</td>
<td>Restart the queue manager.</td>
</tr>
<tr>
<td>2085</td>
<td>The object name is unknown.</td>
<td>Check spelling and case of the queue name that is used in the application and is defined in the queue manager.</td>
</tr>
</tbody>
</table>

Message Queue Polling

You can use a message queue polling server to monitor queues and to start SAS programs. This feature can be useful when you want to deploy the WebSphere MQ Functional Interface in high-volume, time-sensitive situations.
Configure Multiple Clients to Read from a Single Queue

The WebSphere MQ interfaces and the message queue polling feature of the object spawner can be used to distribute the processing of messages on a message queue across one or more machines. The result is enhanced performance, load balancing, and hardware redundancy.

Messages can be retrieved only from local queues. In order to enable multiple machines to process messages on a single queue, you must have a full WebSphere MQ (server) installation on the machine that will act as the server. The WebSphere MQ Clients use the queue manager on the server as their queue manager, so any local queues that are defined on that queue manager are also local to the client installations. The WebSphere MQ Clients can connect to a WebSphere MQ server on any supported platform. Message queuing applications on the machine where the queue manager is installed can access the queues directly. Message queuing applications do not need to be configured as clients.

The following diagram illustrates a sample configuration. The queue manager (MYQMGR) is running on Server1 and is managing the queue for each of the WebSphere MQ Clients (CLIENT1, CLIENT2, and CLIENT3). All three clients are communicating with the queue manager through the same server connection channel (SERVER.CHANNEL1). The object spawners on each of the clients can start one or more SAS sessions as needed in order to receive messages from the queue. SAS sessions can also be started by the object spawner and run on the server. A SAS session running on the server does not need to run as a WebSphere MQ Client application; it behaves as a WebSphere MQ server application.
To configure the queue manager on the server, perform the following steps:

1. Define a queue manager if this has not already been done. In the following example, the queue manager is called MYQMGR.
   
crtmqm MYQMGR

2. Start the queue manager by using the WebSphere MQ Explorer (Windows platforms). You can also use the following command on the command line:
   
   strmqm MYQMGR

3. Define one or more local queues that will be used by the applications.
   - To define a local queue from the command line, start the WebSphere MQ command program MQSC. Here is an example:
     
     runmqsc
     MYQMGR
     DEFINE QLOCAL(LOCAL) DEFPSSIT(YES) DESCR('Local Queue')

     Type end to exit MQSC.
   - To define a local queue from the WebSphere MQ Explorer, click MYQMGR to expand the list. Right-click **Queues**, select **New** ⇒ **Local Queue** and enter the queue name and properties.

4. Define a server connection channel to enable WebSphere MQ Clients to communicate with MYQMGR. You can also define a separate server connection channel for each client.
   - To define a server connection channel from the command line, start the WebSphere MQ command program MQSC. Here is an example:
     
     runmqsc MYQMGR
     DEFINE CHANNEL(SERVER.CHANNEL1) CHLTYPE(SVRCONN)
To define a server connection channel from the WebSphere MQ Explorer, click **MYQMGR** \(\Rightarrow\) **Advanced** to expand the list. Right-click **Channels**, select **NEW** \(\Rightarrow\) **Server Connection Channel** and enter the channel name.

5 On each client, install and configure the WebSphere MQ Client. Use the \texttt{MQSERVER} environment variable to define the client connection to the server. The following code shows examples of how to do this in Windows and UNIX operating environments.

- For Windows, use the following code:

  ```
  set MQSERVER=ChannelName/TransportType/ConnectionName
  ```

  Here is an example:

  ```
  set MQSERVER=SERVER.CHANNEL1/TCP/server_address(port)
  ```

  where \texttt{server_address} is the TCP/IP host name of the server and \texttt{port} is the number of the TCP/IP port on which the server is listening. The default port number is 1414. Here is an example:

  ```
  set MQSERVER=SERVER.CHANNEL1/TCP/10.12.0.0(1414)
  ```

- For UNIX, use the following code:

  ```
  export MQSERVER=ChannelName/TransportType/ConnectionName
  ```

  Here is an example:

  ```
  export MQSERVER=SERVER.CHANNEL1/TCP/'10.12.0.0(1414)'
  ```

6 The queue and queue manager values are required in SAS applications that use the WebSphere MQ functional interface. In the previous examples, the queue manager is named \texttt{MYQMGR}, and the queue is named \texttt{LOCAL}. These values are used as follows in the SAS DATA step application:

```
hConn=0;
Name="MYQMGR";
compCode=0;
reason=0;
CALL MQCONN(Name, hConn, compCode, reason);

action = "GEN";
parms="OBJECTNAME";
objname="LOCAL";
call mqod(hod, action, rc, parms, objname);

options="INPUT_SHARED";
call mqopen(hconn, hod, options, hobj, compCode, reason);
```

If a SAS application is running as a WebSphere MQ Client, then you must include the following line of code before making any calls that use the WebSphere MQ Functional Interface. This line should go at the beginning of the application before the DATA step:
This example provides basic configuration information for configuring several clients to receive messages from a queue on one server.

For more information, see the WebSphere MQ documentation at www.ibm.com.
Configuration on the Windows XP Machine

The trigger samples assume that the following configuration objects have been created on the Windows machine:

- a queue manager named XPQMGR
- a local queue named REPLY, with the following settings:

  | Table 2.2  | Configuration Settings for the Local Queue |
  | Queue Name | REPLY                                      |
  | Type       | Local                                      |
  | Put Messages | Allowed                                   |
  | Get Messages | Allowed                                   |
  | Default Priority | 0                                        |
  | Default Persistence | Not Persistent                          |
  | Scope      | Queue Manager                              |
  | Usage      | Normal                                     |

- a remote queue named AIX.TRIGQUEUE, with the following settings:

  | Table 2.3  | Configuration Settings for the Remote Queue |
  | Queue Name | AIX.TRIGQUEUE                               |
  | Type       | Remote                                     |
  | Put Messages | Allowed                                   |
  | Default Priority | 0                                        |
  | Default Persistence | Not Persistent                          |
  | Scope      | Queue Manager                              |
  | Remote Queue Name | TRIGQUEUE                                |
  | Remote Queue Manager Name | AIX                                        |
  | Usage      | XMITQ                                      |
- a receiver channel named XPQMGR.CHANNEL, with the following settings:

  Table 2.4  Configuration Settings for the Receiver Channel

<table>
<thead>
<tr>
<th>Channel Name</th>
<th>XPQMGR.CHANNEL</th>
</tr>
</thead>
<tbody>
<tr>
<td>Type</td>
<td>Receiver</td>
</tr>
<tr>
<td>Transmission Protocol</td>
<td>TCP/IP</td>
</tr>
</tbody>
</table>

- a sender channel named AIX.CHANNEL, with the following settings:

  Table 2.5  Configuration Settings for the Sender Channel

<table>
<thead>
<tr>
<th>Channel Name</th>
<th>AIX.CHANNEL</th>
</tr>
</thead>
<tbody>
<tr>
<td>Type</td>
<td>Sender</td>
</tr>
<tr>
<td>Transmission Protocol</td>
<td>TCP/IP</td>
</tr>
<tr>
<td>Connection Name</td>
<td>AIX-machine-name</td>
</tr>
<tr>
<td>Transmission Queue</td>
<td>XMITQ</td>
</tr>
</tbody>
</table>

- a process definition named AIX.PROCESS, with the following settings:

  Table 2.6  Configuration Settings for the Process Definition

<table>
<thead>
<tr>
<th>Process Definition Name</th>
<th>AIX.PROCESS</th>
</tr>
</thead>
<tbody>
<tr>
<td>Application Type</td>
<td>Windows NT</td>
</tr>
<tr>
<td>User Data</td>
<td>AIX.CHANNEL</td>
</tr>
</tbody>
</table>

- a transmission queue named XMITQ, with the following settings:

  Table 2.7  Configuration Settings for the Transmission Queue

<table>
<thead>
<tr>
<th>Queue Name</th>
<th>XMITQ</th>
</tr>
</thead>
<tbody>
<tr>
<td>Type</td>
<td>Local</td>
</tr>
<tr>
<td>Put Messages</td>
<td>Allowed</td>
</tr>
<tr>
<td>Get Messages</td>
<td>Allowed</td>
</tr>
<tr>
<td>Default Priority</td>
<td>0</td>
</tr>
<tr>
<td>Default Persistence</td>
<td>Not Persistent</td>
</tr>
</tbody>
</table>
**Configuration on the AIX Machine**

The following code can either be a part of a configuration file, or stanzas that can be entered in the `runmqsc` tool. Modify the following templates and use the WebSphere MQ tool `runmqsc` to define the required objects on a queue manager that is named AIX for this example:

* Local Queue that triggers the batch job to start SAS
  
  ```
  DEFINE QLOCAL(TRIGQUEUE) +
  REPLACE DEFPERSIST(YES) DESCR('TRIGQUEUE Queue') +
  INITQ(MY.INITQ) +
  TRIGGER TRIGTYPE(EVERY) PROCESS(TRIGSAS.PROCESS)
  ```

* TRIGTYPE can also be FIRST or DEPTH. EVERY will trigger the batch job every time a message arrives on the queue.

* Process to start the batch file that starts SAS
  
  ```
  DEFINE PROCESS(TRIGSAS.PROCESS) +
  REPLACE APPLICID('/users/userid/startsas') APPLTYPE(UNIX)
  ```

* Receiver Channel for AIX Queue Manager
  
  ```
  DEFINE CHANNEL(AIX.CHANNEL) CHLTYPE(RECV) +
  REPLACE DESCR('Receiver Channel on AIX') +
  TRP TYPE(TCP)
  ```

* Remote definitions for Windows XP queue manager ---*

* Remote Queue at XPQMGRT
  
  ```
  DEFINE QREMOTE(XPQMGRT.REPLY) +
  REPLACE RNAME(REPLY) RQMNAME(XPQMGRT) XMITQ(XPQMGRT.XMITQ)
  ```

* Transmission Queue
DEFINE QLOCAL(XPQMGR.XMITQ) +
    REPLACE DESCR('Transmit Queue to XP system') +
    USAGE(XMITQ) TRIGGER TRIGTYPE(FIRST) +
    INITQ(SYSTEM.CHANNEL.INITQ) PROCESS(XPQMGR.PROCESS)

* Process definition for XMITQ trigger
DEFINE PROCESS(XPQMGR.PROCESS) +
    REPLACE DESCR('Process definition +
    to start XPQMGR Channel') +
    USERDATA('XPQMGR.CHANNEL')

* Sender Channel - started automatically
* when first message written to XMITQ
DEFINE CHANNEL(XPQMGR.CHANNEL) CHLTYPE(SDR) +
    REPLACE DESCR('Sender Channel to XPQMGR') +
    TRPTYPE(TCP) XMITQ(XPQMGR.XMITQ) +
    CONNAME('XPMACHINE.MYLOCATION.MYCOMPANY.COM')

*---- Setup Client/Server Server Connection Channel ----*
DEFINE CHANNEL(MQCLIENT.CHANNEL) +
    CHLTYPE(SVRCONN) TRPTYPE(TCP) +
    REPLACE DESCR('Server connection for client access') +
    MCAUSER('')

This example uses /users/userid/startsas as the name of the batch file triggered
to run a SAS DATA step. The contents of this file are:

# Make sure the 64-bit
WebSphere MQ client
# libraries are in your LIBPATH.
export LIBPATH=/usr/mqm/lib64

# Define the server that the SAS WebSphere MQ
# client interface will connect through.
export MQSERVER=
    MQCLIENT.CHANNEL/TCP/"server IP address>({port})"
sas -sysin /users/userid/mqserver.sas

You must also make sure that the trigger monitor has been started on the AIX
machine for the proper initiation queue:
runmqsc -m AIX -q MY.INITQ

--- Sample Trigger Programs ---

mqclient.sas

The following program runs on the Windows XP machine and requests a data set:

data _null_;
length msg $ 200;
length qid2 tid rc 8;
length map $80;
length recv1 $50;
length event $10;
length rpname $256;
length type $8;
length qual1 qual2 $40;

libname out '.';
tid=0;
rc=0;
put '----';
put 'Call INIT';
CALL INIT(tid, 'MQSERIES', rc);
if rc ^= 0 then do;
   put 'INIT: failed';
   msg = sysmsg();
   put msg;
end;
else put 'INIT: succeeded';

rc=0;
qid=0;
put '----';
put 'Call OPENQUEUE to open the response queue';
CALL OPENQUEUE(qid, tid, 'XPQMGR:REPLY', 'fetch',
   rc, "POLL(TIMEOUT=20)");
if rc ^= 0 then do;
   put 'OPENQUEUE: failed';
   msg = sysmsg();
   put msg;
end;
else put 'OPENQUEUE: succeeded';

rc=0;
qid2=0;
put 'Call OPENQUEUE to open the request queue on qid2';
CALL OPENQUEUE(qid2, tid, 'XPQMGR:AIX.TRIGQUEUE',
   'DELIVERY', rc, "POLL(Timeout=15)");
if rc ^= 0 then do;
   put 'OPENQUEUE: failed';
   msg = sysmsg();
   put msg;
end;
else put 'OPENQUEUE: succeeded';

rc=0;
put '----';
put 'Call SETMAP';
CALL SETMAP('mqclientmap', 'REGISTRY', rc, 'CHAR,,50');
if rc ^= 0 then do;
   put 'SETMAP: failed';
   msg = sysmsg();
   put msg;
end;
else put 'SETMAP: succeeded';

parm1="calories";
put '---- Send a message to the request queue qid
requesting the specified data set ----';
put 'Call SENDMESSAGE';
call sendmessage(qid2,rc,"map, respqueue",
"mqclientmap","R64:D8650",parm1);
if rc ^= 0 then do;
    put 'send message failed: ';
    msg=sysmsg();
    put msg;
end;
else put 'send message succeeded';
slept = sleep(1);
rc = 0;
put '---- receive a data set from the reply queue ----';
put 'Call RECEIVEMESSAGE';
map = "mqclientmap";
call receivemessage(qid, rc, event,
attchflg,"map", map, recv1);
put 'response queue = ' rpname;
put 'qid = ' qid;
put 'event = ' event;
put 'attchflg = ' attchflg;
if rc ^= 0 then do;
    put 'receive message failed: ';
    msg=sysmsg();
    put msg;
end;
else do;
    put 'receive message succeeded';
    put "map = " map;
    put "recv1 = " recv1;
end;
if event eq 'DELIVERY' then do;
    put 'Message has been delivered';
    if attchflg = 1 then do;
        put '---- check for attachments ----';
call getattachment(qid, lastflag, attachid,
    type, qual1, qual2, rc);
if rc ^= 0 then do;
    put 'get attachment failed: '
    msg=sysmsg();
    put msg;
end;
else put 'get attachment succeeded';
if type="DATASET" then do;
    put '--- accept attachment into a data set ---';
    put "qual2 = " qual2;
call acceptattachment(qid, attachid,
"out", qual2, rc);
if rc ^= 0 then do;
   put 'accept DATASET failed: ';
   msg=sysmsg();
   put msg;
   end;
else put 'accept DATASET succeeded';
end;
end;

rc=0;
put '----';
put 'Call CLOSEQUEUE for queue1';
CALL CLOSEQUEUE(qid, rc);
if rc ^= 0 then do;
   put 'CLOSEQUEUE: failed';
   msg = sysmsg();
   put msg;
end;
else put 'CLOSEQUEUE: succeeded';

rc=0;
put '----';
put 'Call CLOSEQUEUE for queue2';
CALL CLOSEQUEUE(qid2, rc);
if rc ^= 0 then do;
   put 'CLOSEQUEUE: failed';
   msg = sysmsg();
   put msg;
end;
else put 'CLOSEQUEUE: succeeded';

rc=0;
put '----';
put 'Call TERM';
CALL TERM(tid, rc);
if rc ^= 0 then do;
   put 'TERM: failed';
   msg = sysmsg();
   put msg;
end;
else put 'TERM: succeeded';

run;

mqserver.sas

The following program runs on the AIX machine and returns a data set:

data calories;
   input item $ 1 - 16 calories 18-20 ;
datalines;
   ground beef 230
hot dog          100
banana           100
broccoli          45
skim milk         50

data _null_

length msg $ 200;
length qid qid2 tid rc 8;
length map $80;
length recv1 $50;
length attachname $21;
length event $10;
length rpname $256;
tid=0;
rc=0;

put '----';
put 'Call INIT';
CALL INIT(tid, 'MQSERIES-C', rc);
if rc ^= 0 then do;
  put 'INIT: failed';
  msg = sysmsg();
  put msg;
end;
else put 'INIT: succeeded';
rc=0;
quid=0;
put '----';
put 'Call OPENQUEUE for queue1';
CALL OPENQUEUE(qid, tid, 'AIX:TRIGQUEUE', 'fetch', rc, "POLL(Timeout=10)");
if rc ^= 0 then do;
  put 'OPENQUEUE: failed';
  msg = sysmsg();
  put msg;
end;
else put 'OPENQUEUE: succeeded';
rc=0;
put '----';
put 'Call SETMAP';
CALL SETMAP('mqservermap', 'REGISTRY', rc, 'CHAR,,50');
if rc ^= 0 then do;
  put 'SETMAP: failed';
  msg = sysmsg();
  put msg;
end;
else put 'SETMAP: succeeded';

rc = 0;
put '---- receive a message from the remote queue ----';
put 'Call RECEIVEMESSAGE';
map = "mqservermap";
rpname=' ';
call receivemessage(qid, rc, event, attchflg,"map, respqueue", map, rpname, recv1);
put 'recv1 =' recv1;
put 'response queue =' rpname;
pull 'qid = ' qid;
pull 'event = ' event;
pull 'attchflg = ' attchflg;

if rc ^= 0 then do;
  put 'receive message failed: ';
  msg=sysmsg();
  put msg;
end;
else do;
  put 'receive message succeeded';
  put map;
end;

if event eq 'DELIVERY' then do;
  rc = 0;
  qid2=0;

  put '---- open the response queue qid2 ----';
  put 'Call OPENQUEUE for queue2';
  CALL OPENQUEUE(qid2, tid, rpname, 'delivery',
                  rc, "POLL(Timeout=15)");
  if rc ^= 0 then do;
    put 'OPENQUEUE: failed';
    msg = sysmsg();
    put msg;
  end;
  else put 'OPENQUEUE: succeeded';
  put 'rpname =' rpname;

  put '---- send the requested data set to the response queue ----';
  put 'Call SENDMESSAGE';
  attachname = 'dataset,work,' || recv1;
  put "attachname = " attachname;
  call sendmessage(qid2,rc,"map, attachlist",
                  "mqservermap",attachname, recv1 );
  if rc ^= 0 then do;
    put 'send message failed: ';
    msg=sysmsg();
    put msg;
  end;
  else put 'send message succeeded';

rc=0;
put '----';
put 'Call CLOSEQUEUE for queue2';
CALL CLOSEQUEUE(qid2, rc);
if rc ^= 0 then do;
  put 'CLOSEQUEUE: failed';
msg = sysmsg();
put msg;
end;
else put 'CLOSEQUEUE: succeeded';
end;
rc=0;
put '----';
put 'Call CLOSEQUEUE for queue1';
CALL CLOSEQUEUE(qid, rc);
if rc ^= 0 then do;
    put 'CLOSEQUEUE: failed';
    msg = sysmsg();
    put msg;
end;
else put 'CLOSEQUEUE: succeeded';
rc=0;
put '----';
put 'Call TERM';
CALL TERM(tid, rc);
if rc ^= 0 then do;
    put 'TERM: failed';
    msg = sysmsg();
    put msg;
end;
else put 'TERM: succeeded';
run;
SAS Integration Technologies allows application developers to combine the power of both SAS information delivery and IBM message queuing capabilities by providing a SAS interface to the IBM WebSphere MQ product (formerly called MQSeries). With this interface, SAS programs can create new WebSphere MQ message queues or take advantage of existing ones that are available throughout the enterprise. This section explains how to implement this interface by using the SAS DATA step and SAS Macro Language.

Note: WebSphere MQ enables you to trigger, or start, an application automatically when a message arrives on a message queue. For more information, see "Configuring WebSphere MQ to Trigger SAS: An Example" on page 13.
Writing WebSphere MQ Applications

Overview of Writing WebSphere MQ Applications

With WebSphere MQ messaging, two or more applications communicate with each other indirectly and asynchronously using message queues. The applications do not have to be running at the same time or even in the same operating environment. An application can communicate with another application by sending a message to a queue. The receiving application retrieves the message when it is ready.

A typical SAS program using WebSphere MQ services performs the following tasks:

1. Establishes a connection to a WebSphere MQ queue manager. The queue manager is responsible for maintaining the queues and for ensuring that the messages in the queues reach their destination. This insulates the application developer from the details of the network. When a successful connection is made, the queue manager issues a connection handle that is used to identify the connection in subsequent function calls.

   Note: A program can have connections to more than one queue manager if the platform supports multiple queue managers running on it.

2. Opens the desired queue. When opening a queue, the program must define how it intends to use it. For example, the program can send (put) messages to the queue, receive (get) messages from the queue, or it can do both. If a queue is opened by using the INQUIRE option, then the queue can be queried for information about the queue itself. Similarly, if the queue is opened using the SET option, then various queue attributes can be set. If the queue is opened successfully, then the queue manager issues an object handle that is used to identify the queue in subsequent function calls.

3. (Optional) Puts messages on the queue by using the CALL MQPUT routine. The queue is identified using the connection handle for the queue manager and the object handle for the queue. In addition, several other functions are available for creating and manipulating the data in the message as well as setting options that help the receiving program locate the message in the queue.

4. (Optional) Opens the same queue (or a different one) for retrieving messages. The program uses the MQGET routine specifying the connection handle to the queue manager and the object handle for the queue from which it wants to retrieve the message. There are a number of options that can be set to help identify the message to get from the queue.

5. (Optional) Releases the resources allocated by a SAS internal handle. These resources are associated with message options and descriptors.
Interface Models

WebSphere MQ provides two Message Queue Interface (MQI) models:

Base/Server model
- runs on the same machine as the WebSphere MQ Base product and WebSphere MQ Server

Client model
- runs on a different machine from the WebSphere MQ Base product and WebSphere MQ Server

IBM requires programs to be linked with different libraries according to the model that will be used. The default model that is assumed by SAS is the Base/Server model. If you do not want the default model, then you must specify the MQMODEL SAS macro variable and set it to a value of CLIENT:

```sas
%let MQMODEL=CLIENT;
```

You must set this variable before calling any WebSphere MQ interface function.

If the program is using the client model, then it opens a remote queue manager. WebSphere MQ clients always connect across a network. For information about configuring remote access, see “Configuring WebSphere MQ Client Access” on page 7.

Data Conversion

Overview of Data Conversion

If you will be putting or getting messages from heterogeneous systems, then data conversion must be considered. Data conversion is usually categorized as follows:

- Character data conversion
- Numeric data conversion

The Coded Character Set ID (CCSID) or code page is a number that represents a character translation table to be used between two distinct systems. Encoding is the term generally used to represent how numeric data is represented on a particular system. WebSphere MQ channel communication (Transmission Segment Header and Message Descriptor) data are converted internally by WebSphere MQ. However, the user portion of a message is not. It is the responsibility of the program to convert this data.

Data conversion of this user portion can be handled by either WebSphere MQ conversion exit routines or by SAS.

Converting Data within WebSphere MQ

For WebSphere MQ to perform the data conversion of the user portion of a message, you must perform the following steps:
When putting (MQPUT) a message on a queue, specify the FORMAT (conversion exit) that the receiver should use to convert the incoming message.

Convert a message:
- WebSphere MQ provides an internal conversion format, MQSTR, that can be used to convert a message comprised entirely of character data.
- If the message is not comprised entirely of character data, then you must create a conversion exit.

The receiving (MQGET) program must tell WebSphere MQ to do the required data conversion based on the incoming message format and data encoding. The program does this by specifying the CONVERT option on the Get Message Options, which is part of the MQGET call. If you do not want to set up static conversion exit routines, then you can let SAS convert the data for you as an alternative solution.

For more information about conversion within WebSphere MQ, see the WebSphere MQ documentation at www.ibm.com.

Converting Data in SAS

By default, if you do not specify the CONVERT Get Message Option, then SAS converts the data conversion to the default encoding for the SAS session. To disable the automatic SAS data conversion, specify the MQSASCNV SAS macro variable and set it to a value of DISABLE or OFF:

```sas
%let MQSASCNV=OFF;
```

You can use the KCVT function to convert your data manually. Converting data manually is especially useful for programs that put reply messages on a queue.

For example, your SAS program might use messaging to interact with a Java web application that uses UTF-8 encoding. When receiving messages, SAS automatically converts the message data to the session encoding. However, you must convert the data back to UTF-8 before sending it to the reply queue. The following code converts the variable TEXT from WLatin1 to UTF-8:

```sas
text= kcvt(text, wlatin1, utf8));
```

For more information, see "KCVT Function" in SAS National Language Support (NLS): Reference Guide.

WebSphere MQ Coding Examples

Introduction to the WebSphere MQ Examples

This section contains examples of using the WebSphere MQ interface to send and receive messages to and from application messaging queues.

Please note the following points about freeing resources used in conjunction with the WebSphere MQ Interface:
When a SAS DATA step ends, all resources consumed by this DATA step are automatically freed. That is, all internal SAS handles are automatically freed, as well as being disconnected from all queue managers that were connected during this DATA step execution. However, it is good programming practice to free these resources using the functions provided.

When using the SAS Macro Language to interface with WebSphere MQ, ensure that all resources are freed programmatically. Unlike the DATA step, resources consumed by the SAS Macro Language are never implicitly freed during SAS execution.

DATA Step Coding Example: Put a Message on a Queue

This example puts a message on a queue.

data _null_; length hconn hobj cc reason 8; length rc hod hpmo hmap hdata 8; length parms $ 200 options $ 200 action $ 3 msg $ 200; hconn=0; hobj=0; hod=0; hpmo=0; hmap=0; hdata=0; put '---------------- Connect to QMgr --------------'; qmgr="TEST"; call mqconn(qmgr, hconn, cc, reason); if cc ^= 0 then do; if reason = 2002 then do; put 'Already connected to QMgr ' qmgr; end; else do; if reason = 2059 then put 'MQCONN: QMgr not available... needs to be started'; else put 'MQCONN: failed with reason= ' reason; goto exit; end; else put 'MQCONN: successfully connected to QMgr ' qmgr; put '---------- Generate object descriptor ---------'; action="GEN"; parms="OBJECTNAME"; objname="TEST"; call mqod(hod, action, rc, parms, objname); if rc ^= 0 then do; put 'MQOD: failed with rc= ' rc; msg = sysmsg();
put msg;
goto exit;
end;
else put 'MQOD: successfully generated
object descriptor';

put '-------- Open queue object for output ---------';
options="OUTPUT";
call mqopen(hconn, hod, options, hobj, cc, reason);
if cc ^= 0 then do;
put 'MQOPEN: failed with reason= ' reason;
goto exit;
end;
else put 'MQOPEN: successfully opened queue for output';

put '-------- Generate put message options --------';
call mqpmo(hpmo, action, rc);
if rc ^= 0 then do;
put 'MQPMO: failed with rc= ' rc;
msg = sysmsg();
put msg;
goto exit;
end;
else put 'MQPMO: successfully generated put
message options';

put '-------- Generate message descriptor --------';
parms="PERSISTENCE";
persist="PERSISTENT"; /* persistent message */
call mqmd(hmd, action, rc, parms, persist);
if rc ^= 0 then do;
put 'MQMD: failed with rc= ' rc;
msg = sysmsg();
put msg;
goto exit;
end;
else put 'MQMD: successfully generated
message descriptor';

put '----------- Generate map descriptor -----------';
/* data will not be aligned */
desc1="SHORT";
desc2="LONG";
desc3="DOUBLE";
desc4="CHAR,,50"; /* blank pad to 50 bytes */
call mqmap(hmap, rc, desc1, desc2, desc3, desc4);
if rc ^= 0 then do;
put 'MQMAP: failed with rc= ' rc;
msg = sysmsg();
put msg;
goto exit;
end;
else put 'MQMAP: successfully generated map descriptor';

put '--- Generate data descriptor - actual data ----';
parm1=100;
parm2=9999;
parm3=9999.9999;
parm4="This is a test.";
call mqsetparms(hdata, hmap, rc, parm1,
parm2, parm3, parm4);
if rc ^= 0 then do;
   put 'MQSETPARMS: failed with rc= ' rc;
   msg = sysmsg();
   put msg;
   goto exit;
end;
else put 'MQSETPARMS: successfully generated
data descriptor';

put '------------- Put message on queue ------------';
call mqput(hconn, hobj, hmd, hpmo, hdata, cc, reason);
if cc ^= 0 then do;
   put 'MQPUT: failed with reason= ' reason;
   goto exit;
end;
else do;
   put 'MQPUT: successfully put message on queue';

/* inquire about message descriptor
 output parameters */
action="INQ";
parms="MSGID,PUTAPPLTYPE,PUTAPPLNAME,
PUTDATE,PUTTIME";

length msgid $ 48 applname $ 28 date $ 8 time $ 8;
call mqmd(hmd, action, rc, parms, msgid, appltype,
applname, date, time);
if rc ^= 0 then do;
   put 'MQMD: failed with rc= ' rc;
   msg = sysmsg();
   put msg;
end;
else do;
   put 'Message descriptor output parameters are:';
   put 'MSGID= ' msgid;
   put 'PUTAPPLTYPE= ' appltype;
   put 'PUTAPPLNAME= ' applname;
   put 'PUTDATE= ' date;
   put 'PUTTIME= ' time;
end;
exit:
if hobj ^= 0 then do;
   put '----------------- Close queue -----------------';
   options="NONE";
   call mqclose(hconn, hobj, options, cc, reason);
   if cc ^= 0 then do;
       put 'MQCLOSE: failed with reason= ' reason;
   end;
   else put 'MQCLOSE: successfully closed queue';
end;

if hconn ^= 0 then do;
   put '------------ Disconnect from QMgr -----------';
   call mqdisc(hconn, cc, reason);
   if cc ^= 0 then do;
       put 'MQDISC: failed with reason= ' reason;
   end;
   else put 'MQDISC: successfully disconnected from QMgr';
end;

if hod ^= 0 then do;
   call mqfree(hod);
   put 'Object descriptor handle freed';
end;
if hpmo ^= 0 then do;
   call mqfree(hpmo);
   put 'Put message options handle freed';
end;
if hmd ^= 0 then do;
   call mqfree(hmd);
   put 'Message descriptor handle freed';
end;
if hmap ^= 0 then do;
   call mqfree(hmap);
   put 'Map descriptor handle freed';
end;
if hdata ^= 0 then do;
   call mqfree(hdata);
   put 'Data descriptor handle freed';
end;
run;

DATA Step Coding Example: Retrieve a Message
This example retrieves a message from a queue.

   data _null_;
   length hconn hobj cc reason 8;
length rc hod hgmo hmd hmap msglen 8;
length parms $ 200 options $ 200 action $ 3 msg $ 200;
hconn=0;
hobj=0;
hod=0;
hgmo=0;
hmd=0;
hmap=0;

put '---------------- Connect to QMgr ----------------';
qmgr="TEST";
call mqconn(qmgr, hconn, cc, reason);
if cc ^= 0 then do;
   if reason = 2002 then do;
      put 'Already connected to QMgr ' qmgr;
   end;
   else do;
      if reason = 2059 then
         put 'MQCONN: QMgr not available...
                      needs to be started';
      else
         put 'MQCONN: failed with reason= ' reason;
      goto exit;
   end;
end;
else put 'MQCONN: successfully connected to QMgr ' qmgr;

put '---------- Generate object descriptor ----------';
action="GEN";
parms="OBJECTNAME";
objname="TEST";
call mqod(hod, action, rc, parms, objname);
if rc ^= 0 then do;
   put 'MQOD: failed with rc= ' rc;
   msg = sysmsg();
   put msg;
   goto exit;
end;
else put 'MQOD: successfully generated
        object descriptor';

put '-------- Open queue object for input --------';
options="INPUT_SHARED";
call mqopen(hconn, hod, options, hobj, cc, reason);
if cc ^= 0 then do;
   put 'MQOPEN: failed with reason= ' reason;
   goto exit;
end;
else put 'MQOPEN: successfully opened queue for output';

put '-------- Generate get message options --------';
call mqgmo(hgmo, action, rc);
if rc ^= 0 then do;
put 'MQGMO: failed with rc= ' rc;
msg = sysmsg();
put msg;
goto exit;
end;
else put 'MQGMO: successfully generated get message options';

put '--------- Generate message descriptor ---------';
call mqmd(hmd, action, rc);
if rc ^= 0 then do;
  put 'MQMD: failed with rc= ' rc;
  msg = sysmsg();
  put msg;
  goto exit;
end;
else put 'MQMD: successfully generated message descriptor';

put '--------- Generate map descriptor ---------';
desc1="SHORT";
desc2="LONG";
desc3="DOUBLE";
desc4="CHAR,,50";
call mqmap(hmap, rc, desc1, desc2, desc3, desc4);
if rc ^= 0 then do;
  put 'MQMAP: failed with rc= ' rc;
  msg = sysmsg();
  put msg;
  goto exit;
end;
else put 'MQMAP: successfully generated map descriptor';

put '--------- Get message from queue ---------';
call mqget(hconn, hobj, hmd, hgmo, msglen, cc, reason);
if cc ^= 0 then do;
  if reason = 2033 then put 'No message available';
  else put 'MQGET: failed with reason= ' reason;
  goto exit;
end;
else do;
  put 'MQGET: successfully retrieved message from queue';
  put 'message length= ' msglen;

  /* inquire about message descriptor output parameters */
  action="INQ";
  parms="REPORT,MSGTYPE,FEEDBACK,MSGID,
         CORRELID,USERIDENTIFIER,PUTAPPLTYPE,
         PUTAPPLNAME,PUTDATE,PUTTIME";
length report $ 30 msgtype 8 feedback 8 msgid $ 48
correlid $ 48 userid $ 12 appltype 8
applname $ 28 date $ 8 time $8;
call mqmd(hmd, action, rc, parms, report,
msgtype, feedback, msgid, correlid, userid,
appltype, applname, date, time);
if rc ^= 0 then do;
   put 'MQMD: failed with rc= ' rc;
   msg = sysmsg();
   put msg;
end;
else do;
   put 'Message descriptor output parameters are:';
   put 'REPORT= ' report;
   put 'MSGTYPE= ' msgtype;
   put 'FEEDBACK= ' feedback;
   put 'MSGID= ' msgid;
   put 'CORRELID= ' correlid;
   put 'USERIDENTIFIER= ' userid;
   put 'PUTAPPLTYPE= ' appltype;
   put 'PUTAPPLNAME= ' applname;
   put 'PUTDATE= ' date;
   put 'PUTTIME= ' time;
end;
end;

if msglen > 0 then do;
   /* retrieve SAS variables from GET buffer */
   length parm1 parm2 parm3 8 parm4 $ 50;
call mqgetparms(hmap, rc, parm1,
   parm2, parm3, parm4);
   put 'Display SAS variables:';
   put 'parm1= ' parm1;
   put 'parm2= ' parm2;
   put 'parm3= ' parm3;
   put 'parm4= ' parm4;
   if rc ^= 0 then do;
      put 'MQGETPARMS: failed with rc= ' rc;
      msg = sysmsg();
      put msg;
   end;
end;
else put 'No data associated with message';

exit:
if hobj ^= 0 then do;
   put '----------------- Close queue ---------------';
options="NONE";
call mqclose(hconn, hobj, options, cc, reason);
if cc ^= 0 then do;
   put 'MQCLOSE: failed with reason= ' reason;
end;
else put 'MQCLOSE: successfully closed queue';
end;

if hconn ^= 0 then do;
   put '----------- Disconnect from QMgr -----------';
   call mqdisc(hconn, cc, reason);
   if cc ^= 0 then do;
      put 'MQDISC: failed with reason= ' reason;
   end;
   else put 'MQDISC: successfully disconnected
      from QMgr';
end;

if hod ^= 0 then do;
   call mqfree(hod);
   put 'Object descriptor handle freed';
end;
if hgmo ^= 0 then do;
   call mqfree(hgmo);
   put 'Get message options handle freed';
end;
if hmd ^= 0 then do;
   call mqfree(hmd);
   put 'Message descriptor handle freed';
end;
if hmap ^= 0 then do;
   call mqfree(hmap);
   put 'Map descriptor handle freed';
end;

run;

---

Processing a Text File

This example puts a text file to a queue.

/** bits
   within md.msgflags **/
   %let segment_allow=1;
   %let segment=2;
   %let last_segment=4;
   %let group=8;
   %let last_group=16;

   data _null_;
   length rc 8;
   length msg $ 200;
   length hconn hod hpmo hobj hmd hmap hdata 8;
   length cc reason 8;
   length record $ 256;
   length msgflags 8;
/* send this file to the queue */
infile 'd:\test.txt' length=reclen end=eof;

call mqconn("TESTQMGR", hconn, cc, reason);
if cc ^= 0 then do;
   if reason = 2002 then do;
      put 'Already connected to QMgr';
   end;
   else do;
      if reason = 2059 then
         put 'MQCONN: QMgr not available...
            needs to be started';
      else
         put 'MQCONN: failed with reason= ' reason;
         goto exit;
   end;
end;

put '---------- Generate object descriptor ---------';
call mqod(hod, "GEN", rc, "OBJECTNAME", "TESTQ");
if rc ^= 0 then do;
   put 'MQOD: failed with rc= ' rc;
   msg = sysmsg();
   put msg;
   goto exit;
end;

put '-------- Open queue object for output --------';
call mqopen(hconn, hod, "OUTPUT", hobj, cc, reason);
if cc ^= 0 then do;
   put 'MQOPEN: failed with reason= ' reason;
   goto exit;
end;

put '--------- Generate put message options ---------';
/** QMgr will generate a unique msgid on every put as **/
/** well as generate a groupid for all of the msgs    **/
/** and incrementally keep up with the sequencing...  **/
call mqpmo(hpmo, "GEN", rc, "OPTIONS",
   "NEW_MSGID,LOGICAL_ORDER");
if rc ^= 0 then do;
   put 'MQPMO: failed with rc= ' rc;
   msg = sysmsg();
   put msg;
   goto exit;
end;

put '--------- Generate message descriptor ---------';
/** specify the message belongs to a group **/
msgflags=group;
call mqmd(hmd, "GEN", rc, "PERSISTENCE,MSGTYPE,MSGFLAGS",
   "PERSISTENT", 100000, msgflags);
if rc ^= 0 then do;
   put 'MQMD: failed with rc= ' rc;
end;
msg = sysmsg();
put msg;
goto exit;
end;

put '----------- Generate map descriptor -----------';
/* longest record in file is 255 bytes+1 length byte... */
/* therefore all messages on the queue pertaining to */
/* this file will be blank-padded for 256 bytes... */
call mqmap(hmap, rc, "char,,256");
if rc ^= 0 then do;
   put 'MQMAP: failed';
   msg = sysmsg();
   put msg;
   goto exit;
end;

do until(eof);
   input @;
   input record $varying256. reclen;

   call mqsetparms(hdata, hmap, rc, record);
   if( rc ) then do;
      put 'MQSETPARMS: failed';
      msg = sysmsg();
      put msg;
      goto exit;
   end;

   /** set last in group if eof **/
   if( eof ) then do;
      msgflags + last;_group;
      call mqmd(hmd, "SET", rc, "MSGFLAGS", msgflags);
      if rc ^= 0 then do;
         put 'MQMD: failed with rc= ' rc;
         msg = sysmsg();
         put msg;
         goto exit;
      end;
   end;

   put '--- Put msg on queue ----';
call mqput(hconn, hobj, hmd, hpmo, hdata,
cc, reason);
if cc ^= 0 then do;
   put 'MQPUT: failed with reason= ' reason;
   msg = sysmsg();
   put msg;
   goto exit;
end;

   /* free data */
call mqfree(hdata);
end;
exit:
if( hobj ) then do;
call mqclose(hconn, hobj, "NONE", cc, reason);
if( cc ) then do;
   put 'MQCLOSE: failed with reason = ' reason;
   msg = sysmsg();
   put msg;
end;
end;

if( hconn ) then do;
call mqdisc(hconn, cc, reason);
if( cc ) then do;
   put 'MQDISC: failed with reason = ' reason;
   msg = sysmsg();
   put msg;
end;
end;

if hod ^= 0 then do;
call mqfree(hod);
   put 'Object descriptor handle freed';
end;
if hpmo ^= 0 then do;
call mqfree(hpmo);
   put 'Put message options handle freed';
end;
if hmd ^= 0 then do;
call mqfree(hmd);
   put 'Message descriptor handle freed';
end;
if hmap ^= 0 then do;
call mqfree(hmap);
   put 'Map descriptor handle freed';
end;
stop;
run;

Getting a Text File from a Queue

This example gets a text file from a queue.

/* Get
first text file on a queue... ie. msgtype=100000 */
/* This example opens queue with a browse cursor and */
/* browses the first msg in every group looking for */
/* a msg with msgtype=100000... once it is found, */
/* open a fetch instance to remove all msgs in that */
/* particular group... */

/* if you knew upfront the groupid that you wanted, you */
/* could just open a single instance of the queue and */
/* remove the group in logical order without having to */
/* do any initial browsing... */

/* bit test macros */
%let segment_allow_mask='.......1'b;
%let segment_mask='......1.'b;
%let last_segment_mask='.....1..'b;
%let group_mask='....1...'b;
%let last_group_mask='...1....'b;

filename output 'd:\testdup.txt';
data _null_; length rc 8; length msg $ 200; length cc reason 8; length hconn hod hgmo hobj hmap 8; length record $ 256; length msgtype seqno msgflags 8; length groupid $ 48; fileid = fopen('output', 'o', 256, 'v'); if( fileid = 0 ) then do; put 'Error opening output file...'; goto exit; end;

put '---------------- Connect to QMgr -----------------

call mqconn("TESTQMGR", hconn, cc, reason); if cc ^= 0 then do; if reason = 2002 then do; put 'Already connected to QMgr'; end; else do; if reason = 2059 then put 'MQCONN: QMgr not available... needs to be started'; else put 'MQCONN: failed with reason= ' reason; goto exit; end; end;

put '---------- Generate object descriptor ----------

call mqod(hod, "GEN", rc, "OBJECTNAME", "TESTQ"); if rc ^= 0 then do; put 'MQOD: failed with rc= ' rc; msg = sysmsg(); put msg; goto exit; end;
put '--------- Open queue object for input ---------';
call mqopen(hconn, hod, "INPUT_SHARED,BROWSE", hobj,
cc, reason);
if cc ^= 0 then do;
   put 'MQOPEN: failed with reason= ' reason;
   goto exit;
end;

put '--------- Generate get message options --------';
call mqgmo(hgmo, "GEN", rc, "options, matchoptions",
   "browse_next", "seqnumber");
if rc ^= 0 then do;
   put 'MQGMO: failed with rc= ' rc;
   msg = sysmsg();
   put msg;
   goto exit;
end;

put '--------- Generate message descriptor ---------';
/** browse first msg in group only **/
call mqmd(hmd, "GEN", rc, "msgseqnumber", 1);
if rc ^= 0 then do;
   put 'MQMD: failed with rc= ' rc;
   msg = sysmsg();
   put msg;
   goto exit;
end;

recv:
call mqget(hconn, hobj, hmd, hgmo, msglen, cc, reason);
if( cc ) then do;
   if( reason = 2033 ) then do;
      put 'reached end of queue';
      goto exit;
   end;
   else do;
      put 'MQGET: failed with reason = ' reason;
      msg = sysmsg();
      put msg;
      goto exit;
   end;
end;

/* inquire about msg properties */
call mqmd(hmd, "INQ", rc,
   "MSGTYPE,GROUPID,MSGSEQNUMBER,MSGFLAGS",
   msgtype, groupid, seqno, msgflags);
if( rc ) then do;
   put 'MQMD failed';
   msg = sysmsg();
   put msg;
   goto exit;
end;
put msgtype=
put groupid=
put seqno=
put msgflags=

if( msgtype = 100000 ) then do;
  /* file processing... */

  put '---------- Generate map descriptor ----------';
  /* all file messages were sent to the queue as
   256 bytes blank-padded */
call mqmap(hmap, rc, "char,,256");
if( rc ) then do;
  put 'MQMAP: failed';
  msg = sysmsg();
  put msg;
  goto exit;
end;

  /* close browse instance */
call mqclose(hconn, hobj, "NONE", cc, reason);
if( cc ) then do;
  put 'MQCLOSE: failed with reason = ' reason;
  msg = sysmsg();
  put msg;
end;

  /* open queue in fetch mode */
hobj=0;
call mqopen(hconn, hod, "INPUT_SHARED", hobj,
           cc, reason);
if cc ^= 0 then do;
  put 'MQOPEN: failed with reason= ' reason;
  goto exit;
end;

call mqgmo(hgmo, "SET", rc, "options, matchoptions",
           "logical_order,complete_msg,all_msgs_available",
           "groupid");
if rc ^= 0 then do;
  put 'MQGMO: failed with rc= ' rc;
  msg = sysmsg();
  put msg;
  goto exit;
end;

call mqmd(hmd, "SET", rc, "groupid", groupid);
if rc ^= 0 then do;
  put 'MQMD: failed with rc= ' rc;
  msg = sysmsg();
  put msg;
  goto exit;
end;
call mqget(hconn, hobj, hmd, hgmo, msglen, cc, reason);
if( cc ) then do;
   put 'MQGET: failed with reason = ' reason;
   msg = sysmsg();
   put msg;
   goto exit;
end;

/* inquire about msg properties */
call mqmd(hmd, "INQ", rc, "MSGTYPE,GROUPID,MSGSEQNUMBER,MSGFLAGS",
   msgtype, groupid, seqno, msgflags);
if( rc ) then do;
   put 'MQMD failed';
   msg = sysmsg();
   put msg;
   goto exit;
end;

put msgtype=
put groupid=
put seqno=
put msgflags=

/* retrieve record from internal buffer */
call mqgetparms(hmap, rc, record);
if( rc ) then do;
   put 'MQGETPARMS: failed';
   msg = sysmsg();
   put msg;
   goto exit;
end;

put 'write record to file';
rc = fput(fileid, record);
if( rc ) then do;
   put 'Error writing to output file buffer...';
   goto exit;
end;

/* flush it to disk */
rc = fwrite(fileid);
if( rc ) then do;
   put 'Error writing to output file...';
   goto exit;
end;

/** receive until last in group **/
if( (msgflags=group;_mask) AND
   (NOT(msgflags=last;_group_mask)) )
   then goto next;
else goto exit;
end;
else goto recv;

exit:
if( hobj ) then do;
call mqclose(hconn, hobj, "NONE", cc, reason);
if( cc ) then do;
   put 'MQCLOSE: failed with reason = ' reason;
   msg = sysmsg();
   put msg;
end;
end;

if( hconn ) then do;
call mqdisc(hconn, cc, reason);
if( cc ) then do;
   put 'MQDISC: failed with reason = ' reason;
   msg = sysmsg();
   put msg;
end;
end;

if( hod ) then
   call mqfree(hod);
if( hmgm ) then
   call mqfree(hmgm);
if( hmd ) then
   call mqfree(hmd);
if( hmap ) then
   call mqfree(hmap);

/* close file */
rc = fclose(fileid);
if( rc ) then put 'Error closing output file';

run;

Processing a Binary File

This example puts a binary file on a queue.

data _null_;
   length rc 8;
   length msg $ 200;
   length hconn hod hmgm hobj hmd hmap hdata 8;
   length cc reason 8;
   length corrid $ 48;
   length msgbuf $ 256;
   length seqno 8 seqstr $ 4;

   /* send this file to the queue */
infile 'd:\test.exe' recfm=f lrecl=1 end=eof;

call mqconn("TESTQMGR", hconn, cc, reason);
if cc ^= 0 then do;
   if reason = 2002 then do;
      put 'Already connected to QMgr';
   end;
   else do;
      if reason = 2059 then
         put 'MQCONN: QMgr not available... needs to be started';
      else
         put 'MQCONN: failed with reason= ' reason;
      goto exit;
   end;
end;

put '---------- Generate object descriptor ----------';
call mqod(hod, "GEN", rc, "OBJECTNAME", "TESTQ");
if rc ^= 0 then do;
   put 'MQOD: failed with rc= ' rc;
   msg = sysmsg();
   put msg;
   goto exit;
end;

put '-------- Open queue object for output ---------';
call mqopen(hconn, hod, "OUTPUT", hobj, cc, reason);
if cc ^= 0 then do;
   put 'MQOPEN: failed with reason= ' reason;
   goto exit;
end;

put '--------- Generate put message options -------';
call mqpmo(hpmo, "GEN", rc);
if rc ^= 0 then do;
   put 'MQPMO: failed with rc= ' rc;
   msg = sysmsg();
   put msg;
   goto exit;
end;

put '--------- Generate message descriptor ---------';
call mqmd(hmd, "GEN", rc, "PERSISTENCE,MSGTYPE",
   "PERSISTENT", 100001);
if rc ^= 0 then do;
   put 'MQMD: failed with rc= ' rc;
   msg = sysmsg();
   put msg;
   goto exit;
end;

put '----------- Generate map descriptor -----------';
/* send 256 byte messages to the queue */
call mqmap(hmap, rc, "char,,256");
if rc ^= 0 then do;
   put 'MQMAP: failed';
   msg = sysmsg();
   put msg;
   goto exit;
end;

/* all of these messages will have the same
   correlationid+seqno */
corrid="42696e46696c65212121"; /* BinFile!!! */

seqno = 0;
i=1;
do until(eof);
   /* read a byte at a time */
   input x $char1.;
   i+1;
   substr(msgbuf,i,1) = x;
   if( i = 256 or.eof ) then do;
      /* set length of this record embedded
         as first byte of message */
      substr(msgbuf,1,1) = put(i-1,pib1.);
      call mqsetparms(hdata, hmap, rc, msgbuf);
      if( rc ) then do;
         put 'MQSETPARMS: failed';
         msg = sysmsg();
         put msg;
         goto exit;
      end;
      /* add sequence # to correlationid */
      segstr = put(seqno, hex4.).
      substr(corrid,21,4) = segstr;
      segno+1;
      call mqmd(hmd, "SET", rc, "CORRELID,MSGID",
         corrid, "");
      if rc ^= 0 then do;
         put 'MQMD: failed with rc= ' rc;
         msg = sysmsg();
         put msg;
         goto exit;
      end;
   end;

   put '--- Put msg on queue ----';
   call mqput(hconn, hobj, hmd, hpmo, hdata,
      cc, reason);
   if cc ^= 0 then do;
      put 'MQPUT: failed with reason= ' reason;
      msg = sysmsg();
      put msg;

go to exit;
end;

/* free data */
call mqfree(hdata);

/* reset message buffer entities */
i=1;
msgbuf="";
end;
end;

exit:
if( hobj ) then do;
call mqclose(hconn, hobj, "NONE", cc, reason);
if( cc ) then do;
   put 'MQCLOSE: failed with reason = ' reason;
   msg = sysmsg();
   put msg;
end;
end;

if( hconn ) then do;
call mqdisc(hconn, cc, reason);
if( cc ) then do;
   put 'MQDISC: failed with reason = ' reason;
   msg = sysmsg();
   put msg;
end;
end;

if hod ^= 0 then do;
call mqfree(hod);
put 'Object descriptor handle freed';
end;
if hpmo ^= 0 then do;
call mqfree(hpmo);
put 'Put message options handle freed';
end;
if hmd ^= 0 then do;
call mqfree(hmd);
put 'Message descriptor handle freed';
end;
if hmap ^= 0 then do;
call mqfree(hmap);
put 'Map descriptor handle freed';
end;

stop;
run;
Getting a Binary File from a Queue

This example gets the first binary file on a queue.

```
filename output 'd:\testdup.exe';

data _null_;
length rc 8;
length msg $ 200;
length cc reason 8;
length hconn hod hgmo hobj hobj2 hmap 8;
length corrid filecorrid $ 48;
length msgbuf stream $ 256;
length len 8;
length seqno 8;

fileid = fopen('output', 'o', 0, 'b');
if( fileid = 0 ) then do;
   put 'Error opening output file...';
   goto exit;
end;

put '---------------- Connect to QMgr --------------';
call mqconn("TESTQMGR", hconn, cc, reason);
if cc ^= 0 then do;
   if reason = 2002 then do;
      put 'Already connected to QMgr';
   end;
   else do;
      if reason = 2059 then
         put 'MQCONN: QMgr not available... needs to be started';
      else
         put 'MQCONN: failed with reason= ' reason;
   goto exit;
end;

put '---------- Generate object descriptor ---------';
call mqod(hod, "GEN", rc, "OBJECTNAME", "TESTQ");
if rc ^= 0 then do;
   put 'MQOD: failed with rc= ' rc;
   msg = sysmsg();
   put msg;
   goto exit;
end;

put '--------- Open queue object for input ---------';
call mqopen(hconn, hod, "INPUT_SHARED,BROWSE", hobj, cc,
```
reason);
if cc ^= 0 then do;
   put 'MQOPEN: failed with reason= ' reason;
   goto exit;
end;

put '--------- Generate get message options --------';
call mqgmo(hgmo, "GEN", rc, "options", "browse_next");
if rc ^= 0 then do;
   put 'MQGMO: failed with rc= ' rc;
   msg = sysmsg();
   put msg;
   goto exit;
end;

put '--------- Generate message descriptor ---------';
call mqmd(hmd, "GEN", rc);
if rc ^= 0 then do;
   put 'MQMD: failed with rc= ' rc;
   msg = sysmsg();
   put msg;
   goto exit;
end;

seqno=0;
recv:
call mqget(hconn, hobj, hmd, hgmo, msglen, cc, reason);
if( cc ) then do;
   if( reason = 2033 ) then do;
      put 'reached end of queue';
      goto exit;
   end;
else do;
   put 'MQGET: failed with reason = ' reason;
   msg = sysmsg();
   put msg;
   goto exit;
end;

/* inquire about msg properties */
call mqmd(hmd, "INQ", rc, "CORRELID,MSGTYPE",
   corrid, msgtype);
if( rc ) then do;
   put 'MQMD failed';
   msg = sysmsg();
   put msg;
   goto exit;
end;

/* default for getting next msg on queue */
call mqgmo(hgmo, "SET", rc, "options", "browse_next");
if rc ^= 0 then do;
   put 'MQGMO: failed with rc= ' rc;
   msg = sysmsg();
   put msg;
   goto exit;
end;

if( msgtype = 100001 ) then do;
   /* file processing... */
   outofseq=0;

   if( filecorrid = "" ) then do;
      /* file begins at this message */
      /* write all correlating messages to this file */
      filecorrid = substr(corrid,1,20);

      put '--------- Generate map descriptor ---------';
      /* all file messages were sent to the queue as 256 bytes blank-padded */
      call mqmap(hmap, rc, "char,,256");
      if( rc ) then do;
         put 'MQMAP: failed';
         msg = sysmsg();
         put msg;
         goto exit;
      end;
   end;
   /* make sure message belongs to this file */
   if( substr(corrid,1,20) = filecorrid ) then do;
      if( seqno ^= input(substr(corrid,21,4), hex4.) ) then do;
         /* this message is out of sequence so search for it */
         outofseq=1;
      end;
      /* open another instance to search for out-of-seq message */
      call mqopen(hconn, hod, "INPUT_SHARED,BROWSE",
                  hobj2, cc, reason);
      if cc ^= 0 then do;
         put 'MQOPEN: failed with reason= ' reason;
         goto exit;
      end;
      corrid = filecorrid;
      substr(corrid,21,4) = put(seqno, hex4.);
      call mqmd(hmd, "SET", rc, "MSGID,CORRELID",
               ",", corrid);
      if( rc ) then do;
         put 'MQMD: failed';
         msg = sysmsg();
         put msg;
      end;
   end;
end;
call mqgmo(hgmo, "SET", rc, "OPTIONS", "BROWSE_FIRST");
if( rc ) then do;
    put 'MQGMO: failed';
    msg = sysmsg();
    put msg;
    goto exit;
end;

call mqget(hconn, hobj2, hmd, hgmo, msglen, cc, reason);
if( cc ) then do;
    if( reason = 2033 ) then do;
        put 'Error: reached end of queue while searching for out-of-sequence msg';
        goto exit;
    end;
    else do;
        put 'MQGET: failed with reason = ' reason;
        msg = sysmsg();
        put msg;
        goto exit;
    end;
end;
end;

/* increment sequence number for next expected message */
seqno+1;

/* retrieve record from internal buffer */
call mqgetparms(hmap, rc, msgbuf);
if( rc ) then do;
    put 'MQGETPARMS: failed';
    msg = sysmsg();
    put msg;
    goto exit;
end;

/* length of this stream is embedded as 1st byte in msg */
len = input(substr(msgbuf,1,1), pib1.);
stream = substr(msgbuf,2);

put 'write stream to file';
rc = fput(fileid, substr(stream,1,len));
if( rc ) then do;
    put 'Error writing to output file buffer...';
goto exit;
end;

/* flush it to disk */
rc = fwrite(fileid);
if( rc ) then do;
put 'Error writing to output file...';
goto exit;
end;

/* now remove it from the queue... */
call mqgmo(hgmo, "SET", rc, "OPTIONS",
          "MSG_UNDER_CURSOR");
if( rc ) then do;
   put 'MQGMO: failed';
   msg = sysmsg();
   put msg;
   goto exit;
end;

if( outofseq ) then do;
call mqget(hconn, hobj2, hmd, hgmo, msglen,
        cc, reason);
if( cc ) then do;
   put 'problems removing message from queue';
   msg = sysmsg();
   put msg;
   goto exit;
end;
end;
/* close queue */
call mqclosel(hconn, hobj2, "NONE", cc, reason);

/* re-read previous message */
call mqgmo(hgmo, "SET", rc, "OPTIONS",
          "BROWSE_MSG_UNDER_CURSOR");
if( rc ) then do;
   put 'MQGMO: failed';
   msg = sysmsg();
   put msg;
   goto exit;
end;
else do;
call mqget(hconn, hobj, hmd, hgmo, msglen,
        cc, reason);
if( cc ) then do;
   put 'problems removing message from queue';
   msg = sysmsg();
   put msg;
   goto exit;
end;

/* browse next message */
call mqgmo(hgmo, "SET", rc, "OPTIONS",
          "BROWSE_NEXT");
if( rc ) then do;
   put 'MQGMO: failed';
   msg = sysmsg();
   put msg;
goto exit;
end;
end;
end;
end;
/* finish retrieving all messages belonging
to this file */
/* reset message descriptor */
call mqmd(hmd, "SET", rc, "MSGID,CORRELID", ",", ");
if( rc ) then do;
  put 'MQMD: failed';
  msg = sysmsg();
  put msg;
  goto exit;
end;
goto recv;
exit:
if( hobj ) then do;
  call mqclose(hconn, hobj, "NONE", cc, reason);
  if( cc ) then do;
    put 'MQCLOSE: failed with reason = ' reason;
    msg = sysmsg();
    put msg;
  end;
end;
if( hconn ) then do;
  call mqdisc(hconn, cc, reason);
  if( cc ) then do;
    put 'MQDISC: failed with reason = ' reason;
    msg = sysmsg();
    put msg;
  end;
end;
if( hod ) then
  call mqfree(hod);
if( hgmo ) then
  call mqfree(hgmo);
if( hmd ) then
  call mqfree(hmd);
if( hmap ) then
  call mqfree(hmap);
/* close file */
rc = fclose(fileid);
if( rc ) then put 'Error closing output file';
run;
Macro Language Coding Examples

This section shows examples of using the SAS Macro Language to make calls to the MQSeries Interface.

```sas
%macro
    putmsg;
    %let hconn=0;
    %let hobj=0;
    %let hod=0;
    %let hpmo=0;
    %let hmd=0;
    %let hmap=0;
    %let hdatal=0;
    %put ---------------- Connect to QMgr --------------;
    %let qmgr=TEST;
    %let cc=0;
    %let reason=0;
    %syscall mqconn(qmgr, hconn, cc, reason);
    %if cc ^= 0 %then %do;
        %if reason = 2002 %then %do;
            %put Already connected to QMgr qmgr;
        %end;
        %else %do;
            %if reason = 2059 %then
                %put MQCONN: QMgr not available...
                %else
                    %put MQCONN: failed with reason= reason;
                    %goto exit;
            %end;
        %end;
    %end;
    %put ---------- Generate object descriptor ---------;
    %let action=GEN;
    %let rc=0;
    %let parms=OBJECTNAME;
    %let objname=TEST;
    %syscall mqod(hod, action, rc, parms, objname);
    %if rc ^= 0 %then %do;
        %put MQOD: failed with rc= rc;
        %put %sysfunc(sysmsg());
        %goto exit;
    %end;
    %else %put MQOD: successfully generated
                  object descriptor;
    %end;
    %put ---------- Open queue object for output ---------;
    %let options=OUTPUT;
    %syscall mqopen(hconn, hod, options, hobj, cc, reason);
    %if cc ^= 0 %then %do;
```

Chapter 3 / Using IBM WebSphere MQ
%put MQOPEN: failed with Reason= reason;
%goto exit;
%end;
%else %put MQOPEN: successfully opened queue for output;

%put ------- Generate put message options -------;
%syscall mqpmo(hpmo, action, rc);
%if rc ^= 0 %then %do;
   %put MQPMO: failed with rc= rc;
   %put %sysfunc(sysmsg());
   %goto exit;
%end;
%else %put MQPMO: successfully generated put message options;

%put ------- Generate message descriptor -------;
%let parms=PERSISTENCE;
%let persist=PERSISTENT;
%syscall mqmd(hmd, action, rc, parms, persist);
%if rc ^= 0 %then %do;
   %put MQMD: failed with rc= rc;
   %put %sysfunc(sysmsg());
   %goto exit;
%end;
%else %put MQMD: successfully generated message descriptor;

%put ------- Generate map descriptor -------;
/* data will not be aligned */
%let desc1=SHORT;
%let desc2=LONG;
%let desc3=DOUBLE;
%let desc4=CHAR,,50;
%syscall mqmap(hmap, rc, desc1, desc2, desc3, desc4);
%if rc ^= 0 %then %do;
   %put MQMAP: failed with rc= rc;
   %put %sysfunc(sysmsg());
   %goto exit;
%end;
%else %put MQMAP: successfully generated map descriptor;

%put --- Generate data descriptor - actual data ----;
%let parm1=100;
%let parm2=9999;
%let parm3=9999.999;
%let parm4=This is a test.;
%syscall mqsetparms(hdata, hmap, rc, parm1, parm2, parm3, parm4);
%if rc ^= 0 %then %do;
   %put MQSETPARMS: failed with rc= rc;
   %put %sysfunc(sysmsg());

\%goto exit;
\%end;
\%else \%put MQSETPARMS: successfully generated data descriptor;

\%put ----------------- Put message on queue -----------------;
\%syscall mqput(hconn, hobj, hmd, hpmo, hdata, cc, reason);
\%if cc; ^= 0 \%then \%do;
   \%put MQPUT: failed with reason= reason;
   \%goto exit;
\%end;
\%else \%do;
   \%put MQPUT: successfully put message on queue;

\/* inquire about message descriptor */
\let action=INQ;
\let parms=MSGID,PUTAPPLTYPE,PUTAPPLNAME, PUTDATE,PUTTIME;
\/* initialize msgid for return length of 48 */
\let msgid="                                       
\let appltype=0;
\/* initialize applname for return length of 28 */
\let applname="                          
\/* initialize data, time for return length of 8 */
\let date="      
\let time="      

\%syscall mqmd(hmd, action, rc, parms, msgid, appltype, applname, date, time);
\%if rc; ^= 0 \%then \%do;
   \%put MQMD: failed with rc=  rc;
   \%put %sysfunc(sysmsg());
\%end;
\%else \%do;
   \%put Message descriptor output parameters are:;
   \%put MSGID=  msgid;
   \%put PUTAPPLTYPE=  appltype;
   \%put PUTAPPLNAME=  applname;
   \%put PUTDATE=  date;
   \%put PUTTIME=  time;
\%end;
\%end;
\%exit:
\%if hobj; ^= 0 \%then \%do;
   \%put ---------------- Close queue -----------------;
   \let options=NONE;
\%syscall mqclose(hconn, hobj, options, cc, reason);
\%if cc; ^= 0 \%then \%do;
      \%put MQCLOSE: failed with reason= reason;
\%end;
\%end;
%else %put MQCLOSE: successfully closed queue;
%end;

%if hconn; ^= 0 %then %do;
 %put ---------------- Disconnect from QMgr ----------------;
 %syscall mqdisc(hconn, cc, reason);
 %if cc; ^= 0 %then %do;
   %put MQDISC: failed with reason= reason;
 %end;
 %else %put MQDISC: successfully disconnected from QMgr;
%end;

%if hod; ^= 0 %then %do;
 %syscall mqfree(hod);
 %put Object descriptor handle freed;
%end;
%if hpmo; ^= 0 %then %do;
 %syscall mqfree(hpmo);
 %put Put message options handle freed;
%end;
%if hmd; ^= 0 %then %do;
 %syscall mqfree(hmd);
 %put Message descriptor handle freed;
%end;
%if hmap; ^= 0 %then %do;
 %syscall mqfree(hmap);
 %put Map descriptor handle freed;
%end;
%if hdata; ^= 0 %then %do;
 %syscall mqfree(hdata);
 %put Data descriptor handle freed;
%end;

%mend putmsg;

/** invoke macro to Put a message on a queue **/ 
%putmsg;

%macro getmsg;
%let hconn=0;
%let hobj=0;
%let hod=0;
%let hgmo=0;
%let hmd=0;
%let hmap=0;
%put ---------------- Connect to QMgr ----------------;
%let qmgr=TEST;
%let cc=0;
%let reason=0;
%syscall mqconn(qmgr, hconn, cc, reason);
%if cc; ^= 0 %then %do;

%if reason; = 2002 %then %do;
  %put Already connected to QMgr qmgr;
%end;
%else %do;
  %if reason; = 2059 %then
    %put MQCONN: QMgr not available...
    needs to be started;
  %else
    %put MQCONN: failed with reason= reason;
    %goto exit;
  %end;
%end;
%else %put MQCONN: successfully connected
to QMgr qmgr;

%put ---------- Generate object descriptor ----------;
%let rc=0;
%let action=GEN;
%let parms=OBJECTNAME;
%let objname=TEST;
%syscall mqod(hod, action, rc, parms, objname);
%if rc; ^= 0 %then %do;
  %put MQOD: failed with rc= rc;
  %put %sysfunc(sysmsg());
  %goto exit;
%end;
%else %put MQOD: successfully generated
    object descriptor;

%put ---------- Open queue object for input ----------;
%let options=INPUT_SHARED;
%syscall mqopen(hconn, hod, options, hobj, cc, reason);
%if cc; ^= 0 %then %do;
  %put MQOPEN: failed with reason= reason;
  %goto exit;
%end;
%else %put MQOPEN: successfully opened queue for output;

%put ---------- Generate get message options --------;
%syscall mggmo(hgmo, action, rc);
%if rc; ^= 0 %then %do;
  %put MQGMO: failed with rc= rc;
  %put %sysfunc(sysmsg());
  %goto exit;
%end;
%else %put MQGMO: successfully generated get
    message options;

%put ---------- Generate message descriptor ----------;
%syscall mqmd(hmd, action, rc);
%if rc; ^= 0 %then %do;
%put MQMD: failed with rc= rc;
%put %sysfunc(sysmsg());
%goto exit;
%end;
%else %put MQMD: successfully generated message descriptor;

%put ----------- Generate map descriptor -----------;
%let desc1=SHORT;
%let desc2=LONG;
%let desc3=DOUBLE;
%let desc4=CHAR,,50;
%syscall mqmap(hmap, rc, desc1, desc2, desc3, desc4);
%if rc; ^= 0 %then %do;
   %put MQMAP: failed with rc= rc;
   %put %sysfunc(sysmsg());
   %goto exit;
%end;
%else %put MQMAP: successfully generated map descriptor;

%put ----------- Get message from queue -----------;
%let msglen=0;
%syscall mqget(hconn, hobj, hmd, hgmo, msglen, cc, reason);
%if cc; ^= 0 %then %do;
   %if reason; = 2033 %then %put No message available;
   %else %put MQGET: failed with reason= reason;
   %goto exit;
%end;
%else %do;
   %put MQGET: successfully retrieved message from queue;
   %put message length= msglen;

   /* inquire about message descriptor output parameters */
   %let action=INQ;
   %let parms=REPORT,MSGTYPE,FEEDBACK,MSGID,CORRELID,
      USERIDENTIFIER,PUTAPPLTYPE,PUTAPPLNAME,PUTDATE,
      PUTTIME;
   /* initialize report for return length of 30 */
   %let report="                            ";
   %let msgtype=0;
   %let feedback=0;
   /* initialize msgid, correlid for return length of 48 */
   %let msgid="                                       ";
   %let correlid="                                    ";
   /* initialize userid for return length of 12 */
   %let userid="          ";
   %let appltype=0;
   /* initialize applname for return length of 28 */
   %let applname="                          ";
   /* initialize data, time for return length of 8 */
%let date="      ";
%let time="      ";

%syscall mqmd(hmd, action, rc, parms, report,
   msgtype, feedback, msgid, correlid, userid,
   appltype, applname, date, time);
%if rc ^= 0 %then %do;
   %put MQMD: failed with rc rc;
   %put %sysfunc(sysmsg());
%end;
%else %do;
   %put Message descriptor output parameters are:;
   %put REPORT= report;
   %put MSGTYPE= msgtype;
   %put FEEDBACK= feedback;
   %put MSGID= msgid;
   %put CORRELID= correlid;
   %put USERIDENTIFIER= userid;
   %put PUTAPPLTYPE= appltype;
   %put PUTAPPLNAME= applname;
   %put PUTDATE= date;
   %put PUTTIME= time;
%end;
%end;
%end;

%if msglen > 0 %then %do;
   /* retrieve SAS variables from GET buffer */
   %let parm1=0;
   %let parm2=0;
   %let parm3=0;
   /* initialize character return value length of 50 */
   %let parm4="                                       ";

   %syscall mqgetparms(hmap, rc, parm1,
      parm2, parm3, parm4);
   %put Display SAS macro variables:;
   %put parm1= parm1;
   %put parm2= parm2;
   %put parm3= parm3;
   %put parm4= parm4;
   %if rc ^= 0 %then %do;
      %put MQGETPARMS: failed with rc= rc;
      %put %sysfunc(sysmsg());
   %end;
%end;
%else %put No data associated with message;

%exit:
%if hobj ^= 0 %then %do;
   %put ------------------ Close queue ----------------;
   %let options=NONE;
   %syscall mqclose(hconn, hobj, options, cc, reason);
   %if cc ^= 0 %then %do;
%put MQCLOSE: failed with reason= reason;
%end;
%else %put MQCLOSE: successfully closed queue;
%end;

%if hconn; ^= 0 %then %do;
  %put --------------- Disconnect from QMgr ---------------;
  %syscall mqdisc(hconn, cc, reason);
  %if cc; ^= 0 %then %do;
    %put MQDISC: failed with reason= reason;
    %end;
  %else %put MQDISC: successfully disconnected from QMgr;
%end;

%if hod; ^= 0 %then %do;
  %syscall mqfree(hod);
  %put Object descriptor handle freed;
%end;
%if hgmo; ^= 0 %then %do;
  %syscall mqfree(hgmo);
  %put Get message options handle freed;
%end;
%if hmd; ^= 0 %then %do;
  %syscall mqfree(hmd);
  %put Message descriptor handle freed;
%end;
%if hmap; ^= 0 %then %do;
  %syscall mqfree(hmap);
  %put Map descriptor handle freed;
%end;

%mend getmsg;

/** invoke macro to Get a message from a queue **/
%getmsg;
# WebSphere MQ CALL Routines

## Overview of MQ Call Routines

The SAS programming interface to MQSeries was designed to be as similar to WebSphere MQI as possible. Where WebSphere MQI requires a structure, the SAS programming interface requires a handle that represents a data structure. Each supported SAS CALL routine is documented in this section.

## Dictionary

<table>
<thead>
<tr>
<th>Routine Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>CALL MQBACK Routine</td>
<td>64</td>
</tr>
<tr>
<td>CALL MQCLOSE Routine</td>
<td>64</td>
</tr>
<tr>
<td>CALL MQCMIT Routine</td>
<td>65</td>
</tr>
<tr>
<td>CALL MQCONN Routine</td>
<td>65</td>
</tr>
<tr>
<td>CALL MQDISC Routine</td>
<td>66</td>
</tr>
<tr>
<td>CALL MQFREE Routine</td>
<td>66</td>
</tr>
<tr>
<td>CALL MQGET Routine</td>
<td>67</td>
</tr>
<tr>
<td>CALL MQGETPARMS Routine</td>
<td>68</td>
</tr>
<tr>
<td>CALL MQGMO Routine</td>
<td>69</td>
</tr>
<tr>
<td>CALL MQINQ Routine</td>
<td>73</td>
</tr>
<tr>
<td>CALL MQMAP Routine</td>
<td>81</td>
</tr>
<tr>
<td>CALL MQMD Routine</td>
<td>82</td>
</tr>
<tr>
<td>CALL MQOD Routine</td>
<td>84</td>
</tr>
<tr>
<td>CALL MQOPEN Routine</td>
<td>86</td>
</tr>
<tr>
<td>CALL MQPMO Routine</td>
<td>88</td>
</tr>
<tr>
<td>CALL MQPUT Routine</td>
<td>91</td>
</tr>
<tr>
<td>CALL MQPUT1 Routine</td>
<td>92</td>
</tr>
<tr>
<td>CALL MQRMH Routine</td>
<td>94</td>
</tr>
<tr>
<td>CALL MQSET Routine</td>
<td>98</td>
</tr>
<tr>
<td>CALL MQSETPARMS Routine</td>
<td>100</td>
</tr>
</tbody>
</table>

The SAS programming interface to MQSeries was designed to be as similar to WebSphere MQI as possible. Where WebSphere MQI requires a structure, the SAS programming interface requires a handle that represents a data structure. Each supported SAS CALL routine is documented in this section.
CALL MQBACK Routine

Backs out all WebSphere MQ message puts and gets since the last synchpoint.

Syntax

CALL MQBACK(hConn, compCode, reason);

Details

For the complete syntax information, see WebSphere MQ Application Programming Reference at www.ibm.com.

Example

This example reverts the messages in a queue back to the last synchronization point.

compCode=0;
reason=0;
CALL MQBACK(hConn, compCode, reason);

CALL MQCLOSE Routine

Relinquishes access to a WebSphere MQ object (queue, process definition, queue manager).

Syntax

CALL MQCLOSE(hConn, hObj, options, compCode, reason);

Details

For the complete syntax information, see WebSphere MQ Application Programming Reference at www.ibm.com.

Example

This example closes a queue.

options="NONE";
CALL MQCMIT Routine
Commits all WebSphere MQ message puts and gets since the last synchpoint.

Syntax
CALL MQCMIT(hConn, compCode, reason);

Details
For the complete syntax information, see WebSphere MQ Application Programming Reference at www.ibm.com.

Example
This example commits a unit of work.
compCode=0;
reason=0;
CALL MQCMIT(hConn, compCode, reason);

CALL MQCONN Routine
Connects Base SAS to a WebSphere MQ queue manager.

Syntax
CALL MQCONN(name, hConn, compCode, reason);

Details
For the complete syntax information, see WebSphere MQ Application Programming Reference at www.ibm.com.

Example
The following example connects the Base SAS session to the queue manager named TEST.
hConn=0;
CALL MQDISC Routine

Breaks the connection between a WebSphere MQ queue manager and Base SAS.

Syntax

CALL MQDISC(hConn, compCode, reason);

Details

For the complete syntax information, see WebSphere MQ Application Programming Reference at www.ibm.com.

Example

The following example disconnects the Base SAS session from a queue manager identified by the parameter hConn.

compCode=0;
reason=0;
CALL MQDISC(hConn, compCode, reason);

CALL MQFREE Routine

Frees a Base SAS internal handle, thereby releasing its resources.

Syntax

CALL MQFREE(handle);

Required Argument

handle

Specifies the Base SAS internal handle that is obtained from one of the following previous CALL routines:

- CALL MQPMO routine (hpmo)
- CALL MQGMO routine (hgmo)
- CALL MQOD routine (hod)
CALL MQGET Routine

Retrieves a message from a local WebSphere MQ queue that has been previously opened.

Syntax

CALL MQGET(hConn, hObj, hmd, hgmo, msglen, compCode, reason);

Required Arguments

hConn
Specifies the WebSphere MQ connection handle that is obtained from a previous CALL MQCONN routine.

Type: Numeric, Input

hObj
Specifies the WebSphere MQ handle to an open object that is obtained from a previous CALL MQOPEN routine.

Type: Numeric, Input

hmd
Specifies the Base SAS internal message descriptor handle that is obtained from a previous CALL MQMD routine.

Type: Numeric, Input

hgmo
Specifies the Base SAS internal get message options handle that is obtained from a previous CALL MQGMO routine.

Type: Numeric, Input

msglen
Returns the length of the received message. A length of zero signifies a message with no data. In that case, there is no need to use the CALL MQGETPARMS routine.
**compCode**
Returns the WebSphere MQ completion code. This parameter can be used to determine whether an error occurred during the execution of this routine. If an error occurred, then the compCode parameter will be nonzero, and the reason parameter will be set to the appropriate reason code.

**reason**
Returns the WebSphere MQ reason code that qualifies the completion code.

**Note:** A reason code of -1 reflects a Base SAS internal error, not a WebSphere MQ error. To obtain a textual description of a failure (either Base SAS or WebSphere MQ), use the SYSMSG() Base SAS function call.

**Details**
If data accompanies the message, it is retrieved into an internal Base SAS buffer. After the CALL MQGET routine completes, you should use the CALL MQGETPARMS routine to set Base SAS variables (parms) to that data or to retrieve the data into a physical binary or text file.

**Example**
This example gets a message from a queue.

```sas
msglen=0;
compCode=0;
reason=0;
CALL MQGET(hConn, hObj, hmd, hgmo, msglen, 
          compCode, 
          reason);
```

**CALL MQGETPARMS Routine**
Retrieves values of Base SAS variables from a previous WebSphere MQ message that was received by the CALL MQGET routine.

**Syntax**

```sas
CALL MQGETPARMS(hMap, rc, parm1<, parm2, parm3,.....>);
```
Required Arguments

**hMap**
- Specifies a handle to a Base SAS internal map descriptor that is obtained from a previous CALL MQMAP routine.
  - Type: Numeric, Input

**rc**
- Provides the Base SAS return code from this function. If an error occurs, then the return code is nonzero. You can use the Base SAS function SYSMSG() to obtain a textual description of the return code.
  - Type: Numeric, Output

**parm**
- Returns the Base SAS variables.
  - Type: Numeric or Character, Output

Details

This message is available until the next CALL MQGET routine is performed.

**Note:** Initialize variables appropriately to guarantee that truncation does not occur.

Example

This example gets values of Base SAS variables from a received message.

```sas
length parm1 parm2 parm3;
length parm4 $ 200;
rc=0;
CALL MQGETPARMS(hMap, rc, parm1, parm2, parm3, parm4);
```

CALL MQGMO Routine

Manipulates WebSphere MQ get message options to be used on a subsequent CALL MQGET routine.

Syntax

```sas
CALL MQGMO(hgmo, action, rc<, parms, value1, value2, ...>);
```

Required Arguments

**hgmo**
- On input, it specifies a Base SAS internal get message options handle. The handle should be supplied when you are setting or querying an option. The handle is generated as output when *action* is to generate default WebSphere MQ get options.
**action**

Specifies the desired action of this routine. The following action values are valid:

- **GEN**: Generate a handle representing default get message options as defined by WebSphere MQ.
- **SET**: After a get message options handle has been generated, you can continue to set values as necessary.
- **INQ**: After a get message options handle has been generated, you can query its values.

**rc**

Provides the Base SAS return code from this function. If an error occurs, then the return code is nonzero. You can use the Base SAS function SYSMSG() to obtain a textual description of the return code.

**Optional Arguments**

**parms**

Specifies an optional string of get message options that you want to set for subsequent CALL MQGET routines. Each option must be separated by a comma and must have a value associated with it in the function's parameter list.

**value**

Provides the value for a get message option specified in the parms string. You must provide a value parameter for each option specified in the parms string and the data type must be of the proper type. Variables used to store character values being returned in an inquiry (INQ action) should be initialized appropriately to guarantee that truncation of a returned value does not occur.

**Optional Parameters**

The following get message options (parms) and values are valid:

**Options**

Specifies a string of the attributes (options) to associate with subsequent CALL MQGET routines. Each option must be separated by a comma.

The following OPTIONS values are valid:

- **NONE**: Used to unset previously set OPTIONS
- **NO_WAIT** (default): Return immediately if no suitable message
WAIT
  Wait for message to arrive

SYNCPOINT
  Get message with synchpoint control

NO_SYNCPOINT
  Get message without synchpoint control

BROWSE_FIRST
  Browse from start of queue

BROWSE_NEXT
  Browse from current position in queue

MSG_UNDER_CURSOR
  Get message under browse cursor

LOCK
  Lock message
  This option is not supported on z/OS.

UNLOCK
  Unlock message
  This option is not supported on z/OS.

BROWSE_MSG_UNDER_CURSOR
  Browse message under browse cursor
  This option is not supported on z/OS.

FAIL_IF_QUIESCING
  Fail if QMgr is quiescing

CONVERT
  Convert message data
  Type Character, Input
  Notes ACCEPT_TRUNCATED_MSG is not allowed since Base SAS internally maintains resizing of the internal GET buffer to handle any message size.
  Specify CONVERT to allow WebSphere MQ to perform data conversion based on the FORMAT of a PUT message via a conversion exit routine that has been previously established at the QMgr. To allow Base SAS to perform the data conversion instead of using a WebSphere MQ conversion exit routine, do not specify the CONVERT option.

WAITINTERVAL
  Amount of time to wait for message to arrive in milliseconds.
  Type Numeric, Input

RESOLVEDQNAME
  Resolved name of destination queue.
  Type Character48, Output

SASQSID
  A value equal to the environment variable of the same name, a 36–character string. The environment variable can be retrieved using the following in a SAS DATA step:
sid = sysget('SASQSID');

For queues that are monitored by the object spawner, the MsgDeliverySequence property must be set to *Priority*. For more information about this option, see “Configure Your Programs for Message Queue Polling” on page 268.

**Optional Parameters for WebSphere MQ 5.1 and Later**

**MATCHOPTIONS**
Character string of match options that is used to control selection criteria that are associated with subsequent CALL MQGET routines. Each option must be separated by a comma. The following MATCHOPTIONS values are valid:

- **NONE**
  - No matches
- **MSGID**
  - Retrieve message with specified message identifier
- **CORRELID**
  - Retrieve message with specified correlation identifier
- **GROUPID**
  - Retrieve message with specified group identifier
  - This option is not supported on z/OS.
- **SEQNUMBER**
  - Retrieve message with specified sequence number
  - This option is not supported on z/OS.
- **OFFSET**
  - Retrieve message with specified offset
  - This option is not supported on z/OS.

**GROUPSTATUS**
Flag indicating whether message was retrieved within a group

**SEGMENTSTATUS**
Flag indicating whether message was retrieved within a segment of a logical message

**SEGMENTATION**
Flag indicating whether further segmentation is allowed for the retrieved message
Example

This example generates get message options to wait 3 seconds for a GET message operation.

```c
hgmo=0;
action="GEN";
rc=0;
parms="OPTIONS,WAITINTERVAL";
options="WAIT";
interval=3000;
CALL MQGMO(hgmo, action, rc, parms, options, interval);
```

CALL MQINQ Routine

Queries the attributes of a WebSphere MQ object (queue, process definition, queue manager).

Syntax

```c
CALL MQINQ(hConn, hObj, compCode, reason, parms, value1<, value2,...>);
```

Required Arguments

- **hConn**
  Specifies the WebSphere MQ connection handle that is obtained from a previous CALL MQCONN routine.
  - Type: Numeric, Input

- **hObj**
  Specifies the WebSphere MQ Object handle that is obtained from a previous CALL MQOPEN routine that specified the INQUIRE option. This handle can represent a queue, process definition, or queue manager object.
  - Type: Numeric, Input

- **compCode**
  Returns the WebSphere MQ completion code. This parameter can be used to determine whether an error occurred during the execution of this routine. If an error occurred, then the `compCode` parameter will be nonzero, and the `reason` parameter will be set to the appropriate reason code.
  - Type: Numeric, Output

- **reason**
  Returns the WebSphere MQ reason code that qualifies the completion code.

Note: A reason code of -1 reflects a Base SAS internal error, not a WebSphere MQ error. To obtain a textual description of a failure (either Base SAS or WebSphere MQ), use the SYMSG() Base SAS function call.
**parms**
Specifies a string of attributes that you want to query from the WebSphere MQ object. Each object attribute is separated by a comma. The value that is associated with each attribute is returned in a `value` parameter. Not all attributes are valid for each type of object (queue, process definition, or queue manager). Valid object types are listed under each attribute.

**value**
Returns the value for an attribute specified in the `parms` string. You must provide a `value` parameter for each attribute specified `parms` string. Variables used to store character values should be initialized appropriately to guarantee that truncation of a returned value does not occur.

The attributes in the following three tables are valid.

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Data Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ALTERATION_DATE</td>
<td>Character12</td>
<td>Date definition last changed</td>
</tr>
<tr>
<td>ALTERATION_TIME</td>
<td>Character8</td>
<td>Time definition was last changed</td>
</tr>
<tr>
<td>BACKOUT_REQ_Q_NAME</td>
<td>Character48</td>
<td>Excessive backout requeue name</td>
</tr>
<tr>
<td>BACKOUT_THRESHOLD</td>
<td>Numeric</td>
<td>Backout threshold</td>
</tr>
<tr>
<td>BASE_Q_NAME</td>
<td>Character48</td>
<td>Name of queue to which alias resolves</td>
</tr>
<tr>
<td>CF_STRUC_NAME</td>
<td>Character12</td>
<td>Coupling-facility structure name (z/OS only)</td>
</tr>
<tr>
<td>CLUSTER_NAME</td>
<td>Character48</td>
<td>Name of cluster to which queue belongs</td>
</tr>
<tr>
<td>CLUSTER_NAMELIST</td>
<td>Character48</td>
<td>Name of namelist containing names of clusters to which queue belongs</td>
</tr>
<tr>
<td>CLUSTER_WORKLOAD_DATA</td>
<td>Character32</td>
<td>User data for cluster workload exit</td>
</tr>
<tr>
<td>CLUSTER_WORKLOAD_LENGTH</td>
<td>Numeric</td>
<td>Maximum length of message data passed to cluster workload exit</td>
</tr>
<tr>
<td>CREATION_DATE</td>
<td>Character12</td>
<td>Queue creation date</td>
</tr>
<tr>
<td>Attribute</td>
<td>Data Type</td>
<td>Description</td>
</tr>
<tr>
<td>--------------------------</td>
<td>-------------</td>
<td>--------------------------------------------------</td>
</tr>
<tr>
<td>CREATION_TIME</td>
<td>Character8</td>
<td>Queue creation time</td>
</tr>
<tr>
<td>CURRENT_Q_DEPTH</td>
<td>Numeric</td>
<td>Number of messages on queue</td>
</tr>
<tr>
<td>DEF_BIND</td>
<td>Numeric</td>
<td>Default binding</td>
</tr>
<tr>
<td>DEF_INPUT_OPEN_OPTION</td>
<td>Numeric</td>
<td>Default open-for-input option</td>
</tr>
<tr>
<td>DEF_PERSISTENCE</td>
<td>Numeric</td>
<td>Default message persistence</td>
</tr>
<tr>
<td>DEF_PRIORITY</td>
<td>Numeric</td>
<td>Default message priority</td>
</tr>
<tr>
<td>DEF_XMIT_Q_NAME</td>
<td>Character48</td>
<td>Default transmission queue name</td>
</tr>
<tr>
<td>DEFINITION_TYPE</td>
<td>Numeric</td>
<td>Queue definition type</td>
</tr>
<tr>
<td>EXPIRY_INTERVAL</td>
<td>Numeric</td>
<td>Interval between scans for expired messages (z/OS only)</td>
</tr>
<tr>
<td>HARDEN_GET_BACKOUT</td>
<td>Numeric</td>
<td>Whether to harden backout count</td>
</tr>
<tr>
<td>IGQ_PUT_AUTHORITY</td>
<td>Numeric</td>
<td>Intra-group queuing put authority (z/OS only)</td>
</tr>
<tr>
<td>IGQ_USER_ID</td>
<td>Character12</td>
<td>Intra-group queuing agent user ID (z/OS only)</td>
</tr>
<tr>
<td>INDEX_TYPE</td>
<td>Numeric</td>
<td>Index type (z/OS only)</td>
</tr>
<tr>
<td>INHIBIT_GET</td>
<td>Numeric</td>
<td>Whether get operations are allowed</td>
</tr>
<tr>
<td></td>
<td></td>
<td>This output type is not supported on z/OS.</td>
</tr>
<tr>
<td>INHIBIT_PUT</td>
<td>Numeric</td>
<td>Whether put operations are allowed</td>
</tr>
<tr>
<td>INITIATION_Q_NAME</td>
<td>Character48</td>
<td>Initiation queue name</td>
</tr>
<tr>
<td>INTRA_GROUP_QUEUING</td>
<td>Numeric</td>
<td>Intra-group queuing support (z/OS only)</td>
</tr>
<tr>
<td>MAX_MSG_LENGTH</td>
<td>Numeric</td>
<td>Maximum message length</td>
</tr>
<tr>
<td>Attribute</td>
<td>Data Type</td>
<td>Description</td>
</tr>
<tr>
<td>--------------------------</td>
<td>-------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>MAX_Q_DEPTH</td>
<td>Numeric</td>
<td>Maximum number of messages allowed on queue</td>
</tr>
<tr>
<td>MSG_DELIVERY_SEQUENCE</td>
<td>Numeric</td>
<td>Whether message priority is relevant</td>
</tr>
<tr>
<td>OPEN_INPUT_COUNT</td>
<td>Numeric</td>
<td>Number of CALL MQOPEN routines that have the queue open for input</td>
</tr>
<tr>
<td>OPEN_OUTPUT_COUNT</td>
<td>Numeric</td>
<td>Number of CALL MQOPEN routines that have the queue open for output</td>
</tr>
<tr>
<td>PROCESS_NAME</td>
<td>Character32</td>
<td>Name of process definition</td>
</tr>
<tr>
<td>Q_DEPTH_HIGH_EVENT</td>
<td>Numeric</td>
<td>Control attribute for queue depth high events</td>
</tr>
<tr>
<td></td>
<td></td>
<td>This output type is not supported on z/OS</td>
</tr>
<tr>
<td>Q_DEPTH_HIGH_LIMIT</td>
<td>Numeric</td>
<td>High limit for queue depth</td>
</tr>
<tr>
<td></td>
<td></td>
<td>This output type is not supported on z/OS</td>
</tr>
<tr>
<td>Q_DEPTH_LOW_EVENT</td>
<td>Numeric</td>
<td>Control attribute for queue depth low events</td>
</tr>
<tr>
<td></td>
<td></td>
<td>This output type is not supported on z/OS</td>
</tr>
<tr>
<td>Q_DEPTH_LOW_LIMIT</td>
<td>Numeric</td>
<td>Low limit for queue depth</td>
</tr>
<tr>
<td></td>
<td></td>
<td>This output type is not supported on z/OS</td>
</tr>
<tr>
<td>Q_DEPTH_MAX_EVENT</td>
<td>Numeric</td>
<td>Control attribute for queue depth max events</td>
</tr>
<tr>
<td></td>
<td></td>
<td>This output type is not supported on z/OS</td>
</tr>
<tr>
<td>Q_DESC</td>
<td>Character64</td>
<td>Queue description</td>
</tr>
<tr>
<td>Q_NAME</td>
<td>Character48</td>
<td>Queue name</td>
</tr>
<tr>
<td>Q_SERVICE_INTERVAL</td>
<td>Numeric</td>
<td>Limit for queue service interval</td>
</tr>
<tr>
<td></td>
<td></td>
<td>This output type is not supported on z/OS</td>
</tr>
<tr>
<td>Attribute</td>
<td>Data Type</td>
<td>Description</td>
</tr>
<tr>
<td>-----------------------------------</td>
<td>-----------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>Q_SERVICE_INTERVAL_EVENT</td>
<td>Numeric</td>
<td>Control for queue service interval events</td>
</tr>
<tr>
<td></td>
<td></td>
<td>This output type is not supported on z/OS.</td>
</tr>
<tr>
<td>Q_TYPE</td>
<td>Numeric</td>
<td>Queue type</td>
</tr>
<tr>
<td>QSG_DISP</td>
<td>Numeric</td>
<td>Queue-sharing group disposition (z/OS only)</td>
</tr>
<tr>
<td>QSG_NAME</td>
<td>Character4</td>
<td>Name of queue-sharing group (z/OS only)</td>
</tr>
<tr>
<td>REMOTE_Q_MGR_NAME</td>
<td>Character4</td>
<td>Name of remote queue manager</td>
</tr>
<tr>
<td>REMOTE_Q_NAME</td>
<td>Character4</td>
<td>Name of remote queue as known on remote queue manager</td>
</tr>
<tr>
<td>RETENTION_INTERVAL</td>
<td>Numeric</td>
<td>Queue retention interval</td>
</tr>
<tr>
<td>SCOPE</td>
<td>Numeric</td>
<td>Queue definition scope</td>
</tr>
<tr>
<td></td>
<td></td>
<td>This output type is not supported on z/OS.</td>
</tr>
<tr>
<td>SHAREABILITY</td>
<td>Numeric</td>
<td>Whether queue can be shared</td>
</tr>
<tr>
<td>STORAGE_CLASS</td>
<td>Character8</td>
<td>Storage class for queue (z/OS only)</td>
</tr>
<tr>
<td>TRIGGER_CONTROL</td>
<td>Numeric</td>
<td>Trigger control</td>
</tr>
<tr>
<td>TRIGGER_DATA</td>
<td>Character64</td>
<td>Trigger data</td>
</tr>
<tr>
<td>TRIGGER DEPTH</td>
<td>Numeric</td>
<td>Trigger depth</td>
</tr>
<tr>
<td>TRIGGER_MSG_PRIORITY</td>
<td>Numeric</td>
<td>Threshold message priority for triggers</td>
</tr>
<tr>
<td>TRIGGER_TYPE</td>
<td>Numeric</td>
<td>Trigger type</td>
</tr>
<tr>
<td>USAGE</td>
<td>Numeric</td>
<td>Usage</td>
</tr>
<tr>
<td>XMIT_Q_NAME</td>
<td>Character48</td>
<td>Default transmission queue name</td>
</tr>
</tbody>
</table>
### Table 4.2 Attributes for Queue Managers

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Data Type</th>
<th>Description</th>
</tr>
</thead>
</table>
| ACTIVE_CHANNELS                               | Numeric      | Maximum number of active channels  
|                                               |              | This output type is supported on z/OS only.                                 |
| AUTHORITY_EVENT                               | Numeric      | Control attribute for authority events 
|                                               |              | This output type is not supported on z/OS.                                 |
| CLUSTER_WORKLOAD_EXIT                        | Character    | Name of user exit for cluster workload management                           |
| (MQ_EXIT_NAME_LENGTH)                         | Variable Length |                                                 |
| CODED_CHAR_SET_ID                             | Numeric      | Coded character set identifier                                              |
| COMMAND_INPUT_Q_NAME                         | Character48  | System command input queue name                                              |
| COMMAND_LEVEL                                | Numeric      | Command level supported by queue manager                                    |
| DEAD_LETTER_Q_NAME                           | Character48  | Dead letter queue name                                                       |
| INHIBIT_EVENT                                | Numeric      | Control attribute for inhibit events  
<p>|                                               |              | This output type is not supported on z/OS.                                 |
| LOCAL_EVENT                                  | Numeric      | Control attribute for local events                                          |
|                                               |              | This output type is not supported on z/OS.                                 |
| MAX_CHANNELS                                 | Numeric      | Maximum number of current channels                                           |
|                                               |              | This output type is supported on z/OS only.                                 |
| MAX_HANDLES                                  | Numeric      | Maximum number of handles                                                    |
| MAX_MSG_LENGTH                               | Numeric      | Maximum message length                                                       |
| MAX_PRIORITY                                 | Numeric      | Maximum priority                                                             |</p>
<table>
<thead>
<tr>
<th>Attribute</th>
<th>Data Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>MAX_UNCOMMITTED_MS</td>
<td>Numeric</td>
<td>Maximum number of uncommitted messages within a unit of work&lt;br&gt;This output type is not supported on z/OS.</td>
</tr>
<tr>
<td>PERFORMANCE_EVENT</td>
<td>Numeric</td>
<td>Control attribute for performance events&lt;br&gt;This output type is not supported on z/OS.</td>
</tr>
<tr>
<td>PLATFORM</td>
<td>Numeric</td>
<td>Platform on which the queue manager resides</td>
</tr>
<tr>
<td>Q_MGR_DESC</td>
<td>Character64</td>
<td>Queue manager description</td>
</tr>
<tr>
<td>Q_MGR_IDENTIFIER</td>
<td>Character48</td>
<td>Unique internally generated identifier of queue manager</td>
</tr>
<tr>
<td>Q_MGR_NAME</td>
<td>Character48</td>
<td>Queue manager name</td>
</tr>
<tr>
<td>REMOTE_EVENT</td>
<td>Numeric</td>
<td>(Queue Manager)&lt;br&gt;Control attribute for remote events&lt;br&gt;This output type is not supported on z/OS.</td>
</tr>
<tr>
<td>REPOSITORY_NAME</td>
<td>Character48</td>
<td>Name of cluster for which this queue manager provides repository services</td>
</tr>
<tr>
<td>REPOSITORY_NAMELIST</td>
<td>Character48</td>
<td>Name of namelist object containing names of clusters for which this queue manager provides repository services</td>
</tr>
<tr>
<td>SYNCPOINT</td>
<td>Numeric</td>
<td>(Queue Manager)&lt;br&gt;Synchpoint availability</td>
</tr>
<tr>
<td>START_STOP_EVENT</td>
<td>Numeric</td>
<td>(Queue Manager)&lt;br&gt;Control attribute for start stop events&lt;br&gt;This output type is not supported on z/OS.</td>
</tr>
<tr>
<td>Attribute</td>
<td>Data Type</td>
<td>Description</td>
</tr>
<tr>
<td>-----------------</td>
<td>-----------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>TCP_CHANNELS</td>
<td>Numeric</td>
<td>Maximum number of current channels and TCP/IP client connections</td>
</tr>
<tr>
<td></td>
<td></td>
<td>This output type is supported on z/OS only.</td>
</tr>
<tr>
<td>TRIGGER_INTERVAL</td>
<td>Numeric</td>
<td>(Queue Manager) Trigger interval</td>
</tr>
</tbody>
</table>

Table 4.3  Attributes for Process Definitions

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Data Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>APPL_ID</td>
<td>Character256</td>
<td>Application identifier</td>
</tr>
<tr>
<td>APPL_TYPE</td>
<td>Numeric</td>
<td>(Process Definition) Application type</td>
</tr>
<tr>
<td>ENV_DATA</td>
<td>Character128</td>
<td>Environment data</td>
</tr>
<tr>
<td>PROCESS_DESC</td>
<td>Character48</td>
<td>Description of process definition</td>
</tr>
<tr>
<td>PROCESS_NAME</td>
<td>Character32</td>
<td>Name of process definition</td>
</tr>
<tr>
<td>USER_DATA</td>
<td>Character128</td>
<td>User data</td>
</tr>
</tbody>
</table>

Type  Character or Numeric, Output

Example

This example queries about a queue's maximum depth and the maximum message length.

```
length parms $ 30;
compCode=0;
reason=0;
parms="MAX_Q_DEPTH,MAX_MSG_LENGTH";
CALL MQINQ(hConn, hObj, compCode, reason, parms, maxdepth, maxmsgl);
```
CALL MQMAP Routine

Defines a data map that can be subsequently used on the CALL MQSETPARMS or CALL MQGETPARMS routine.

Syntax

CALL MQMAP(hMap, rc, desc1<, desc2, desc3,…>);

Required Arguments

**hMap**

Returns a Base SAS internal map descriptor handle. The handle generated will be used to reference the data map when setting or getting Base SAS variables in a message.

Type Numeric, Output

**rc**

Provides the Base SAS return code from this function. If an error occurs, then the return code is nonzero. You can use the Base SAS function SYSMSG() to obtain a textual description of the return code.

Type Numeric, Output

**desc**

Specifies a data map descriptor that defines the data type, data offset from the beginning of the message, and data length. A descriptor has the following format:

"TYPE<,OFFSET,LENGTH>"

**TYPE** can be one of the following values:

- CHAR (character data)
- SHORT (short integer)
- LONG (long integer)
- DOUBLE (double precision floating point)

**OFFSET** is the offset from beginning of the message. This property is optional so that by default data is not aligned (data starts at next available position in message).

**LENGTH** is the length of the data being represented. This property is optional in most cases. The only time length is required is when setting up to receive character data. Specifying length for numeric data is ignored since length is implicitly defined.

Type Character, Input

**Note**

Type coercion is performed transparently when you put Base SAS variables into a WebSphere MQ message (CALL MQSETPARMS routine) and also when you get Base SAS variables from a WebSphere MQ.
message (CALL MQGETPARMS routine). That is, if the data that you are sending or receiving is of a different type than the Base SAS variable itself, the data will be coerced into the appropriate data type.

Example

This example defines a map to use to send and receive a message with a short, a long, a double, and a character string. No alignment is specified for any data type. For the character string, a length of 250 is specified.

```
hMap=0;
rc=0;
desc1="SHORT";
desc2="LONG";
desc3="DOUBLE";
desc4="CHAR,,250";
CALL MQMAP(hMap, rc, desc1, desc2, desc3, desc4);
```

CALL MQMD Routine

Manipulates message descriptor parameters to be used on a subsequent CALL MQPUT, CALL MQPUT1, or CALL MQGET routine.

Syntax

```
CALL MQMD(hmd, action, rc<, parms, value1, value2,...>);
```

Required Arguments

**hmd**
On input, specifies a Base SAS internal message descriptor handle. The handle should be supplied when you are setting or querying a value. The handle is generated as output when **action** is to generate default "message descriptor" parameters.

Type: Numeric, Input or Output

**action**
Specifies the desired action of this routine. The following **action** values are valid:

- **GEN**
  Generate a handle that represents default message descriptor parameters as defined by WebSphere MQ.

- **SET**
  After a message descriptor handle has been generated, you can continue to set values as necessary.

- **INQ**
  After a message descriptor handle has been generated, you can query its values.
Type Character, Input

rc
Provides the Base SAS return code from this function. If an error occurs, then the return code is nonzero. You can use the Base SAS function SYSMSG() to obtain a textual description of the return code.

Type Numeric, Output

Optional Arguments

parms
Specifies an optional string of message descriptor parameters that you want to set for subsequent CALL MQPUT, CALL MQPUT1, or CALL MQGET routines. Each parameter must be separated by a comma and must have a value associated with it in the function's parameter list.

For a list of the parameters that you can specify, see the documentation for the CALL MQMD routine structure in the WebSphere MQ Application Programming Reference at www.ibm.com.

Type Character, Input

value
Provides a value for a message descriptor parameter specified in the parms string. You must provide a value parameter for each message descriptor parameter specified in the parms string and the data type must be of the proper type. Variables used to store character values that are being returned in an inquiry (INQ action) should be initialized appropriately to guarantee that truncation of a returned value does not occur.

Type Character or Numeric, Input or Output

Note This routine supports both sending a message (CALL MQPUT and CALL MQPUT1 routines) and receiving a message (CALL MQGET routine). Therefore, the parameters and values serve as both input and as output to the function.

Details

ENCODING and CODEDCHARSETID should not be set in most situations since you want a message to be described by its native numeric and character encoding, which are the default attributes for these parms.

FORMAT should be set if you intend for a WebSphere MQ QMgr conversion exit to be invoked when an application GETs a message. The FORMAT name is the actual name of the conversion exit that is invoked when an application GETs a message with the CONVERT get message option specified. The FORMAT name in the message descriptor is set when a message is PUT on a queue. Refer to WebSphere MQ literature for details about creating a conversion exit.

MSGID and CORRELID are updated on PUTs and GETs, so remember to reset their values appropriately when performing multiple PUTs or GETs with the same message descriptor.
Example

This example sends a message to a queue, and then queries and displays the message descriptor values.

```plaintext
length parms $ 57;
length report $ 30 msgtype $ 8 msgid $ 48 correlid $48
applname $ 28 putdate $ 8 puttime $ 8;
/* generate a message descriptor to PUT a persistent */
/* message on a permanent queue */
hmd=0;
action="GEN";
rc=0;
parms="PERSISTENCE"
persist="PERSISTENT";
CALL MQMD(hmd, action, rc, parms, persist);
/* inquire about message descriptor values after GET */
/* operation completes successfully */
action="INQ";
parms="REPORT,MSGTYPE,MSGID,CORRELID,
PUTAPPLNAME,PUTDATE,PUTTIME";
CALL MQMD(hmd, action, rc, parms, report, msgtype,
msgid, correlid, applname, putdate, puttime);
put 'report type is ' report;
put 'message type is ' msgtype;
put 'message id is ' msgid;
put 'correlation id is ' correlid;
put 'put application name is ' applname;
put 'put date is ' putdate;
put 'put time is '
puttime;
```

CALL MQOD Routine

Manipulates object descriptor parameters to be used on a subsequent CALL MQOPEN or CALL MQPUT1 routine.

Syntax

```plaintext
CALL MQOD(hod, action, rc<, parms, value1, value2,...>);
```

Required Arguments

**hod**

On input, it specifies a Base SAS internal object descriptor handle. The handle should be supplied when you are setting or querying a value. The handle is generated as output when `action` is to generate default object descriptor parameters.

Type: Numeric, Input or Output

**action**

Specifies the desired action of this routine. The following `action` values are valid:
GEN
Generate a handle representing default object descriptor parameters as defined by WebSphere MQ.

SET
After an object descriptor handle has been generated, you can continue to set values as necessary.

INQ
After an object descriptor handle has been generated, you can query its values.

**Type**  Character, Input

**rc**
Provides the Base SAS return code from this function. If an error occurs, then the return code is nonzero. You can use the Base SAS function SYSMSG() to obtain a textual description of the return code.

**Type**  Numeric, Output

Optional Arguments

**parms**
Specifies an optional string of object descriptor parameters that you want to set for subsequent CALL MQOPEN or CALL MQPUT1 routines. Each parameter must be separated by a comma and must have a value associated with it in the function's parameter list.

For a list of the parameters that you can specify, see the documentation for the CALL MQOD routine structure in the WebSphere MQ Application Programming Reference at www.ibm.com.

**Type**  Character, Input

**value**
Provides a value for an object descriptor parameter specified in the *parms* string. You must provide a value parameter for each object descriptor parameter specified in the *parms* string and the data type must be of the proper type. Variables used to store character values being returned in an inquiry (INQ action) should be initialized appropriately to guarantee that truncation of a returned value does not occur.

**Type**  Character or Numeric, Input or Output

Example

This example generates an object descriptor to OPEN a temporary dynamic queue that begins with the name Base SAS and is unique within the system. The example then queries the name of the temporary dynamic queue that was created after a successful OPEN.

```sas
length qname $ 48;
hod=0;
action="GEN";
rc=0;
parms="OBJECTNAME,DYNAMICQNAME"
model="SAMPLE.TEMP.MODEL";
```
CALL MQOPEN Routine

Establishes access to a WebSphere MQ object (queue, process definition, or queue manager).

Syntax

CALL MQOPEN(hConn, hod, options, hObj, compCode, reason <, compCode1, reason1<, compCode2, reason2,...>>);

Required Arguments

**hConn**

Specifies the WebSphere MQ connection handle that is obtained from a previous CALL MQCONN routine.

Type  Numeric, Input

**hod**

Specifies the Base SAS internal object descriptor handle that is obtained from a previous CALL MQOD routine.

Type  Numeric, Input

**options**

Specifies a string of open options, each separated by a comma. The following open options are valid:

- **INPUT_AS_Q_DEF**
  Open to get messages using queue-defined default.

- **INPUT_SHARED**
  Open to get messages with shared access.

- **INPUT_EXCLUSIVE**
  Open to get messages with exclusive access.

- **BROWSE**
  Open to browse messages.

- **OUTPUT**
  Open to put messages.

- **INQUIRE**
  Open to query object attributes.

- **SET**
  Open to set object attributes.
SAVE_ALL_CONTEXT
   Save context when message is received.

PASS.IDENTITY_CONTEXT
   Allow identity context to be passed.

PASS_ALL_CONTEXT
   Allow all context to be passed.

SET.IDENTITY_CONTEXT
   Allow identity context to be set.

SET_ALL_CONTEXT
   Allow all context to be set.

ALTERNATE_USER_AUTHORITY
   Validate with specified user identifier.

FAIL_IFQUIESCING
   Fail if MQ is quiescing.

The following options apply only when opening a cluster queue:

BIND_AS_Q_DEF
   Use default binding for queue.

BIND_NOT_FIXED
   Do not bind to a specific destination.

BIND_ON_OPEN
   Bind handle to destination when queue is opened.

Type  Character, Input

hObj
   Returns the WebSphere MQ handle that will be used in subsequent message
   queuing calls to identify the object that is being accessed (a queue, a process
   definition, or queue manager).

Type  Numeric, Output

compCode
   Returns the WebSphere MQ completion code. This parameter can be used to
determine whether an error occurred during the execution of this routine. If an
error occurred, then the compCode parameter will be nonzero, and the reason
parameter will be set to the appropriate reason code.

Type  Numeric, Output

reason
   Returns the WebSphere MQ reason code that qualifies compCode.

Note: A reason code of -1 reflects a Base SAS internal error, not a WebSphere
MQ error. To obtain a textual description of a failure (either Base SAS or
WebSphere MQ), use the SYSMSG() Base SAS function call.

Type  Numeric, Output
Optional Argument

**compCodex, reasonx**

The `compCodex` and `reasonx` are an optional series of paired values that can be used when opening a distribution list in order to discern failures for specific queues within the distribution list. These parameters support features of WebSphere MQ Version 5.1 and later.

**Type** Numeric, Output

### Example

This example opens a queue for input and output.

```sql
options="INPUT_SHARED,OUTPUT";
hObj=0;
compCode=0;
reason=0;
CALL MQOPEN(hConn, hod, options, hObj,
compCode,
reason);
```

### CALL MQPMO Routine

Manipulates WebSphere MQ put message options to be used on a subsequent CALL MQPUT routine.

**Syntax**

```
CALL MQPMO(hpmo, action, rc<, parms, value1, value2, ...>);
```

**Required Arguments**

**hpmo**

On input, it specifies the Base SAS internal put message options handle. The handle should be supplied when you are setting or querying an option. The handle is generated as output when `action` is to generate default WebSphere MQ put options.

**Type** Numeric, Input or Output

**action**

Specifies the desired action of this routine. The following `action` values are valid:

**GEN**

Generate a handle representing default put message options as defined by WebSphere MQ.

**SET**

After a put message options handle has been generated, you can continue to set values as necessary.

**INQ**

After a put message options handle has been generated, you can query its values.
Type Character, Input

rc
Provides the Base SAS return code from this function. If an error occurs, then the return code is nonzero. You can use the Base SAS function SYMSMSG() to obtain a textual description of the return code.

Type Numeric, Output

Optional Arguments

parms
Specifies an optional string of put message options that you want to set for subsequent CALL MQPUT routines. Each option must be separated by a comma and must have a value associated with it in the function's parameter list.

Type Character, Input

value
Provides the value for an option specified in the parms string. You must provide a value parameter for each option specified in the parms string and the data type must be of the proper type. Variables used to store character values that are being returned in an inquiry (INQ action) should be initialized appropriately to guarantee that truncation of a returned value does not occur.

Type Character or Numeric, Input or Output

Put Message Options

The following put message options are valid:

CONTEXT
Object handle of input queue.

Type Numeric, Input

RESOLVEDQNAME
Resolved name of destination queue.

Type Character48, Output

RESOLVEDQMGRNAME
Resolved name of destination queue manager.

Type Character48, Output

OPTIONS
Character string of the attributes (options) to associate with subsequent CALL MQPUT routines. Each option must be separated by a comma.

The following OPTIONS values are valid:

NONE
Default

SYNCPOINT
Put message inside current unit of work
NO_SYNCPOINT
   Put message outside current unit of work

DEFAULT_CONTEXT
   Associate default context with the message

PASS_IDENTITY_CONTEXT
   Pass identity context from an input queue handle

PASS_ALL_CONTEXT
   Pass all context from an input queue handle

SET_IDENTITY_CONTEXT
   Set identity context from the application

SET_ALL_CONTEXT
   Set all context from the application

ALTERNATE_USER_AUTHORITY
   Validate with specified user identifier

FAIL_IF_QUIESCING
   Fail if QMgr is quiescing

NO_CONTEXT
   Associate no context with the message

The following OPTIONS values support WebSphere MQ Version 5.1 and later (these values are not supported on z/OS):

NEW_MSGID
   Generate a new message identifier

NEW_CORRELID
   Generate a new correlation identifier

LOGICAL_ORDER
   Messages in groups and segments are put in logical order

Type  Character, Input

Example
This example demonstrates the generate, set, and inquire actions of the CALL MQPMO routine.

length parms $ 30;
length rq rqmgr $ 48;
   /* generate default put message options */
   hpmo=0;
   action="GEN";
   rc=0;
   CALL MQPMO(hpmo, action, rc);
   /* set non-default put message options parameters */
   action="SET";
   parms="OPTIONS";
   options="SYNCPOINT,FFAIL_IF_QUIESCINC";
   CALL MQPMO(hpmo, action, rc, parms, options);
   /* inquire about resolved names after successful PUT */
   action="INQ";
   parms="RESOLVEDQNAME,RESOLVEDQMGRNAME";
   CALL MQPMO(hpmo, action, rc, parms, rq,
CALL MQPUT Routine

Puts a message on a WebSphere MQ queue that has been previously opened.

Syntax

\[
\text{CALL MQPUT}(hConn, hObj, hmd, hpamo, hData, compCode, reason <, compCode1, reason1<, compCode2, reason2, \ldots \ldots >); \]

Required Arguments

\textit{hConn}

Specifies the WebSphere MQ Connection handle that is obtained from a previous CALL MQCONN routine.

Type Numeric, Input

\textit{hObj}

Specifies the WebSphere MQ handle to an open object that is obtained from a previous CALL MQOPEN routine.

Type Numeric, Input

\textit{hmd}

Specifies the Base SAS internal message descriptor handle that is obtained from a previous CALL MQMD routine.

Type Numeric, Input

\textit{hpamo}

Specifies the Base SAS internal put message options handle that is obtained from a previous CALL MQPMO routine.

Type Numeric, Input

\textit{hData}

Specifies the Base SAS internal data descriptor handle that is obtained from a previous CALL MQSETPARMS routine. If set to zero, then it is assumed that no data will accompany this message. For WebSphere MQ Version 5.1 and later, \textit{hData} can also represent a reference message header that is obtained from a previous CALL MQRMH routine.

Type Numeric, Input

\textit{compCode}

Returns the WebSphere MQ completion code. This parameter can be used to determine whether an error occurred during the execution of this routine. If an error occurred, then the \textit{compCode} parameter will be nonzero, and the \textit{reason} parameter will be set to the appropriate reason code.

Type Numeric, Output
**reason**

Returns the WebSphere MQ reason code that qualifies `compCode`.

---

**Note:** A reason code of -1 reflects a Base SAS internal error, not a WebSphere MQ error. To obtain a textual description of a failure (either Base SAS or WebSphere MQ), use the `SYSMSG()` Base SAS function call.

---

**Optional Argument**

**compCodex, reasonx**

The `compCodex` and `reasonx` are an optional series of paired values that can be used when opening a distribution list in order to discern failures for specific queues within the distribution list. These parameters support features of WebSphere MQ Version 5.1 and later.

---

**Example**

This example sends a message to a queue.

```sas
compCode=0;
reason=0;
CALL MQPUT(hConn, hObj, hmd, hpmo, hData, compCode, reason);
```

---

**CALL MQPUT1 Routine**

Sends a single message, often a reply, to a queue.

---

**Syntax**

```sas
CALL MQPUT1(hConn, hod, hmd, hpmo, hData, compCode, reason <, compCode1, reason1<, compCode2, reason2,…>>>);
```

---

**Required Arguments**

**hConn**

Specifies the WebSphere MQ connection handle that is obtained from a previous CALL MQCONN routine.

**Type** Numeric, Input

**hod**

Specifies the Base SAS internal object descriptor handle that is obtained from a previous CALL MQOD routine.
CALL MQPUT1 Routine

**hmd**
Specifies the Base SAS internal message descriptor handle that is obtained from a previous CALL MQMD routine.

**hpmo**
Specifies the Base SAS internal put message options handle that is obtained from a previous CALL MQPMO routine.

**hData**
Specifies the Base SAS internal data descriptor handle that is obtained from a previous CALL MQSETPARMS routine. If set to zero, then it is assumed that no data will accompany this message. For WebSphere MQ Version 5.1 and later, `hData` can also represent a reference message header that is obtained from a previous CALL MQRMH routine.

**compCode**
Returns the WebSphere MQ completion code. This parameter can be used to determine whether an error occurred during the execution of this routine. If an error occurred, then the `compCode` parameter will be nonzero, and the `reason` parameter will be set to the appropriate reason code.

**reason**
Returns the WebSphere MQ reason code that qualifies the completion code.

**Note:** A reason code of -1 reflects a Base SAS internal error, not a WebSphere MQ error. To obtain a textual description of a failure (either Base SAS or WebSphere MQ), use the SYSMSG() Base SAS function call.

**Optional Argument**

**compCodex, reasonx**
The `compCodex` and `reasonx` are an optional series of paired values that can be used when opening a distribution list in order to discern failures for specific queues within the distribution list. These parameters support features of WebSphere MQ Version 5.1 and later.

**Details**
Essentially, the CALL MQPUT1 routine uses the CALL MQOPEN, CALL MQPUT, and CALL MQCLOSE routines in one API call. The queue does not have to be open
before you make this call. Also note that the queue will be closed during the execution of this call.

Example
This example sends a message to a queue that might not already be opened.

```c
compCode=0;
reason=0;
CALL MQPUT1(hConn, hod, hmd, hpmo, hData, compCode, reason);
```

CALL MQRMH Routine

Creates or manipulates a reference message header so that an application can put a message in this format, omitting the bulk data.

Syntax

```
CALL MQRMH(hrmh, action, rc, parms, value1<, value2,…>);
```

Required Arguments

**hrmh**

Specifies a Base SAS internal handle to a reference message header. The handle is generated as output when *action* is to generate default message header parameters. The handle should be supplied when you are setting or querying a parameter.

*Type* Numeric, Input or Output

**action**

Specifies the desired action of this routine. The following *action* values are valid:

- **GEN**
  Generate a handle representing default reference message header parameters as defined by WebSphere MQ.

- **SET**
  After a message header handle has been generated, you can set values as necessary.

- **INQ**
  After a message header handle has been generated, you can query its values.

*Type* Character, Input

**rc**

Provides the Base SAS return code from this function. If an error occurs, then the return code is nonzero. You can use the Base SAS function SYSMSG() to obtain a textual description of the return code.
parms
Specifies an optional string of reference message header parameters that you want to set. Each parameter must be separated by a comma and must have a value associated with it in the function's parameter list. The OBJECTTYPE, SRCNAME, and DESTNAME parameters should be defined.

value
Provides a value for a reference message header parameter that is specified in the parms string. You must provide a value parameter for each reference message header parameter that is specified in the parms string and the data type must be of the proper type. Variables that are used to store character values being returned in an inquiry (INQ action) should be initialized appropriately to guarantee that truncation of a returned value does not occur.

RMH Parameters and Values
The following reference message header parameters (parms) and values are valid:

**ENCODING**
Data encoding
- Type: Numeric, Input

**CODEDCHARSETID**
Coded character set identifier
- Type: Numeric, Input

**FORMAT**
Format name
- Type: Character8, Input

**OBJECTTYPE**
Object type
- Type: Character8, Input

**SRCNAME**
Source object name
- Type: Character, Input

**DESTNAME**
Destination object name
- Type: Character, Input
Details

When the reference message header is read from the transmission queue by a message channel agent (MCA), a user-supplied message exit is invoked to process the reference message. A sample message exit is supplied by WebSphere MQ, amqxsrm. You must add this message exit to the sending and receiving channel definitions. The message exit on the sending side can append to the reference message the bulk data identified by the reference message header before the MCA sends the message through the channel to the next queue manager.

When a reference message is received, the receiving message exit should create the object from the bulk data that is associated with the reference message header, and then pass on the reference message without the bulk data so that the reference message (without the bulk data) can later be retrieved by a program.

Example

This example goes through the process of connecting to a queue manager, preparing the queue, generating the message, closing the queue, and freeing all resources.

```plaintext
data _null_; length hconn hobj cc reason 8; length rc hod hpmo hmd hrmh 8; length msg $ 200; hconn=0; hobj=0; hod=0; hpmo=0; hmd=0; hrmh=0; put '---------------- Connect to QMgr ----------------'; call mqconn("TESTQMGR", hconn, cc, reason); if cc ^= 0 then do; if reason = 2002 then do; put 'Already connected to QMgr ' qmgr; end; else do; if reason = 2059 then put 'MQCONN: QMgr not available... needs to be started'; else put 'MQCONN: failed with reason= ' reason; goto exit; end; end; else put 'MQCONN: successfully connected to QMgr ' qmgr; put '--------- Generate object descriptor ---------'; call mqod(hod, "GEN", rc, "OBJECTNAME", "TESTQ"); if rc ^= 0 then do; put 'MQOD: failed with rc= ' rc; msg = sysmsg(); put msg; goto exit; end; else put 'MQOD: successfully generated object descriptor';```
put '-------- Open queue object for output ---------';
call mqopen(hconn, hod, "OUTPUT", hobj, cc, reason);
if cc ^= 0 then do;
    put 'MQOPEN: failed with reason= ' reason;
    goto exit;
end;
else put 'MQOPEN: successfully opened queue for output';
put '--------- Generate put message options --------';
call mqpmo(hpmo, "GEN", rc);
if rc ^= 0 then do;
    put 'MQPMO: failed with rc= ' rc;
    msg = sysmsg();
    put msg;
    goto exit;
end;
else put 'MQPMO: successfully generated put message options';
put '--------- Generate message descriptor ---------';
/* format must be set to reference message header */
call mqmd(hmd, "GEN", rc, "PERSISTENCE,FORMAT",
            "PERSISTENT", "MQHREF");
if rc ^= 0 then do;
    put 'MQMD: failed with rc= ' rc;
    msg = sysmsg();
    put msg;
    goto exit;
end;
else put 'MQMD: successfully generated message descriptor';
/** reference message header **/
call mqrmh(hrmh, "GEN", rc,
            "SRCNAME,DESTNAME,OBJECTTYPE",
            "d:\test.txt", "d:\testdup.txt", "FLATFILE");
if rc ^= 0 then do;
    put 'MQRMH: failed with rc= ' rc;
    msg = sysmsg();
    put msg;
    goto exit;
end;
else put 'MQRMH: successfully generated reference message header';
put '------------- Put message on queue -------------';
call mqput(hconn, hobj, hmd, hpmo, hrmh, cc, reason);
if cc ^= 0 then do;
    put 'MQPUT: failed with reason= ' reason;
    msg = sysmsg();
    put msg;
    goto exit;
end;
else put 'MQPUT: successfully put message on queue';
exit:
if hobj ^= 0 then do;
    put '----------------- Close queue -----------------';
call mqclose(hconn, hobj, "NONE", cc, reason);
if cc ^= 0 then do;
    put 'MQCLOSE: failed with reason= ' reason;

CALL MQSET Routine

Changes the attributes of a queue object.

Syntax

CALL MQSET(hConn, hObj, compCode, reason, parms, value1<, value2, ...>);

Required Arguments

hConn

Specifies the WebSphere MQ connection handle that is obtained from a previous CALL MQCONN routine.

Type  Numeric, Input

hObj

Specifies the WebSphere MQ object handle that is obtained from a previous CALL MQOPEN routine that specified the SET option. This handle represents a queue object.
**compCode**

Returns the WebSphere MQ completion code. This parameter can be used to
determine whether an error occurred during the execution of this routine. If an
error occurred, then the `compCode` parameter will be nonzero, and the `reason`
parameter will be set to the appropriate reason code.

**reason**

Returns the WebSphere MQ reason code that qualifies the completion code.

*Note:* A reason code of -1 reflects a Base SAS internal error, not a WebSphere
MQ error. To obtain a textual description of a failure (either Base SAS or
WebSphere MQ), use the SYSMSG() Base SAS function call.

**parms**

Specifies a string of queue attributes that you want to set for a WebSphere MQ
queue. Each queue attribute must be separated by a comma and must have a
value associated with it. Only certain attributes (a subset of list for the CALL
MQINQ routine) can be changed by using this CALL routine. Refer to the IBM
WebSphere MQ documentation for more details.

**value**

Provides the value for an attribute that is specified in the `parms` string. You must
provide a `value` parameter for each attribute that is specified in the `parms` string,
and the data type must be of the proper type.

**Example**

This example changes the queue properties by inhibiting messages to be sent (put)
to the queue.

```sas
length parms $ 30;
compCode=0;
reason=0;
parms="INHIBIT_PUT";
inhibit=1;
CALL MQSET(hConn, hObj, compCode,
reason, parms,
inhibit);
```
CALL MQSETPARMS Routine

Creates a data descriptor that describes the actual Base SAS variables along with an associated data mapping. This data descriptor can then be used on a subsequent CALL MQPUT or CALL MQPUT1 routine.

Syntax

CALL MQSETPARMS(hData, hMap, rc\<, parm1, parm2, parm3, \ldots\>);

Required Arguments

**hData**

Returns a Base SAS internal data descriptor handle. The handle that is generated can be used to reference the data when sending a message to a queue.

Type Numeric, Output

**hMap**

Specifies a Base SAS internal map descriptor handle that is obtained from a previous CALL MQMAP routine. If set to zero, no external defined mapping is assumed and therefore, all data is mapped according to Base SAS internal representations. That is, all numerics are mapped as doubles and all strings are mapped as character data of the current string length.

Type Numeric, Input

**rc**

Provides the Base SAS return code from this function. If an error occurs, then the return code is nonzero. You can use the Base SAS function SYSMSG() to obtain a textual description of the return code.

Type Numeric, Output

**parm**

Specifies the Base SAS variables to set.

Type Character or Numeric, Input

Example

This example sets values of Base SAS variables into a message.

```
hData=0;
rc=0;
parm1=100;
parm2=9999;
parm3=9999.9999;
parm4="This is a test."
CALL MQSETPARMS(hData, hMap, rc,
parm1, parm2, parm3, parm4);
```
Using Microsoft Message Queueing Services (MSMQ)

MSMQ Functional Interface

SAS Integration Technologies allows applications developers to combine the power of both SAS information delivery and Microsoft message queuing capabilities by providing a SAS interface to the Microsoft Message Queuing Services (MSMQ), which are part of Windows. With this interface, SAS programs can create new MSMQ message queues or use existing message queue that are available throughout the enterprise.

Writing MSMQ Applications

In MSMQ messaging, two or more applications communicate with each other indirectly and asynchronously by using message queues. The applications do not have to be running at the same time or even in the same operating environment. An application can communicate with another application by sending a message to a queue. The receiving application retrieves the message when it is ready.

A typical SAS program that uses MSMQ services performs the following tasks:

1. A program must first either open an existing queue or create a new queue. A function is available to help find queues based on their property values. If opening an existing queue, the program supplies a queue identifier to select the appropriate queue. If creating a new queue, a queue identifier is returned to the program to be used in subsequent calls. The queue identifier is used by MSMQ in a distributed database that maintains information about users, queues, queue managers, host machines, and network layout. This database is referred to as
the MSMQ Information Store (MQIS) and helps insulate the application developer from the details of the network.

2 When creating a queue, you can declare it public or private. Public queues are registered in the MQIS and can be accessed throughout the network. Private queues, on the other hand, can be accessed only by systems that know the queue's full pathname or format name. Other properties can be set when creating a queue such as security, message handling, and types of services provided by the queue. These same types of properties can also be retrieved from or set on a queue that has been opened.

3 A program that has opened a queue can compose and send a message. To compose a message, a function is used to identify a data map that describes the format, the number and the type of parameters to be sent as part of the message. The data map is used by a function that creates a data descriptor of the actual values of the SAS variables to be included in the message. If your distributed application uses a Microsoft Transaction Server (MTS), then a transaction object can be used to send the message based on the success of the transaction.

4 A program can also retrieve messages from an opened queue. MSMQ uses the concept of a cursor to identify the location of the message within a queue. A message can be read from the current cursor location, or you can see the next location. When a message is read, the program can choose to remove the message or leave it on the queue. In addition, a number of message properties such as security issues, size, identification, and statistics on the delivery can also be retrieved.

5 After a program has sent or retrieved all its messages, queues can also be closed or deleted. This releases the resources that were allocated when the queue was opened or created.

Note: MSMQ uses several representations to identify a queue, such as format name, pathname, instance UUID, and queue handle. There are functions available that you can use to convert between representations.

MSMQ Code Samples

Introduction to the MSMQ Code Samples

This section provides examples of using the MSMQ interface with DATA step code to illustrate the semantics of sending a message to a queue and receiving the same message from the queue.

For DATA step code examples that show how to send and receive files, see “Processing a Text File” on page 109 and “Processing a Binary File” on page 115.

Note: When a SAS DATA step ends, all resources that are consumed by this DATA step are automatically freed. That is, all internal SAS handles are automatically
freed. When using the SAS Macro Language to interface with MSMQ, ensure that all resources are freed programmatically. Unlike the DATA step, resources consumed by the SAS Macro Language are never implicitly freed during SAS execution.

DATA Step Coding Examples

Sending a Message to a Queue

This example sends a message to a queue. Note that it assumes that the queue "respq" has been created before this example.

data _null_; length rc 8; length msg $ 200; length Qid hQueue transobj 8; length msgid $ 40; length hData hMap 8; length parm1 parm2 parm3 8; length parm4 $ 50;

hQueue=0; hMap=0; hData=0;

put '-------- Obtain formatname from pathname ------'; Qid=0; rc=0; call msmqpathtoformat("pcpad\testq", Qid, rc); if rc ^= 0 then do; if rc = input('03000EC0'x, ib4.) then do; /* C00E0003 - MSMQ QUEUE_NOT_FOUND error */ /* so create it... */ put 'Queue does not exist so creating it...'; call msmqcreatequeue(Qid, rc, "PATHNAME, LABEL", "pcpad\testq", "Test Queue"); if rc ^= 0 then do; put 'MSMQCreateQueue: failed'; msg = sysmsg(); put msg; goto exit; end; else put 'MSMQCreateQueue: succeeded'; end; else do; put 'MSMQPathToFormat: failed'; msg = sysmsg(); put msg; goto exit; end; else put 'MSMQPathToFormat: succeeded';
put '----------- Open queue for sending -----------';
call msmqopenqueue(Qid, "SEND", "SHARE", hQueue, rc);
if rc ^= 0 then do;
   put 'MSMQOpenQueue: failed';
   msg = sysmsg();
   put msg;
   goto exit;
end;
else put 'MSMQOpenQueue: succeeded';

put '----------- Generate map descriptor -----------';
/* data will not be aligned */
desc1="SHORT";
desc2="LONG";
desc3="DOUBLE";
desc4="CHAR,,50"; /* blank pad to 50 bytes */
call msmqmap(hMap, rc, desc1, desc2, desc3, desc4);
if rc ^= 0 then do;
   put 'MSMQMap: failed';
   msg = sysmsg();
   put msg;
   goto exit;
end;
else put 'MSMQMap: succeeded';

put '--- Generate data descriptor - actual data ----';
parm1=100;
parm2=9999;
parm3=9999.9999;
parm4="This is a test."

call msmqsetparms(hData, hMap, rc, parm1,
                   parm2, parm3, parm4);
if rc ^= 0 then do;
   put 'MSMQSetParms: failed';
   msg = sysmsg();
   put msg;
   goto exit;
end;
else put 'MSMQSetParms: succeeded';

put '------------ Send message to queue ------------';
transobj=0;
msgid="";
call msmqsendmsg(hQueue, hData, transobj, rc,
                   "BODY_TYPE,CORRELATIONID,LABEL,MSGID,
                   PRIV_LEVEL,RESP_QUEUE",
                   999, "0102030405060708090A0B0C0D0E0F1011121314",
                   "Secret test message", msgid,
                   "PRIVATE", "pcpad\respq");
if rc ^= 0 then do;
   put 'MSMQSendMsg: failed';
msg = sysmsg();
put msg;
end;
else do;
   put 'MSMQSendMsg: succeeded';
   /* display MSMQ-generated MSGID */
   put 'msgid is ' msgid;
end;

exit:
if hQueue ^= 0 then do;
   put '----------------- Close queue ---------------';
   call msmqclosequeue(hQueue, rc);
   if rc ^= 0 then do;
      put 'MSMQCloseQueue: failed';
      msg = sysmsg();
      put msg;
   end;
   else put 'MSMQCloseQueue: succeeded';
end;
if Qid ^= 0 then do;
   call msmqfree(Qid);
   put 'Qid handle freed';
end;
if hMap ^= 0 then do;
   call msmqfree(hMap);
   put 'Map descriptor handle freed';
end;
if hData ^= 0 then do;
   call msmqfree(hData);
   put 'Data descriptor handle freed';
end;

run;

Receiving a Message from a Queue
This example receives a message from a queue.

data _null_;
   length rc 8;
   length msg $ 200;
   length Qid hQueue transobj 8;
   length hMap 8;
   length arrivet auth size sentt 8;
   length correlid msgid $ 40;
   length label $ 80;
   length parm1 parm2 parm3 8;
   length parm4 $ 50;
length hRespQ 8;
length respq $ 80;
length respQid 8;

hQueue=0;
hMap=0;
hRespQ=0;
respQid=0;

put '-------- Obtain formatname from pathname ------';
Qid=0;
rc=0;
call msmqpathtoformat("pcpad\testq", Qid, rc);
if rc ^= 0 then do;
    put 'MSMQPathToFormat: failed';
    msg = sysmsg();
    put msg;
    goto exit;
end;
else put 'MSMQPathToFormat: succeeded';

put '---------- Open queue for receiving ---------';
call msmqopenqueue(Qid, "RECEIVE", "SHARE", hQueue, rc);
if rc ^= 0 then do;
    put 'MSMQOpenQueue: failed';
    msg = sysmsg();
    put msg;
    goto exit;
end;
else put 'MSMQOpenQueue: succeeded';

put '---------- Receive message from queue ----------';
transobj=0;
hCursor=0;
call msmqreceivingmsg(hQueue, 0, "RECEIVE", hCursor, transobj, rc, "ARRIVEDTIME, AUTHENTICATED, BODY_SIZE, CORRELATIONID, LABEL, MSGID, RESP_QUEUE, SENTTIME",
    arrivet, auth, size, correlid, label, msgid, respq, sentt);
if rc ^= 0 then do;
    put 'MSMQReceiveMsg: failed';
    msg = sysmsg();
    put msg;
    goto exit;
end;
else do;
    put 'MSMQReceiveMsg: succeeded';
    /* convert MSMQ arrived time to
    SAS datetime format */
    arrivet =
        arrivet + 10*365*24*3600 + 3*24*3600 - 5*3600;
    put 'arrived time is' arrivet datetime.;
    if auth = 1 then put 'message was authenticated';
    else put 'message was not authenticated';
put 'message body size is ' size;
put 'correlation id is ' correlid;
put 'label is ' label;
put 'msg id is ' msgid;
put 'resp_queue Qid handle is ' respq;
/* convert MSMQ sent time to SAS datetime format */
  sentt = sentt + 10*365*24*3600 + 3*24*3600 - 5*3600;
  put 'sent time was' sentt datetime.;
end;

if size ^= 0 then do;
  put '---------- Generate map descriptor ----------';
desc1="SHORT";
desc2="LONG";
desc3="DOUBLE";
desc4="CHAR,,50";
call msmqmap(hMap, rc, desc1, desc2, desc3, desc4);
if rc ^= 0 then do;
  put 'MSMQMap: failed';
  msg = sysmsg();
  put msg;
  goto exit;
end;
else put 'MSMQMap: succeeded';
call msmqgetparms(hMap, rc, parm1, parm2, parm3, parm4);
if rc ^= 0 then do;
  put 'MSMQGetParms: failed';
  msg = sysmsg();
  put msg;
  goto exit;
end;
else do;
  put 'MSMQGetParms: succeeded';
  put 'parm1 = ' parm1;
  put 'parm2 = ' parm2;
  put 'parm3 = ' parm3;
  put 'parm4 = ' parm4;
end;
else put 'No data was associated with the message';

/* post a reply to the response queue if available */
if respq ^= "" then do;
call msmqpathtoformat(respq, respQid, rc);
if rc ^= 0 then do;
  put 'MSMQPathToFormat: failed to open response queue';
  msg = sysmsg();
  goto exit;
end;
call msmqopenqueue(respQid, "SEND", "SHARE", hRespQ, rc);
if rc ^= 0 then do;
    put 'MSMQOpenQueue: failed to open response queue';
    msg = sysmsg();
    put msg;
    goto exit;
end;

hMap=0;
call msmqsetparms(hData, hMap, rc, "Message received OK");
if rc ^= 0 then do;
    put 'MSMQSetParms: failed to send response message';
    msg = sysmsg();
    put msg;
    goto exit;
end;

transobj=0;
call msmqsendmsg(hRespQ, hData, transobj, rc);
if rc ^= 0 then do;
    put 'MSMQSendMsg: failed to send response message';
    msg = sysmsg();
    put msg;
    end;
else put 'reply sent to the response queue';
end;

exit:
if hQueue ^= 0 then do;
    put '----------------- Close queue ---------------';
call msmqclosequeue(hQueue, rc);
if rc ^= 0 then do;
    put 'MSMQCloseQueue: failed';
    msg = sysmsg();
    put msg;
    end;
else put 'MSMQCloseQueue: succeeded';
end;

if hRespQ ^= 0 then do;
    put '------------ Close Response Queue -----------';
call msmqclosequeue(hRespQ, rc);
if rc ^= 0 then do;
    put 'MSMQCloseQueue: failed to close response queue';
    msg = sysmsg();
    put msg;
    end;
else put 'MSMQCloseQueue: succeeded';
to close response queue;
end;
if Qid ^= 0 then do;
call msmqfree(Qid);
put 'Qid handle freed';
end;
if respQid ^= 0 then do;
call msmqfree(respQid);
put 'respQid handle freed';
end;
if hMap ^= 0 then do;
call msmqfree(hMap);
put 'Map descriptor handle freed';
end;
run;

Processing a Text File

This example shows how to put a text file on a queue.

data _null_;
  length rc 8;
  length msg $ 200;
  length Qid hQueue hmap 8;
  length appspec 8;
  length corrid $ 40;
  length record $ 256;
  length seqno 8 seqstr $ 4;
/* send this file to the queue */
  infile 'd:\test.txt' length=reclen end=eof;
  put '--------- Obtain Formatname from Pathname -------';
  call msmqpathtoformat(".\	estq", Qid, rc);
  if( rc ) then do;
    if( rc = input('03000EC0'x,ib4.) ) then do;
      put 'Queue does not exist so create it...';
      call msmqcreatequeue(Qid, rc, "pathname,label",
        ".\testq", "test queue");
      if( rc ) then do;
        put 'MSMQCreateQueue: failed';
        msg = sysmsg();
        put msg;
        goto exit;
      end;
    end;
  end;
  else do;
    put 'MSMQPathToFormat: failed';
    msg = sysmsg();
    put msg;
  end;
goto exit;
end;
end;

put '------------------ Open Queue ------------------';
call msmqopenqueue(Qid, "SEND", "SHARE", hQueue, rc);
if( rc ) then do;
   put 'MSMQOpenQueue: failed';
   msg = sysmsg();
   put msg;
   goto exit;
end;

put '----------- Generate map descriptor -----------';
/* longest record in file is 255 bytes+1 length byte... */
/* therefore all messages on the queue pertaining to */
/* this file will be blank-padded for 256 bytes... */
call msmqmap(hmap, rc, "char,,256");
if rc ^= 0 then do;
   put 'MSMQMap: failed';
   msg = sysmsg();
   put msg;
   goto exit;
end;

/* designate that messages belong to a text file */
appspec=100000;

/* all of these messages will have 
   the same correlationid+seqno */
corrid="46696c65212121"; /* File!!! */
segno = 0;

do until(eof);
   input @;
   input record $varying256. reclen;
   
   call msmqsetparms(hdata, hmap, rc, record);
   if( rc ) then do;
      put 'MSMQSetParms: failed';
      msg = sysmsg();
      put msg;
      goto exit;
   end;

   /* add sequence # to correlationid */
   seqstr = put(seqno, hex4.);
   substr(corrid,15,4) = seqstr;
   seqno = seqno+1;

   put '--- Send message to queue ----';
call msmqsendmsg(hQueue, hdata, 0, rc,
      "appspecific,correlationid", appspec, corrid);
if( rc ) then do;
  put 'MSMQSendMsg: failed';
  msg = sysmsg();
  put msg;
  goto exit;
end;

/* free data */
call msmqfree(hdata);
end;

exit:
if( hQueue ) then do;
call msmqclosequeue(hQueue, rc);
if( rc ) then do;
  put 'MSMQCloseQueue: failed';
  msg = sysmsg();
  put msg;
end;
end;

if( Qid ) then
call msmqfree(Qid);

if( hmap ) then
call msmqfree(hmap);

stop;
run;

Getting a Text File from a Queue

This example shows how to receive the first text file on a queue. The appspecific parameter is equal to 100000.

filename output
'd:\testdup.txt';
data _null_;
length rc 8;
length msg $ 200;
length Qid hQueue hmap hCursor hCursor2 8;
length corrid corrid2 filecorrid $ 40;
length appspec 8;
length action action2 $ 12;
length record $ 256;
length seqno 8;

fileid = fopen('output', 'o', 256, 'v');
if( fileid = 0 ) then do;
  put 'Error opening output file...';
  goto exit;
end;
put '--------- Obtain Formatname from Pathname --------';
call msmqpathtoformat("\testq", Qid, rc);
if( rc ) then do;
   put 'MSMQPathToFormat: failed';
   msg = sysmsg();
   put msg;
   goto exit;
end;

put '--------------- Open Queue ------------------';
call msmqopenqueue(Qid, "RECEIVE", "SHARE", hQueue, rc);
if( rc ) then do;
   put 'MSMQOpenQueue: failed';
   msg = sysmsg();
   put msg;
   goto exit;
end;
call msmqcursorexist(hQueue, hCursor, rc);
if( rc ) then do;
   put 'MSMQCreateCursor failed';
   msg = sysmsg();
   put msg;
end;

/* peek first to see if belongs to the file you want */
action="PEEK_CURRENT";
seqno=0;
recv:
call msmqreceive(hQueue, 0, action, hCursor, 0, rc,
                 "APPSPECIFIC,CORRELATIONID", appspec, corrid);
if( rc ) then do;
   if( rc = input('1B000EC0X,ib4.) ) then do;
      put 'reached end of queue';
      goto exit;
   end;
   put 'MSMQReceiveMsg: failed';
   msg = sysmsg();
   put msg;
   goto exit;
end;

/* default action */
action="PEEK_NEXT";
if( appspec = 100000 ) then do;
   /* file processing... */
   outofseq=0;
   if( filecorrid = "" ) then do;
      /* file begins at this message */
      /* write all correlating messages to this file */
filecorrid = substr(corrid,1,14);

put '--------- Generate map descriptor ---------';
/* all file messages were sent to the queue as 256 bytes blank-padded */
call msmqmap(hmap, rc, "char,,256");
if( rc ) then do;
   put 'MSMQMap: failed';
   msg = sysmsg();
   put msg;
   goto exit;
end;
end;
/* make sure message belongs to this file */
if( substr(corrid,1,14) = filecorrid ) then do;
   if( seqno ^= input(substr(corrid,15,4),
     hex4.) ) then do;
      /* this message is out of sequence so search for it */
      outofseq=1;
      call msmqcreatecursor(hQueue, hCursor2, rc);
      if( rc ) then do;
         put 'MSMQCreateCursor failed';
         msg = sysmsg();
         put msg;
         end;
      action2="PEEK_CURRENT";
      peeknxt:
      call msmqreceivemsg(hQueue, 0, action2,
       hCursor2, 0, rc, "CORRELATIONID", corrid2);
      if( rc ) then do;
         if( rc = input('1B000EC0'x,ib4.) ) then do;
            put 'Error: reached end of queue while searching for out-of-sequence msg'
            goto exit;
         end;
         if( seqno ^= input(substr(corrid2,15,4),
           hex4.) ) then do;
            action2="PEEK_NEXT";
            goto peeknxt;
         end;
      end;
      put 'MSMQReceiveMsg: failed';
      msg = sysmsg();
      put msg;
      goto exit;
   end;
end;
/* increment sequence number for next expected message */
SEQNO=SEQNO+1;

/* retrieve record from internal buffer */
call msmqgetparms(hmap, rc, record);
if( rc ) then do;
    put 'MSMQGetParms: failed';
    msg = sysmsg();
    put msg;
    goto exit;
end;

put 'write record to file';
rc = fput(fileid, record);
if( rc ) then do;
    put 'Error writing to output file buffer...';
    goto exit;
end;

/* flush it to disk */
rc = fwrite(fileid);
if( rc ) then do;
    put 'Error writing to output file...';
    goto exit;
end;

/* now remove it from the queue... 
   don't care about receiving body */
body=0;
if( outofseq ) then do;
    call msmqreleasemsg(hQueue, 0, "RECEIVE",
    hCursor2, 0, rc, "body", body);
    /* close this cursor */
    call msmqclosecursor(hCursor2, rc);
else do;
    call msmqreleasemsg(hQueue, 0, "RECEIVE",
    hCursor, 0, rc, "body", body);
end;

/* we are now pointing at the next message */
action="PEEK_CURRENT";
end;
end;

/* finish retrieving all messages belonging 
to this file */
goto recv;

exit:
if( hQueue ) then do;
    call msmqclosequeue(hQueue, rc);
    if( rc ) then do;
put 'MSMQCloseQueue: failed';
msg = sysmsg();
put msg;
end;
end;

if( Qid ) then
call msmqfree(Qid);

if( hmap ) then
call msmqfree(hmap);

/* close file */
rc = fclose(fileid);
if( rc ) then put 'Error closing output file';

run;

Processing a Binary File
This example shows how to put a binary file on a queue. It assumes that the queue named "adminq" has been created before this.

data _null_;
length rc 8;
length msg $ 200;
length Qid hQueue hmap 8;
length appspec 8;
length corrid $ 40;
length msgbuf $ 256;
length seqno 8 seqstr $ 4;

/* read in as a stream of bytes */
infile 'd:\test.exe' recfm=f lrecl=1 end=eof;

put '--------- Obtain Formatname from Pathname -------';
call msmqpathtoformat(".\	estq", Qid, rc);
if( rc ) then do;
  if( rc = input('03000EC0'x,ib4.) ) then do;
    put 'Queue does not exist so create it';
call msmqcreatequeue(Qid, rc, "pathname,label", ".\testq", "test queue:");
if( rc ) then do;
  put 'MSMQCreateQueue: failed';
  msg = sysmsg();
  put msg;
goto exit;
end;
end;
else do;
  put 'MSMQPathToFormat: failed';
  msg = sysmsg();
  put msg;
goto exit;

end;
end;

put '------------------ Open Queue ------------------';
call msmqopenqueue(Qid, "SEND", "SHARE", hQueue, rc);
if( rc ) then do;
   put 'MSMQOpenQueue: failed';
   msg = sysmsg();
   put msg;
   goto exit;
end;

put '----------- Generate map descriptor -----------';
/* send 256 byte messages to the queue */
call msmqmap(hmap, rc, "char,,256");
if( rc ) then do;
   put 'MSMQMap: failed';
   msg = sysmsg();
   put msg;
   goto exit;
end;

/* designate messages belong to a binary file */
appspec=100001;

/* all of these messages will have the same correlationid */
corrid="42696e46696c65212121"; /* BinFile!!! */
segno = 0;
i=1;
do until(eof);
   /* read a byte at a time */
   input x $char1.;
i+1;
   substr(msgbuf,i,1) = x;
   if i = 256 or eof then do;
      /* set length of this record embedded as first byte of message */
      substr(msgbuf,1,1) = put(i-1,pib1.);
      call msmqsetparms(hdata, hmap, rc, msgbuf);
      if( rc ) then do;
         put 'MSMQSetParms: failed';
         msg = sysmsg();
         put msg;
         goto exit;
      end;
      /* add sequence # to correlationid */
      segstr = put(segno, hex4.);
      substr(corrid,21,4) = segstr;
      segno = segno + 1;
      put '--- Send message to queue ----';
call msmqsendmsg(hQueue, hdata, 0, rc,
   "appspecific,correlationid,acknowledge,
   admin_queue", appspec, corrid,
   "nack_reach_queue", ".\adminq");
if( rc ) then do;
   put 'MSMQSendMsg: failed';
   msg = sysmsg();
   put msg;
   goto exit;
end;

/* free data */
call msmqfree(hdata);

/* reset message buffer entities */
i=1;
msgbuf="";
end;
end;

exit:
if( hQueue ) then do;
call msmqclosequeue(hQueue, rc);
if( rc ) then do;
   put 'MSMQCloseQueue: failed';
   msg = sysmsg();
   put msg;
end;
end;
if( Qid ) then
call msmqfree(Qid);
if( hmap ) then
call msmqfree(hmap);
ostop;
run;

Getting a Binary File from a Queue
This example shows how to receive the first binary file on a queue.

filename output 'd:\testdup.exe';
data _null_;
length rc 8;
length msg $ 200;
length Qid hQueue hmap hCursor hCursor2 8;
length corrid corrid2 filecorrid $ 40;
length appspec 8;
length action action2 $ 12;
length msgbuf stream $ 256;
length len 8;
length seqno 8;

fileid = fopen('output', 'o', 0, 'b');
if( fileid = 0 ) then do;
    put 'Error opening output file...';
    goto exit;
end;

put '---------- Obtain Formatname from Pathname ---------';
call msmqpathtoformat(".\	estq", Qid, rc);
if( rc ) then do;
    put 'MSMQPathToFormat: failed';
    msg = sysmsg();
    put msg;
    goto exit;
end;

put '------------------ Open Queue ------------------';
call msmqopenqueue(Qid, "RECEIVE", "SHARE", hQueue, rc);
if( rc ) then do;
    put 'MSMQOpenQueue: failed';
    msg = sysmsg();
    put msg;
    goto exit;
end;

call msmqcreatecursor(hQueue, hCursor, rc);
if( rc ) then do;
    put 'MSMQCreateCursor failed';
    msg = sysmsg();
    put msg;
end;

/* peek first to see if belongs to the file you want */
action="PEEK_CURRENT";

seqno=0;
recv:
call msmqreceivingmsg(hQueue, 0, action, hCursor, 0, rc,
    "APPSPECIFIC, CORRELATIONID", appspec, corrid);
if( rc ) then do;
if( rc = input('1b000ec0'x, ib4.) ) then do;
    put 'reached end of queue';
    goto exit;
end;

put 'MSMQReceiveMsg: failed';
msg = sysmsg();
put msg;
goto exit;
end;
/* default action */
action="PEEK_NEXT";

if ( appspec = 100001 ) then do;
    /* file processing */
    outofseq=0;

    if( filecorrid = "" ) then do;
        /* file begins at this message */

        /* write all correlating messages to this file */
        filecorrid = substr(corrid,1,20);

        put '--------- Generate map descriptor ---------';
        /* all file messages were sent to the queue as 
        256 bytes blank-padded */
        call msmqmap(hmap, rc, "char,,256");
        if( rc ) then do;
            put 'MSMQMap: failed';
            msg = sysmsg();
            put msg;
            goto exit;
        end;
    end;

    /* make sure message belongs to this file */
    if( substr(corrid,1,20) = filecorrid ) then do;
        if( seqno ^= input(substr(corrid,21,4), hex4.) ) then do;
            /* this message is out of sequence 
            so search for it */
            outofseq=1;

            call msmqcreatecursor(hQueue, hCursor2, rc);
            if( rc ) then do;
                put 'MSMQCreateCursor failed';
                msg = sysmsg();
                put msg;
                goto exit;
            end;
        end;
    end;

    action2="PEEK_CURRENT";

    peeknxt:
        call msmqreceivingmsg(hQueue, 0, action2,
            hCursor2, 0, rc, "CORRELATIONID", corrid2);
        if( rc ) then do;
            if( rc = input('1B000EC0\x, ib4.) ) then do;
                put 'Error: reached end of queue while 
                searching for out-of-sequence msg'
                goto exit;
            end;

            put 'MSMQReceiveMsg: failed';
            msg = sysmsg();
            put msg;
            goto exit;
        end;
end;

if( seqno ^= input(substr(corrid2,21,4), hex4.))
  then do;
    action2="PEEK_NEXT";
    goto peeknxt;
  end;
end;

/* increment sequence number for next expected message */
seqno=seqno+1;

/* retrieve record from internal buffer */
call msmqgetparms(hmap, rc, msgbuf);
if( rc ) then do;
  put 'MSMQGetParms: failed';
  msg = sysmsg();
  put msg;
  goto exit;
end;

/* length of this stream is embedded as 1st byte in msg */
len = input(substr(msgbuf,1,1), pib1.);
stream = substr(msgbuf,2);
put 'write stream to file';
rc = fput(fileid, substr(stream,1,len));
if( rc ) then do;
  put 'Error writing to output file buffer...';
  goto exit;
end;

/* flush it to disk */
rc = fwrite(fileid);
if( rc ) then do;
  put 'Error writing to output file...';
  goto exit;
end;

/* now remove it from the queue... don't care about receiving body */
body=0;
if( outofseq ) then do;
  call msmqreceivingmsg(hQueue, 0, "RECEIVE",
                        hCursor2, 0, rc, "body", body);
  /* close this cursor */
  call msmqclosecursor(hCursor2, rc);
end;
else do;
  call msmqreceivingmsg(hQueue, 0, "RECEIVE",
                        hCursor, 0, rc, "body", body);
end;

    /* we are now pointing at the next message */
    action="PEEK_CURRENT";
    end;
end;

/* finish retrieving all messages belonging
to this file */
goto recv;

exit:
if( hQueue ) then do;
    call msmqclosequeue(hQueue, rc);
    if( rc ) then do;
        put 'MSMQCloseQueue: failed';
        msg = sysmsg();
        put msg;
    end;
end;

if( Qid ) then
    call msmqfree(Qid);

if( hmap ) then
    call msmqfree(hmap);

    /* close file */
    rc = fclose(fileid);
    if( rc ) then put 'Error closing output file';;

run;
### MSMQ CALL Routines

#### Dictionary

<table>
<thead>
<tr>
<th>Routine</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>CALL MSMQABORTTRANS Routine</td>
<td>123</td>
</tr>
<tr>
<td>CALL MSMQBEGINTRANS Routine</td>
<td>124</td>
</tr>
<tr>
<td>CALL MSMQCLOSECURSOR Routine</td>
<td>125</td>
</tr>
<tr>
<td>CALL MSMQCLOSEQUEUE Routine</td>
<td>126</td>
</tr>
<tr>
<td>CALL MSMQCOMMITTRANS Routine</td>
<td>127</td>
</tr>
<tr>
<td>CALL MSMQCREATECURSOR Routine</td>
<td>127</td>
</tr>
<tr>
<td>CALL MSMQCREATEQUEUE Routine</td>
<td>128</td>
</tr>
<tr>
<td>CALL MSMQDELETEQUEUE Routine</td>
<td>131</td>
</tr>
<tr>
<td>CALL MSMQFREE Routine</td>
<td>132</td>
</tr>
<tr>
<td>CALL MSMQFREESCONTEXT Routine</td>
<td>132</td>
</tr>
<tr>
<td>CALL MSMQGETPARMS Routine</td>
<td>133</td>
</tr>
<tr>
<td>CALL MSMQGETQPROP Routine</td>
<td>134</td>
</tr>
<tr>
<td>CALL MSMQGETQSEC Routine</td>
<td>137</td>
</tr>
<tr>
<td>CALL MSMQHNDLTOFORMAT Routine</td>
<td>138</td>
</tr>
<tr>
<td>CALL MSMQINSTTOFORMAT Routine</td>
<td>139</td>
</tr>
<tr>
<td>CALL MSMQLOCATE Routine</td>
<td>140</td>
</tr>
<tr>
<td>CALL MSMQMAP Routine</td>
<td>141</td>
</tr>
<tr>
<td>CALL MSMQOPENQUEUE Routine</td>
<td>144</td>
</tr>
<tr>
<td>CALL MSMQOPENQUEUE Routine</td>
<td>146</td>
</tr>
<tr>
<td>CALL MSMQPATHTOFORMAT Routine</td>
<td>147</td>
</tr>
<tr>
<td>CALL MSMQRECEIVEMSG Routine</td>
<td>148</td>
</tr>
<tr>
<td>CALL MSMQRELEASETRANS Routine</td>
<td>154</td>
</tr>
<tr>
<td>CALL MSMQSENDMSG Routine</td>
<td>155</td>
</tr>
<tr>
<td>CALL MSMQSETPARMS Routine</td>
<td>159</td>
</tr>
<tr>
<td>CALL MSMQSETQPROP Routine</td>
<td>160</td>
</tr>
<tr>
<td>CALL MSMQSETQSEC Routine</td>
<td>161</td>
</tr>
</tbody>
</table>

---

**CALL MSMQABORTTRANS Routine**

Cancels a unit of work from an MSMQ transaction.
Syntax

CALL MSMQABORTTRANS(transObj, rc);

Required Arguments

transObj
  Specifies the transaction object that is obtained from a previous CALL
  MSMQBEGINTRANS routine.
  Type  Numeric, Input

rc
  Provides the return code from the CALL routine. If an error occurs, then the
  return code is nonzero. A return code of -1 reflects a SAS internal error.
  Otherwise, it represents an MSMQ error code. You can use the SAS function
  SYSMSG() to obtain a textual description of the return code.
  Type  Numeric, Output

Example

This example cancels a unit of work from an MSMQ transaction.

length msg $ 200;
rc=0;
CALL MSMQABORTTRANS(transobj, rc);
if rc ^= 0 then do;
  put 'MSMQAbortTrans: failed';
  msg = sysmsg();
  put msg;
end;
else put 'MSMQAbortTrans:
succeeded';

CALL MSMQBEGINTRANS Routine

Creates an internal MSMQ transaction object that can be used to send messages to a queue or read
messages from a queue.

Syntax

CALL MSMQBEGINTRANS(transObj, rc);

Required Arguments

transObj
  Returns the transaction object.
  Type  Numeric, Output
**rc**

Provides the return code from the CALL routine. If an error occurs, then the return code is nonzero. A return code of -1 reflects a SAS internal error. Otherwise, it represents an MSMQ error code. You can use the SAS function SYSMSG() to obtain a textual description of the return code.

**Example**

This example creates a transaction object.

```sas
length msg $ 200;
transobj=0;
rc=0;
CALL MSMQBEGINTRANS(transobj, rc);
if rc ^= 0 then do;
   put 'MSMQBeginTrans: failed';
   msg = sysmsg();
   put msg;
end;
else put 'MSMQBeginTrans: succeeded';
```

---

**CALL MSMQCLOSECURSOR Routine**

Closes a given cursor thereby allowing MSMQ to release the associated resources.

**Syntax**

```
CALL MSMQCLOSECURSOR(hCursor, rc);
```

**Required Arguments**

**hCursor**

Specifies the handle to a cursor that is used for looking at messages in the queue. The CALL MSMQCREATECURSOR routine is used to create a cursor and obtain its handle.

**Type** Numeric, Input

**rc**

Provides the return code from the CALL routine. If an error occurs, then the return code is nonzero. A return code of -1 reflects a SAS internal error. Otherwise, it represents an MSMQ error code. You can use the SAS function SYSMSG() to obtain a textual description of the return code.

**Type** Numeric, Output
Example

This example closes a cursor.

```sas
length msg $ 200;
rc=0;
CALL MSMQCLOSECURSOR(hCursor, rc);
if rc ^= 0 then do;
    put 'MSMQCloseCursor: failed';
    msg = sysmsg();
    put msg;
end;
else put 'MSMQCloseCursor: succeeded';
```

CALL MSMQCLOSEQUEUE Routine

Closes a given queue.

Syntax

```sas
CALL MSMQCLOSEQUEUE(hQueue, rc);
```

Required Arguments

**hQueue**

Specifies the MSMQ handle to an open queue. This parameter is obtained from a previous CALL MSMQOPENQUEUE routine.

_Type_ Numeric, Input

**rc**

Provides the return code from the CALL routine. If an error occurs, then the return code is nonzero. A return code of -1 reflects a SAS internal error. Otherwise, it represents an MSMQ error code. You can use the SAS function SYSMSG() to obtain a textual description of the return code.

_Type_ Numeric, Output

Example

This example closes a queue.

```sas
length msg $ 200;
rc=0;
CALL MSMQCLOSEQUEUE(hQueue, rc);
if rc ^= 0 then do;
    put 'MSMQCloseQueue: failed';
    msg = sysmsg();
    put msg;
end;
else put 'MSMQCloseQueue: succeeded';
```
CALL MSMQCOMMITTRANS Routine

Commits a unit of work from an MSMQ transaction.

**Syntax**

```plaintext
CALL MSMQCOMMITTRANS(transObj, rc);
```

**Required Arguments**

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>transObj</code></td>
<td>Specifies the transaction object that is obtained from a previous CALL MSMQBEGINTRANS routine. Type: Numeric, Input</td>
</tr>
<tr>
<td><code>rc</code></td>
<td>Provides the return code from the CALL routine. If an error occurs, then the return code is nonzero. A return code of -1 reflects a SAS internal error. Otherwise, it represents an MSMQ error code. You can use the SAS function SYSMSG() to obtain a textual description of the return code. Type: Numeric, Output</td>
</tr>
</tbody>
</table>

**Example**

This example commits a unit of work from an MSMQ transaction.

```plaintext
length msg $ 200;
rc=0;
CALL MSMQCOMMITTRANS(transobj, rc);
if rc ^= 0 then do;
    put 'MSMQCommitTrans: failed';
    msg = sysmsg();
    put msg;
end;
else put 'MSMQCommitTrans: succeeded';
```

CALL MSMQCREATECURSOR Routine

Creates a cursor that is used to maintain a specific location in a queue when reading its messages.
CALL MSMQCREATECURSOR(hQueue, hCursor, rc);

Required Arguments

**hQueue**
Specifies the MSMQ handle to an open queue. This parameter is obtained from a previous CALL MSMQOPENQUEUE routine.

*Type* Numeric, Input

**hCursor**
Returns the handle of the cursor that is used for looking at messages in the queue. The CALL MSMQCREATECURSOR routine is used to create a cursor and obtain its handle.

*Type* Numeric, Output

**rc**
Provides the return code from the CALL routine. If an error occurs, then the return code is nonzero. A return code of -1 reflects a SAS internal error. Otherwise, it represents an MSMQ error code. You can use the SAS function SYSMSG() to obtain a textual description of the return code.

*Type* Numeric, Output

Example

This example creates a cursor.

```sas
length msg $ 200;
hCursor=0;
rc=0;
CALL MSMQCREATECURSOR(hQueue, hCursor, rc);
if rc ^= 0 then do;
  put 'MSMQCreateCursor: failed';
  msg = sysmsg();
  put msg;
end;
else put 'MSMQCreateCursor: succeeded';
```

CALL MSMQCREATEQUEUE Routine

Creates a queue at a specified MSMQ pathname.

Syntax

**CALL MSMQCREATEQUEUE(qid, rc, props, value1<, value2,...>);**
Required Arguments

qid
Returns the queue identifier that represents the format name of the queue that is created. The format name of the queue is a unique name generated by MSMQ.

| Type   | Numeric, Output |

rc
Provides the return code from the CALL routine. If an error occurs, then the return code is nonzero. A return code of -1 reflects a SAS internal error. Otherwise, it represents an MSMQ error code. You can use the SAS function SYSMSG() to obtain a textual description of the return code.

| Type   | Numeric, Output |

propids
Specifies one or more properties that the queue exhibits when it is created. This parameter is a character string with each applicable property separated by a comma. PATHNAME is the only required property. You must provide a value parameter for each property specified in the propids string. Each property ID in the propids string is associated positionally with a value parameter.

The following creation properties are valid:

AUTHENTICATE
Specifies whether the queue accepts only authenticated messages. The following values are valid:

- NONE (default)
  Specifies the queue accepts either authenticated or non-authenticated messages.
- ALWAYS
  Specifies the queue always requires authenticated messages.

BASEPRIORITY
Specifies a single base priority for all messages that are sent to a public queue. Values range from -32768 to 32767, where 32767 is the highest priority and 0 is the default priority.

JOURNAL
Determines whether messages retrieved from the queue are also copied to its journal queue. The following values are valid:

- NONE (default)
  Specifies that messages that are removed from the queue are discarded.
- ALWAYS
  Specifies that messages removed from the queue are always stored in its journal queue.

JOURNAL_QUOTA
Specifies the maximum size (in kilobytes) of the journal queue. The default size is infinite.

LABEL
Describes the queue. The default is a blank label ( ).

PATHNAME
Specifies the MSMQ pathname of the queue. The format of a public queue is:

MachineName\QueueName
The format of a private queue is:

```
MachineName\PRIVATE$\QueueName
```

**PRIV_LEVEL**
Specifies the privacy level that is required by the queue. The following values are valid:

- **NONE**
  Specifies that the queue accepts only non-private (clear) messages.

- **BODY**
  Specifies that the queue accepts only private (encrypted) messages.

- **OPTIONAL** (default)
  Specifies that the queue accepts both private and non-private messages.

**QUOTA**
Specifies the maximum size (in kilobytes) of the queue. The default size is infinite.

**TRANSACTION**
Specifies whether the queue is a transaction queue or a non-transaction queue. The following values are valid:

- **NONE** (default)
  Specifies that the queue does not accept transaction operations.

- **ALWAYS**
  Specifies that all messages that are sent to the queue must always be done through an MSMQ transaction.

**TYPE**
Specifies the type of service that is provided by the queue. The value of the TYPE property is a universal unique identifier (UUID) in the form of a character string that represents the binary data.

These defaults can either be changed programmatically by using the CALL MSMQSETQSEC routine or via the MSMQ Explorer interface.

**Type**  Character, Input

**Details**

The routine also registers the queue in the MSMQ Information Store (MQIS) for public queues or registers it on the local computer for private queues.

Security of the queue defaults as follows:

- **Owner**: process user
- **Group**: process group
- **DACL**: queue creator, has full control
- **Queue users**
  - get queue properties
  - get queue security
  - send messages

These defaults can be changed by using either the CALL MSMQSETQSEC routine or the MSMQ Explorer interface.
Example

This example creates a public queue.

```plaintext
length msg $ 200;
qid=0;
rc=0;
CALL MSMQCREATEQUEUE(qid, rc, "PATHNAME, LABEL", "pcpad\testq", "Test Queue");
if rc ^= 0 then do;
   put 'MSMQCreateQueue: failed';
   msg = sysmsg();
   put msg;
end;
else put 'MSMQCreateQueue: succeeded';
```

CALL MSMQDELETEQUEUE Routine

Deletes a queue from the MQIS in the case of public queues, or from the local computer in the case of private queues.

Syntax

```plaintext
CALL MSMQDELETEQUEUE(qid, rc);
```

Required Arguments

**qid**

Specifies the queue identifier that represents the format name of the queue to be deleted.

*Type* Numeric, Input

**rc**

Provides the return code from the CALL routine. If an error occurs, then the return code is nonzero. A return code of -1 reflects a SAS internal error. Otherwise, it represents an MSMQ error code. You can use the SAS function SYSMSG() to obtain a textual description of the return code.

*Type* Numeric, Output

Example

This example deletes a queue.

```plaintext
length msg $ 200;
rc=0;
CALL MSMQDELETEQUEUE(qid, rc);
if rc ^= 0 then do;
   put 'MSMQDeleteQueue: failed';
   msg = sysmsg();
   put msg;
end;
else put 'MSMQDeleteQueue: succeeded';
```
CALL MSMQFREE Routine

Frees a SAS internal handle, thereby releasing its resources.

Syntax

CALL MSMQFREE(handle);

Required Argument

handle

Specifies a SAS internal handle that is obtained from a previous CALL routine.

The following CALL routines return handles that can be used as input to this routine (the type of handle is also shown after the CALL routine name):

- CALL MSMQCREATEQUEUE routine - qid (format name representation)
- CALL MSMQPATHTOFORMAT routine - qid
- CALL MSMQINSTTOFORMAT routine - qid
- CALL MSMQHNDLTOFORMAT routine - qid
- CALL MSMQMQP routine - hMap
- CALL MSMQSETPARMS routine - hData

Type Numeric, Input

Example

This example frees a handle and its resources.

CALL MSMQFREE(Handle);

CALL MSMQFREESCONTEXT Routine

Frees the memory that is allocated by the CALL MSMQGETSCONTEXT routine.

Syntax

CALL MSMQFREESCONTEXT(hContext, rc);

Required Arguments

hContext

Specifies the handle to the security context buffer that is allocated by MSMQ.

Type Numeric, Input
Provides the return code from the CALL routine. If an error occurs, then the return code is nonzero. A return code of -1 reflects a SAS internal error. Otherwise, it represents an MSMQ error code. You can use the SAS function SYSMSG() to obtain a textual description of the return code.

**Example**

This example frees the security context buffer.

```sas
length msg $ 200;
rc=0;
CALL MSMQFREESCONTEXT(hContext, rc);
if rc ^= 0 then do;
  put 'MSMQFreeSContext: failed';
  msg = sysmsg();
  put msg;
end;
else put 'MSMQFreeSContext: succeeded';
```

**CALL MSMQGETPARMS Routine**

Retrieves values of SAS variables from a previous MSMQ message that was received by the CALL MSMQRECEIVEMSG routine.

**Syntax**

```sas
CALL MSMQGETPARMS(hMap, rc, parm1<, parm2, parm3,…>);
```

**Required Arguments**

- **hMap**
  
  Specifies the SAS internal map descriptor handle that is obtained from a previous CALL MSMQMAP routine.

  **Type** Numeric, Input

- **rc**
  
  Provides the return code from the CALL routine. If an error occurs, then the return code is nonzero. You can use the SAS function SYSMSG() to obtain a textual description of the return code.

  **Type** Numeric, Output

- **parm**
  
  Specifies one or more parameters that are used to define the values of SAS variables in a message. Initialize the variables appropriately to guarantee that truncation of the returned values does not occur.

  **Type** Numeric or Character, Input
Details
This message is available until the next CALL MSMQRECEIVEMSG routine is performed.

Example
This example gets values of SAS variables from a received message.

```sas
length parm1 parm2 parm3;
length parm4 $ 200;
rc=0;
CALL MSMQGETPARMS(hMap, rc, parm1, parm2, parm3, parm4);
```

CALL MSMQGETQPROP Routine
Retrieves properties for a specific queue.

Syntax
```sas
CALL MSMQGETQPROP(qid, rc, propids, value1<, value2…>);
```

Required Arguments

**qid**
Specifies the queue identifier that represents the format name of the queue.

Type  Numeric, Input

**rc**
Provides the return code from the CALL routine. If an error occurs, then the return code is nonzero. A return code of -1 reflects a SAS internal error. Otherwise, it represents an MSMQ error code. You can use the SAS function SYSMSG() to obtain a textual description of the return code.

Type  Numeric, Output

**propids**
Identifies one or more properties that you want to retrieve. This parameter is a character string with each applicable property separated by a comma. For each property identified by propids, you must provide a value parameter that specifies a variable name to use to hold the returned property value.

The following propids and values are valid:

**AUTHENTICATE**
Retrieves whether the queue accepts only authenticated messages. Initialize the variable appropriately to prevent truncation of the returned value from occurring. The following values are valid:

**NONE**
Specifies the queue accepts either authenticated or non-authenticated messages.
ALWAYS
   Specifies the queue always requires authenticated messages.

BASEPRIORITY
   Retrieves the base priority for all messages that are sent to a public queue.
   The value is a numeric that ranges from -32768 to 32767, where 32767 is the
   highest priority and 0 is the default priority.

CREATE_TIME
   Retrieves the time and date on which the queue was created. The value is a
   numeric that represents the number of seconds elapsed since midnight
   (00:00:00), January 1, 1970 (Coordinated Universal time).

INSTANCE
   Retrieves the queue's identifier (UUID). The value is a character string that
   represents binary data. Initialize the variable appropriately to guarantee that
   truncation of the returned value does not occur.

JOURNAL
   Retrieves if messages are also copied to its journal queue. Initialize the
   variable to a size of at least 32 to guarantee that truncation of the returned
   value does not occur. The following values are valid:

NONE
   Specifies that messages removed from the queue are discarded.

ALWAYS
   Specifies that messages removed from the queue are always stored in its
   journal queue.

JOURNAL_QUOTA
   Retrieves the maximum size (in kilobytes) of the journal queue.

LABEL
   Retrieves a description of the queue. The value is a character string. Initialize
   the variable appropriately to prevent truncation of the returned value from
   occurring.

MODIFY_TIME
   Retrieves the last time the queue's properties were modified. The value is a
   numeric that represents the number of seconds elapsed since midnight
   (00:00:00), January 1, 1970 (Coordinated Universal time).

PATHNAME
   Retrieves the MSMQ pathname of the queue. The value is a character string.
   Initialize the variable appropriately to prevent truncation of the returned value
   from occurring.

PRIV_LEVEL
   Retrieves the privacy level that is required by the queue. The value is a
   character string. Initialize the variable appropriately to prevent truncation of
   the returned value from occurring. The following values are valid:

NONE
   Specifies that the queue accepts only non-private (clear-text) messages.

BODY
   Specifies that the queue accepts only private (encrypted) messages.

OPTIONAL
   Specifies that the queue accepts both private and non-private messages.

QUOTA
   Retrieves the maximum size (in kilobytes) of the queue.
TRANSACTION
Retrieves whether the queue uses MQMQ transactions. The value is a character string. Initialize the variable appropriately to prevent truncation of the returned value from occurring. The following values are valid:

NONE
Specifies that the queue does not accept transaction operations.

ALWAYS
Specifies that all messages that are sent to the queue must always be done through an MSMQ transaction.

TYPE
Retrieves the type of service that is provided by the queue. The value of the TYPE property is a universal unique identifier (UUID) in the form of a character string that represents the binary data. Initialize the variable to a size of at least 32 to guarantee that truncation of the returned value does not occur.

Example
This example gets the queue properties and displays them.

```
length msg $ 200;
length base createt jquota modifyt quota 8;
length auth journal priv trans $ 10;
length inst type $ 32;
length label path $ 80;
rc=0;
CALL MSMQGETQPROP(qid, rc,
"AUTHENTICATE,BASEPRIORITY,CREATE_TIME,INSTANCE,JOURNAL,
JOURNAL_QUOTA,LABEL,MODIFY_TIME,PATHNAME,PRIV_LEVEL,
QUOTA,TRANSACTION>Type",
auth, base, createt, inst, journal, jquota,
label,
modifyt, path, priv, quota, trans, type);
if rc ^= 0 then do;
  put 'MSMQGetQProp: failed';
  msg = sysmsg();
  put msg;
end;
else do;
  put 'MSMQGetQProp: succeeded';
  put 'authenticate is ' auth;
  put 'base priority is ' base;
  /* convert MSMQ create time to SAS datetime format */
  createt = createt + 10*365*24*3600 + 3*24*3600 - 5*3600;
  put 'create time was ' createt datetime.;
  put 'instance identifier is ' inst;
  put 'journal enablement is ' journal;
  put 'journal quota is ' jquota;
  put 'label is ' label;
  /* convert MSMQ modify time to SAS datetime format */
  modifyt = modifyt + 10*365*24*3600 + 3*24*3600 - 5*3600;
  put 'last modification time was ' modifyt datetime.;
```
CALL MSMQGETQSEC Routine

Retrieves the access control security information for the specified queue.

Syntax

CALL MSMQGETQSEC(qid, rc, owner, dacl);

Required Arguments

**qid**

Specifies the queue identifier that represents the format name of the queue.

Type: Numeric, Input

**rc**

Provides the return code from the CALL routine. If an error occurs, then the return code is nonzero. A return code of -1 reflects a SAS internal error. Otherwise, it represents an MSMQ error code. You can use the SAS function SYSMSG() to obtain a textual description of the return code.

Type: Numeric, Output

**owner**

Returns the owner of the queue. Initialize this variable appropriately to guarantee that truncation of the returned value does not occur.

Type: Character, Output

**dacl**

Returns the discretionary access control list for the queue. Initialize this variable appropriately to guarantee that truncation of the returned value does not occur. This parameter is returned in the form of

```
Domain\Account:accessType:Permissions,...
```

where **accessType** is one of the following:

- ALLOW (Permissions allowed)
- DENY (Permissions denied)

**Permissions** is one or more of the following separated by '+':

- Rj (Receive Journal)
- Rq (Receive Message)
- Pq ( Peek Message)
Example
This example gets the queue security properties and displays them.

```plaintext
length msg $ 200;
length owner $ 60;
length dacl $ 200;
rc=0;
CALL MSMQGETQSEC(qid, rc, owner, dacl);
if rc ^= 0 then do;
    put 'MSMQGetQSec: failed';
    msg = sysmsg();
    put msg;
end;
else do;
    put 'MSMQGetQSec: succeeded';
    put 'owner is ' owner;
    put 'dacl is ' dacl;
end;
```

CALL MSMQGETSCONTEXT Routine
Retrieves security information that is needed to authenticate messages.

Syntax

```plaintext
CALL MSMQGETSCONTEXT(certStor, hContext, rc);
```

Required Arguments

`certStor`
Specifies the name of the system certificate store to use to locate the desired external certificate. If NULL, then the internal security certificate that is provided by MSMQ is used. Generally, MY is used. The corresponding registry entry is:

`HKEY_CURRENTUSER\Software\Microsoft\SystemCertificates\MY\Certificates`

Type Character, Input
**hContext**

Returns a handle to the security context buffer that is allocated by MSMQ.

*Type* Numeric, Output

**rc**

Provides the return code from the CALL routine. If an error occurs, then the return code is nonzero. A return code of -1 reflects a SAS internal error. Otherwise, it represents an MSMQ error code. You can use the SAS function SYSMSG() to obtain a textual description of the return code.

*Type* Numeric, Output

---

**Example**

This example gets the security context from internal MSMQ certificate.

```sas
length msg $ 200;
hContext=0;
rc=0;
CALL MSMQGETSCONTEXT("", hContext, rc);
if rc ^= 0 then do;
   put 'MSMQGetSContext: failed';
   msg = sysmsg();
   put msg;
end;
else put 'MSMQGetSContext: succeeded';
```

---

**CALL MSMQHNDLTOFORMAT Routine**

Returns a queue identifier that represents a format name based on its open handle.

**Syntax**

```sas
CALL MSMQHNDLTOFORMAT(hQueue, qid, rc);
```

**Required Arguments**

**hQueue**

Specifies the MSMQ handle to an open queue. This parameter is obtained from a previous CALL MSMQOPENQUEUE routine.

*Type* Numeric, Input

**qid**

Returns the queue identifier that represents the format name of the queue.

*Type* Numeric, Output

**rc**

Provides the return code from the CALL routine. If an error occurs, then the return code is nonzero. A return code of -1 reflects a SAS internal error.
Otherwise, it represents an MSMQ error code. You can use the SAS function SYSMSG() to obtain a textual description of the return code.

**Example**

This example obtains the format name of a queue from a queue handle.

```sas
length msg $ 200;
qid=0;
rc=0;
CALL MSMQHNDLTOFORMAT(hQueue, qid, rc);
if rc ^= 0 then do;
   put 'MSMQHndlToFormat: failed';
   msg = sysmsg();
   put msg;
end;
else put 'MSMQHndlToFormat: succeeded';
```

---

**CALL MSMQINSTTOFORMAT Routine**

Returns a queue identifier that represents a format name based on the instance identifier provided.

**Syntax**

```sas
CALL MSMQINSTTOFORMAT(instance, qid, rc);
```

**Required Arguments**

- **instance**
  - Specifies the universal unique identifier (UUID) instance of the queue.
  - Type: Character, Input

- **qid**
  - Returns the queue identifier that represents the format name of the queue.
  - Type: Numeric, Output

- **rc**
  - Provides the return code from the CALL routine. If an error occurs, then the return code is nonzero. A return code of -1 reflects a SAS internal error. Otherwise, it represents an MSMQ error code. You can use the SAS function SYSMSG() to obtain a textual description of the return code.
  - Type: Numeric, Output

**Example**

This example obtains the format name of a queue from an instance UUID.
CALL MSMQLOCATE Routine

Provides a means of locating a single public queue (or set of public queues) based on a set of criteria.

Syntax

```
CALL MSMQLOCATE(criteria, sortpref, rc, cProps, propids, value1<, value2, ...>);
```

Required Arguments

### criteria
Identifies the criteria to use for locating the queue or queues. The criteria are based on a queue's properties and each property's value. The *criteria* parameter uses the following format:

```
propid:op:value, ...
```

where *propid* is a queue property, *value* is the propid value, and *op* is an operator used as the selection criteria. The *op* parameter can be:

- LT (Less than)
- LE (Less than or equal)
- EQ (Equal)
- NE (Not equal)
- GE (Greater than or equal)
- GT (Greater than)

**Type** Character, Input

### sortpref
Specifies the queue sorting preference. This parameter uses the following format:

```
propid:order, ...
```

where *propid* is a queue property, and *order* is the order preference. The order parameter can be specified as:

- ASCEND (Ascending order)
- DESCEND (Descending order)
**rc**

Provides the return code from the CALL routine. If an error occurs, then the return code is nonzero. A return code of -1 reflects a SAS internal error. Otherwise, it represents an MSMQ error code. You can use the SAS function SYSMSG() to obtain a textual description of the return code.

**cProps**

Returns the number of property values that resulted from the criteria search.

**propids**

Identifies one or more properties that you want to retrieve. This parameter is a character string with each applicable property separated by a comma.

The following propids and values are valid:

**AUTHENTICATE**

Retrieves whether the queue accepts only authenticated messages. Initialize the variable appropriately to prevent truncation of the returned value from occurring. The following values are valid:

- **NONE**
  
  Specifies the queue accepts either authenticated or non-authenticated messages.

- **ALWAYS**
  
  Specifies the queue always requires authenticated messages.

**BASEPRIORITY**

Retrieves the base priority for all messages that are sent to a public queue. The value is a numeric that ranges from -32768 to 32767, where 32767 is the highest priority and 0 is the default priority.

**CREATE_TIME**

Retrieves the time and date on which the queue was created. The value is a numeric that represents the number of seconds elapsed since midnight (00:00:00), January 1, 1970 (Coordinated Universal time).

**INSTANCE**

Retrieves the queue's identifier (UUID). The value is a character string that represents binary data. Initialize the variable to a size of at least 32 to guarantee that truncation of the returned value does not occur.

**JOURNAL**

Queries whether messages are also copied to its journal queue. Initialize the variable to a size of at least 32 to guarantee that truncation of the returned value does not occur. The following values are valid:

- **NONE**
  
  Specifies that messages removed from the queue are discarded.

- **ALWAYS**
  
  Specifies that messages removed from the queue are always stored in its journal queue.

**JOURNAL_QUOTA**

Retrieves the maximum size (in kilobytes) of the journal queue.
LABEL
Retrieves a description of the queue. The value is a character string. Initialize the variable appropriately to prevent truncation of the returned value from occurring.

PATHNAME
Retrieves the MSMQ pathname of the queue. The value is a character string. Initialize the variable appropriately to prevent truncation of the returned value from occurring.

PRIV_LEVEL
Retrieves the privacy level that is required by the queue. The value is a character string. Initialize the variable appropriately to prevent truncation of the returned value from occurring. The following values are valid:

NONE
Specifies that the queue accepts only non-private (clear-text) messages.

BODY
Specifies that the queue accepts only private (encrypted) messages.

OPTIONAL
Specifies that the queue accepts both private and non-private messages.

QUOTA
Retrieves the maximum size (in kilobytes) of the queue.

TRANSACTION
Retrieves whether the queue uses MSMQ transactions. The value is a character string. Initialize the variable appropriately to prevent truncation of the returned value from occurring. The following values are valid:

NONE
 Specifies that the queue does not accept transaction operations.

ALWAYS
 Specifies that all messages that are sent to the queue must always be done through an MSMQ transaction.

TYPE
Retrieves the type of service that is provided by the queue. The value of the TYPE property is a universal unique identifier (UUID) in the form of a character string that represents the binary data. Initialize the variable to a size of at least 32 to guarantee that truncation of the returned value does not occur.

Type Character, Input

Note The number of values specified should be a multiple of propids specified. For example, if you specified two propids and wanted to retrieve these properties for the first three queues that meet the specified criteria, you must specify six (3x2) value parameters in order to retrieve these property values for all of the queues.

Example
This example locates the first three queues with a label Test Queue and returns AUTHENTICATE, PRIV_LEVEL, and PATHNAME properties.

length msg $ 200;
length cProps 8;
length auth1 auth2 auth3 priv1 priv2 priv3 $ 10;
length path1 path2 path3 $ 80;
rc=0;
cProps=0;
CALL MSMQLOCATE("LABEL:EQ:Test Queue", ",", rc, cProps,
"AUTHENTICATE,PRIV_LEVEL,PATHNAME",
auth1, priv1, path1, auth2, priv2, path2, auth3, priv3,
path3);
if rc ^= 0 then do;
   put 'MSMQLocate: failed';
   msg = sysmsg();
   put msg;
   end;
else do;
   put 'MSMQLocate: succeeded';
   if cProps = 0 then put 'no queues were found';
   else do;
      cProps = cProps/3; /* # queues */
      if cProps GE 1 then do;
         put 'queue 1 - authenticate is ' auth1;
         put 'queue 1 - privacy is ' priv1;
         put 'queue 1 - pathname is ' path1;
      end;
      if cProps GE 2 then do;
         put 'queue 2 - authenticate is ' auth2;
         put 'queue 2 - privacy is ' priv2;
         put 'queue 2 - pathname is ' path2;
      end;
      if cProps EQ 3 then do;
         put 'queue 3 - authenticate is ' auth3;
         put 'queue 3 - privacy is ' priv3;
         put 'queue 3 - pathname is ' path3;
      end;
   end;
end;

CALL MSMQMAP Routine
Defines a data map that can be subsequently used on the CALL MSMQSETPARMS or CALL MSMQGETPARMS routine.

Syntax

CALL MSMQMAP(hMap, rc, desc1<, desc2, desc3,...>);

Required Arguments

hMap
Returns the SAS internally generated map descriptor handle.

Type   Numeric, Output
**rc**
Provides the return code from the CALL routine. If an error occurs, then the return code is nonzero. You can use the SAS function SYSMSG() to obtain a textual description of the return code.

**Type** Numeric, Output

**desc**
Specifies descriptor parameters that are used to describe the different data types in a map. Each description (desc1, desc2, ...) defines the data type, an offset from the beginning of the message, and the length of the data. A descriptor has the following format:

\[\text{TYPE}<,\text{OFFSET},\text{LENGTH}>\]

where:

**TYPE** is one of the following:
- CHAR (Character data)
- SHORT (Short binary)
- LONG (Long binary)
- DOUBLE (Floating point double)

**OFFSET**
Specifies the offset from the beginning of the message. This property is optional, so by default the data is not aligned (data starts at next available position in message).

**LENGTH**
Specifies the length of the data being represented. This property is optional in most cases. The only time length is required is when setting up to receive character data. Specifying length for numeric data is ignored because length is implicitly defined.

**Type** Character, Input

**Note**
Type coercion is performed transparently when you put SAS variables into an MSMQ message (CALL MSMQSETPARMS routine) and also when you get SAS variables from an MSMQ message (CALL MSMQGETPARMS routine). That is, if the data that you are sending or receiving is a different type than the SAS variable itself, then the data is coerced into the appropriate data type.

**Example**
This example defines a map to use to send and receive a message with a short, a long, a double, and a character string. No alignment is specified for any data type, and strings will always be 200 characters in length (blank padded).

```sas
hMap=0;
rc=0;
desc1="SHORT";
desc2="LONG";
desc3="DOUBLE";
desc4="CHAR,,200"
CALL MSMQMAP(hMap, rc, desc1, desc2, desc3, desc4);
```
CALL MSMQOPENQUEUE Routine

Opens a queue for sending message to the queue or for reading its messages.

Syntax

```
CALL MSMQOPENQUEUE(qid, access, shareMode, hQueue, rc);
```

Required Arguments

- **qid**: Specifies the queue identifier that represents the format name of the queue to be opened.
  - Type: Numeric, Input

- **access**: Indicates the level of access that users have to the messages in the queue being opened. The following values are valid:
  - **PEEK**: Specifies that messages can only be read.
  - **SEND**: Specifies that messages can be sent only to the queue.
  - **RECEIVE**: Specifies that messages can be read and removed from the queue.
  - Type: Character, Input

- **shareMode**: Specifies how the queue is shared. The following values are valid:
  - **SHARE**: Specifies that the queue is available to everyone.
  - **DENY_SHARE**: Specifies that the process using this CALL routine is the only one that can receive messages from this queue. If the queue is already opened for receiving messages by another process, then this call will fail.
  - Type: Character, Input

- **hQueue**: Returns the MSMQ handle of the opened queue. This handle is used by subsequent CALL routines to identify and access the queue.
  - Type: Numeric, Output

- **rc**: Provides the return code from the CALL routine. If an error occurs, then the return code is nonzero. A return code of -1 reflects a SAS internal error. Otherwise, it represents an MSMQ error code. You can use the SAS function SYSMSG() to obtain a textual description of the return code.
Example
This example opens a queue for sending messages.

```plaintext
length msg $ 200;
hQueue=0;
rc=0;
CALL MSMQOPENQUEUE(qid, "SEND", "SHARE", hQueue, rc);
if rc ^= 0 then do;
  put 'MSMQOpenQueue: failed';
  msg = sysmsg();
  put msg;
end;
else put 'MSMQOpenQueue: succeeded';
```

CALL MSMQPATHTOFORMAT Routine

Returns a queue identifier (qid) handle that represents the format name of the desired queue.

Syntax

```plaintext
CALL MSMQPATHTOFORMAT(pathName, qid, rc);
```

Required Arguments

**pathName**

Represents the queue's pathname or actual format name of the queue, if known. If an MSMQ pathname is used to represent the queue, then it is converted to an MSMQ format name. Possible **pathName** representations are as follows:

- Public queue: `machineName\QueueName`
- Public queue's journal: `machineName\QueueName;Journal`
- Private queue: `machineName\PRIVATE$\QueueName`
- Private queue's journal: `machineName\PRIVATE$\QueueName;Journal`
- Machine journal queue: `machineName\JOURNAL`
- Machine deadletter queue: `machineName\DEADLETTER`
- Machine transaction deadletter queue: `machineName\DEADXACT`

**Note:** `machineName` can be substituted with `.' to designate the local machine. If the actual format name of the queue is known, then this call can be used to transform it into the expected unicode string.

- **Possible format name representations are as follows:**
  - Public queue: `public=QueueGUID`
  - Public queue's journal: `public=QueueGUID;JOURNAL`
Private queue: private=machineGUID\QueueNumber
Private queue’s journal: private=machineGUID\QueueNumber;JOURNAL
Direct public queue: direct=AddressSpec\QueueName
Direct private queue: direct=AddressSpec\PRIVATE$\QueueName
where AddressSpec is of the form protocol:address (For example, tcp:10.26.1.177)
Machine journal queue: machine=machineGUID;JOURNAL
Machine deadletter queue: machine=machineGUID;DEADLETTER
Machine transaction deadletter queue: machine=machineGUID;DEADXACT
Foreign queue: connector=ForeignCNGUID
Foreign transaction queue: connector=ForeignCNGUID:XACTONLY

qid
Returns the queue identifier that represents the format name of the queue.

rc
Provides the return code from the CALL routine. If an error occurs, then the
return code is nonzero. A return code of -1 reflects a SAS internal error.
Otherwise, it represents an MSMQ error code. You can use the SAS function
SYSMSG() to obtain a textual description of the return code.

Example
This example obtains the format name of a queue from the pathname.

length msg $ 200;
qid=0;
rc=0;
CALL MSMQPATHTOFORMAT("pcpad\testq", qid, rc);
if rc ^= 0 then do;
   put 'MSMQPathToFormat: failed';
   msg = sysmsg();
   put msg;
end;
else put 'MSMQPathToFormat: succeeded';

CALL MSMQRECEIVEMSG Routine
Reads a message from the queue.
Syntax

CALL MSMQRECEIVEMSG(hQueue, timeout, action, hCursor, transObj, rc <, propids, value1, value2,…..>);

Required Arguments

**hQueue**
- Specifies the MSMQ handle to an open queue. This parameter is obtained from a previous CALL MSMQOPENQUEUE routine.
  - Type: Numeric, Input

**timeout**
- Specifies the amount of time (in milliseconds) to wait for a message to be received from the queue. If you want to wait indefinitely for the message to be received, then set the time-out parameter to -1.
  - Type: Numeric, Input

**action**
- Determines how and where the message is read from the queue. This parameter is also used to determine whether the message is removed after reading.
  - Possible valid values:
    - **RECEIVE**: Reads the message at the current cursor location and removes it from the queue.
    - **PEEK_CURRENT**: Reads a message at the current cursor location but does not remove it from the queue. The cursor remains at the current message. If the **hCursor** parameter is 0, then the queue's cursor can point only to the first message in the queue.
    - **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. A cursor must already be created (by using the CALL MSMQCREATECURSOR routine) before using this CALL routine. (hCursor = 0 is not allowed.)
  - Type: Character, Input

**hCursor**
- Specifies the handle to a cursor that is used for looking at messages in the queue. The CALL MSMQCREATECURSOR routine is used to create a cursor and obtain its handle.
  - Type: Numeric, Input

**transObj**
- Specifies the transaction object that is obtained from a previous CALL MSMQBEGINTRANS routine. If this value is set to zero, then it is assumed that this operation will not be part of a transaction.
  - Type: Numeric, Input
rc
Provides the return code from the CALL routine. If an error occurs, then the return code is nonzero. A return code of -1 reflects a SAS internal error. Otherwise, it represents an MSMQ error code. You can use the SAS function SYSMSG() to obtain a textual description of the return code.

Type   Numeric, Output

Optional Argument

**propids**
Identifies one or more message properties that affects the message being received from the queue. This parameter is a character string with each applicable property separated by a comma. You must provide a value parameter for each property specified in the propids string. Each property ID in the propids string is associated positionally with a value parameter. The CALL routine returns the corresponding property value into each value parameter.

The following receive message properties and values are valid:

**ACKNOWLEDGE**
Retrieves the type of acknowledgment messages that MSMQ posts when the message was sent. Initialize the variable appropriately to prevent truncation of the retrieved value from occurring. Possible acknowledge types are as follows:

**NONE**
Specifies no acknowledgment messages are posted.

**FULL_REACH_QUEUE**
Specifies that positive and negative acknowledgments are posted, indicating whether the message reaches the queue.

**FULL_RECEIVE**
Specifies that positive and negative acknowledgments are posted, depending on whether the message is retrieved from the queue before its time-to-be-received timer expires.

**NACK_REACH_QUEUE**
Specifies that negative acknowledgments are posted when a message cannot reach the queue.

**NACK_RECEIVE**
Specifies that negative acknowledgments are posted when a message cannot be retrieved from the queue.

**ADMIN_QUEUE**
Retrieves the queue used for MSMQ-generated acknowledgment messages. This value is a character string that represents the pathname of the administration queue. You can use the CALL MSMQPATHTOFORMAT routine to obtain a queue identifier for this queue. Initialize the variable appropriately to prevent truncation of the returned value from occurring.

**APPSPECIFIC**
Retrieves the application-generated information. The value is numeric, and the default is 0.

**ARRIVEDTIME**
Retrieves the time at which the message arrived at the queue. The value is a numeric that represents the number of seconds elapsed since midnight (00:00:00), January 1, 1970 (Coordinated Universal time).
AUTHENTICATED
Retrieves whether the message was authenticated. The following values are valid:

- 0: Message is not authenticated.
- 1: Message is authenticated.

BODY
Specifies whether the message body should be received. The following values are valid:

- 0: Specifies not to retrieve the body of the message.
- 1 (default): Specifies to retrieve the body of the message

BODY_SIZE
Retrieves the actual size of the message body. The body size is a numeric value.

BODY_TYPE
Retrieves the type of body the message contains. The value is numeric.

CLASS
Retrieves the class of the message. The value is a numeric.

CORRELATIONID
Retrieves the correlation identifier of the message. The value is a character string that represents binary data. Initialize the variable to a size of at least 40 to guarantee that truncation of the returned value does not occur.

DELIVERY
Retrieves how the message is delivered. Initialize the variable appropriately to prevent truncation of the returned value from occurring. The following values are valid:

- EXPRESS: Specifies faster, non-guaranteed delivery.
- RECOVERABLE: Specifies guaranteed delivery.

DEST_QUEUE
Retrieves the target queue of the message. This value is a character string that represents the pathname of the destination queue. You can use the CALL MSMQPATHTOFORMAT routine to obtain a queue identifier for this queue. Initialize the variable appropriately to prevent truncation of the returned value from occurring.

JOURNAL
Retrieves journal enablement. Initialize the variable appropriately to prevent truncation of the returned value from occurring. The following values are valid:

- NONE (default): Specifies the message is not kept in the originating machine's journal queue.
- JOURNAL: Specifies the message is kept in the originating machine's journal queue.
- DEADLETTER: Specifies the message is kept in a dead letter queue if it cannot be delivered.
Note: A combination can be specified by separating each value with a comma (for example, JOURNAL,DEADLETTER.)

**LABEL**
Retrieves a label for the message. The label value is a character string. Initialize the variable appropriately to prevent truncation of the returned value from occurring.

**MSGID**
Retrieves MSMQ-generated identifier of the message. The value is a character string that represents binary data. Initialize the variable to a size of at least 40 to guarantee that truncation of the returned value does not occur.

Note: This value is returned as a binary string. MSMQ Explorer displays the message identifier as a UUID concatenated with a sequence number.

**PRIORITY**
Retrieves the message’s priority. The value is a numeric between 0 and 7. The highest priority is 7, and the default priority is 3.

**PRIV_LEVEL**
Retrieves the privacy level of the message. Initialize the variable appropriately to prevent truncation of the returned value from occurring. The following values are valid:

- **PUBLIC**
  Specifies the message is to be sent as clear-text.

- **PRIVATE**
  Specifies end-to-end encryption of the message body.

**RESP_QUEUE**
Retrieves the pathname of the queue where application-generated response messages are returned. The value is a character string that represents the pathname of the response queue. You can use the CALL MSMQPATHTOFORMAT routine to obtain a queue identifier for this queue. Initialize the variable appropriately to prevent truncation of the returned value from occurring.

**SENDER_CERT**
Retrieves the certificate that was used to authenticate the message. This value is a character string. If an external certificate was used to authenticate the message, the information that is returned can be used to verify who sent the message (subject).

**SENDERID**
Retrieves who sent the message. The value is a character string.

**SENTTIME**
Retrieves the time at which the message was sent. The value is a numeric that represents the number of seconds elapsed since midnight (00:00:00), January 1, 1970 (Coordinated Universal time).

**SRC_MACHINE_ID**
Retrieves the UUID of the computer where the message was sent. This value is a UUID in the form of a character string that represents the binary data. Initialize the variable to a size of at least 32 to guarantee that truncation of the returned value does not occur.
TIME_TO_BE_RECEIVED
Retrieves the total time (in seconds) that the message is to be available. The value is a numeric with a default of infinity.

TIME_TO_REACH_QUEUE
Retrieves time limit (in seconds) for the message to reach the queue.

TRACE
Retrieves where report messages are sent when tracing a message. Initialize the variable appropriately to guarantee that truncation of the returned value does not occur. The following values are valid:

NONE
Specifies no tracing for this message.

REPORT
Specifies report messages are to be sent to the report queue that is specified by the source queue manager.

VERSION
Retrieves the version of MSMQ that is used to send the message. The value is a numeric.

Type: Character, Input

Details
When reading messages, you can either peek at or retrieve them from the queue. The message is retrieved into an internal SAS buffer at which time you should use the CALL MSMQGETPARMS routine to set SAS variables (parameters) to that data.

Example
This example receives a message.

```sas
length msg $ 200;
length arrivet auth size respq sentt 8;
length correlid msgid $ 40;
length label $ 80;
rc=0;
hCursor=0;
transobj=0;
CALL MSMQRECEIVEMSG(hQueue, 0, "RECEIVE", hCursor, transobj, rc, 
  "ARRIVEDTIME,AUTHENTICATED,BODY_SIZE,CORRELATIONID, 
  LABEL,MSGID,RESP_QUEUE,SENTTIME", 
  arrivet, auth, size, correlid, label, msgid, respq, sentt);
if rc ^= 0 then do;
  put 'MSMQReceiveMsg: failed';
  msg = sysmsg();
  put msg;
  end;
else do;
  put 'MSMQReceiveMsg: succeeded';
  /* convert MSMQ arrived time to SAS datetime format */
  arrivet = arrivet + 10*365*24*3600 + 3*24*3600 - 5*3600;
  put 'arrived time is' arrivet datetime.;
  if auth = 1 then put 'message was authenticated';
  else put 'message was not authenticated';
```
put 'message body size is ' size;
put 'correlation id is ' correlid;
put 'label is ' label;
put 'msg id is ' msgid;
put 'resp_queue qid handle is ' respq;
/* convert MSMQ sent time to SAS datetime format */
sentt = sentt + 10*365*24*3600 + 3*24*3600 - 5*3600;
put 'sent time was' sentt datetime.;
end;

CALL MSMQRELEASETRANS Routine

Releases an internal MSMQ transaction object, thereby allowing MSMQ to release the associated resources.

Syntax

CALL MSMQRELEASETRANS(transObj, rc);

Required Arguments

transObj

Specifies the transaction object that is obtained from a previous CALL MSMQBEGINTRANS routine.

Type    Numeric, Input

rc

Provides the return code from the CALL routine. If an error occurs, then the return code is nonzero. A return code of -1 reflects a SAS internal error. Otherwise, it represents an MSMQ error code. You can use the SAS function SYSMSG() to obtain a textual description of the return code.

Type    Numeric, Output

Example

This example releases a transaction unit of work.

length msg $ 200;
rc=0;
CALL MSMQRELEASETRANS(transobj, rc);
if rc ^= 0 then do;
   put 'MSMQReleaseTrans: failed';
   msg = sysmsg();
   put msg;
end;
else put 'MSMQReleaseTrans: succeeded';
CALL MSMQSENDSMSG Routine

Sends a message to the specified queue.

Syntax

CALL MSMQSENDSMSG(hQueue, hData, transObj, rc, propids, value1<, value2,...>);

Required Arguments

**hQueue**
- Specifies the MSMQ handle to an open queue. This parameter is obtained from a previous CALL MSMQOPENQUEUE routine.
  - Type: Numeric, Input

**hData**
- Specifies the SAS internal data descriptor handle that is obtained from a previous CALL MSMQSETPARMS routine. If this value is set to zero, then it is assumed that no data will accompany this message.
  - Type: Numeric, Input

**transObj**
- Specifies the transaction object obtained from a previous CALL MSMQBEGINTRANS routine. If this value is set to zero, then it is assumed that this operation will not be part of a transaction.
  - Type: Numeric, Input

**rc**
- Provides the return code from the CALL routine. If an error occurs, then the return code is nonzero. A return code of -1 reflects a SAS internal error. Otherwise, it represents an MSMQ error code. You can use the SAS function SYSMSG() to obtain a textual description of the return code.
  - Type: Numeric, Output

**propids**
- Identifies one or more message properties that affects the message being sent. This parameter is a character string with each applicable property separated by a comma. You must provide a value parameter for each property specified in the propids string. Each property ID in the propids string is associated positionally with a value parameter.

Note: All values are inputs to the CALL MSMQSENDSMSG routine except MSGID which returns a message identifier.

The following send message properties and values are valid:
ACKNOWLEDGE
Specifies the type of acknowledgment messages that MSMQ posts when the message is sent. A positive acknowledgment indicates the message sent was received successfully. A negative acknowledgment indicates the message was not received.

Possible acknowledge types are as follows:

NONE (default)
Specifies no acknowledgment messages are posted.

FULL REACH QUEUE
Specifies that positive and negative acknowledgments are posted, indicating whether the message reaches the queue.

FULL RECEIVE
Specifies that positive and negative acknowledgments are posted, indicating whether the message is retrieved from the queue.

NACK REACH QUEUE
Specifies that negative acknowledgments are posted when a message cannot reach the queue.

NACK RECEIVE
Specifies that negative acknowledgments are posted when a message cannot be retrieved from the queue.

ADMIN QUEUE
Specifies the pathname of the queue that is used for MSMQ-generated acknowledgment messages. The value is a character string that represents the pathname of the administration queue.

APPSPECIFIC
Specifies application-generated information. The value is numeric and the default is 0.

AUTH_LEVEL
Specifies whether the message needs to be authenticated.

The following AUTH_LEVEL types are valid:

NONE (default)
Specifies that no authentication is necessary. (Messages are not signed.)

ALWAYS
Specifies that messages are always signed and authenticated by the destination queue manager.

BODY_TYPE
Specifies the type of body the message contains. The value is numeric and is defined by the application and must be coordinated between the sending and receiving portions of the application. The default value is 0.

CORRELATIONID
Specifies the correlation identifier of the message. The value is a character string that represents binary data.

DELIVERY
Specifies how the message is delivered. The following values are valid:

EXPRESS (default)
Specifies faster, non-guaranteed delivery.

RECOVERABLE
Specifies guaranteed delivery.
ENCRIPTION_ALG
   Specifies the encryption algorithm that is used to encrypt the message body of a private message. Possible values are as follows:
   - RC2 (Block cipher) (Default)
   - RC4 (Stream cipher)

HASH_ALG
   Specifies the hashing algorithm that is used when authenticating messages. The following values are valid:
   - MD2
     Message Digest 2 Algorithm
   - MD4
     Message Digest 4 Algorithm
   - MD5 (default)
     Message Digest 5 Algorithm

JOURNAL
   Specifies whether the message should be kept in a machine journal, sent to a dead letter queue, or neither. The following values are valid:
   - NONE (default)
     Specifies the message is not kept in the originating machine's journal queue.
   - JOURNAL
     Specifies the message is kept in the originating machine's journal queue.
   - DEADLETTER
     Specifies the message is kept in a dead letter queue if it cannot be delivered.

   Note: A combination can be specified by separating each value with a comma (for example, JOURNAL,DEADLETTER.)

LABEL
   Specifies a label for the message. The default is a blank label ( ).

MSGID
   Specifies MSMQ-generated identifier of the message. The value is a character string that represents binary data. Initialize the variable to a size of at least 40 to guarantee that truncation of the returned value does not occur.

   Note: This value is returned as a binary string. MSMQ Explorer displays the message identifier as a UUID concatenated with a sequence number.

PRIORITY
   Specifies the message's priority. The value is a numeric between 0 and 7. The highest priority is 7, and the default priority is 3.

PRIV_LEVEL
   Specifies the privacy level of the message. The following values are valid:
   - PUBLIC (default)
     Specifies the message is to be sent as clear-text.
   - PRIVATE
     Specifies end-to-end encryption of the message body.
RESP_QUEUE
Specifies the pathname of the queue where application-generated response messages are returned. The value is a character string that represents the pathname of the response queue.

SECURITY_CONTEXT
Specifies security information that MSMQ uses to authenticate messages. The value is the handle to the security context buffer that is returned from MSMQGETCONTEXT.

SENDER_CERT
Specifies the name of the system certificate store to use in order to locate external certificates during the authentication process. Generally, MY is used. For example, if a value of MY is used, the registry location used to retrieve the system certificate is as follows:

HKEY_CURRENTUSER\Software\Microsoft\SystemCertificates\MY\Certificates

TIME_TO_BE_RECEIVED
Specifies the total time (in seconds) that the message is to be available. The default value is infinite.

TIME_TO_REACH_QUEUE
Specifies time limit (in seconds) for the message to reach the queue.

TRACE
Specifies where report messages are sent when tracing a message. The following values are valid:

NONE (default)
Specifies no tracing for this message.

REPORT
Specifies report messages are to be sent to the report queue that is specified by the source queue manager.

Note: The BODY message property is set internally, based on whether data is present.

Type  Character, Input

Example
This example sends a message.

```
length msg $ 200;
rc=0;
transobj=0;

CALL MSMQSENDMSG(hQueue, hData, transobj, rc,
    "AUTH_LEVEL,APPSPECIFIC,CORRELATIONID,LABEL,PRIV_LEVEL,RESP_QUEUE",
    "ALWAYS", 999, "01203405060708980A0B0C0D0E0F1011121314",
    "Secret test message", "PRIVATE", "mypc\respq");
if rc ^= 0 then do;
    put 'MSMQSendMsg: failed';
    msg = sysmsg();
    put msg;
end;
else put 'MSMQSendMsg: succeeded';
```
CALL MSMQSETPARMS Routine

Creates a data descriptor that describes the actual SAS variables along with an associated data mapping. This data descriptor can then be used on a subsequent CALL MSMQSENDMSG routine.

Syntax

CALL MSMQSETPARMS(hData, hMap, rc, parm1<, parm2, parm3,…>);

Required Arguments

**hData**

Returns the SAS internal data descriptor handle that is generated.

*Type* Numeric, Output

**hMap**

Specifies the SAS internal map descriptor handle that is obtained from a previous CALL MSMQMAP routine. If set to zero, then no external defined mapping is assumed and therefore, all data is mapped according to SAS internal representations. That is, all numerics are mapped as doubles and all strings are mapped as character data of the current string length.

*Type* Numeric, Input

**rc**

Provides the return code from the CALL routine. If an error occurs, then the return code is nonzero. You can use the SAS function SYSMSG() to obtain a textual description of the return code.

*Type* Numeric, Output

**parm**

Specifies one or more parameters that are used to define the values of SAS variables in a message.

*Type* Numeric or Character, Input

Example

This example sets values of SAS variables into a message.

```
hData=0;
rc=0;
parm1=100;
parm2=9999;
parm3=9999.9999;
parm4="This is a test."
CALL MSMQSETPARMS(hData, hMap, rc, parm1, parm2, parm3, parm4);
```
CALL MSMQSETQPROP Routine
Sets the properties of a specific queue.

Syntax
CALL MSMQSETQPROP(qid, rc, propids, value1<, value2,…>);

Required Arguments

qid
Specifies the queue identifier that represents the format name of the queue.
Type Numeric, Input

rc
Provides the return code from the CALL routine. If an error occurs, then the return code is nonzero. A return code of -1 reflects a SAS internal error. Otherwise, it represents an MSMQ error code. You can use the SAS function SYSMSG() to obtain a textual description of the return code.
Type Numeric, Output

propids
Identifies one or more properties that you want to set. This parameter is a character string with each applicable property separated by a comma. For each property identified by propids, you must provide a value parameter that specifies the value to use to set the property. The following propids and values are valid:

AUTHENTICATE
Specifies whether the queue accepts only authenticated messages. The following values are valid:
NONE
Specifies the queue accepts either authenticated or non-authenticated messages.
ALWAYS
Specifies the queue always requires authenticated messages.

BASEPRIORITY
Specifies the base priority for all messages that are sent to a public queue. The value is a numeric that ranges from -32768 to 32767, where 32767 is the highest priority and 0 is the default priority.

JOURNAL
Specifies if messages are also copied to its journal queue. The following values are valid:
NONE
Specifies that messages removed from the queue are discarded.
ALWAYS
Specifies that messages removed from the queue are always stored in its journal queue.
JOURNAL_QUOTA
Specifies the maximum size (in kilobytes) of the journal queue.

LABEL
Specifies a description of the queue. The value is a character string.

PRIV_LEVEL
Specifies the privacy level that is required by the queue. The value is a character string. The following values are valid:

NONE
Specifies that the queue accepts only non-private (clear-text) messages.

BODY
Specifies that the queue accepts only private (encrypted) messages.

OPTIONAL
Specifies that the queue accepts both private and non-private messages.

QUOTA
Specifies the maximum size (in kilobytes) of the queue.

TYPE
Specifies the type of service that is provided by the queue. The value of the TYPE property is a universal unique identifier (UUID) in the form of a character string that represents the binary data.

Type  Character, Input

Example
This example sets the queue properties.

length msg $ 200;
rc=0;
CALL MSMQSETQPROP(qid, rc, "AUTHENTICATE,BASEPRIORITY,JOURNAL,
JOURNAL_QUOTA,LABEL,PRIV_LEVEL,QUOTA,
TYPE", "ALWAYS", 1, "ALWAYS", 32767,
"New Label", "BODY", 32767,
"0A0B0C0D0E0F0102030405060708090A");

if rc ^= 0 then do;
  put 'MSMQSetQProp: failed';
  msg = sysmsg();
  put msg;
end;
else put 'MSMQSetQProp: succeeded';

CALL MSMQSETQSEC Routine
Sets the access control information for a specified queue.

Syntax
CALL MSMQSETQSEC(qid, rc<, owner><, dacl>);
Required Arguments

**qid**
Specifies the queue identifier that represents the format name of the queue.

**Type** Numeric, Input

**rc**
Provides the return code from the CALL routine. If an error occurs, then the return code is nonzero. A return code of -1 reflects a SAS internal error. Otherwise, it represents an MSMQ error code. You can use the SAS function SYSMSG() to obtain a textual description of the return code.

**Type** Numeric, Output

Optional Arguments

**owner**
Identifies the owner of the queue. This parameter must be specified as Domain \Account.

**Type** Character, Input

**dacl**
Specifies the discretionary access control list for the queue. This parameter must be specified in the form of

\textit{Domain\Account:accessType:Permissions,...}

where \textit{accessType} is one of the following:

- **ALLOW**
  Permissions allowed

- **DENY** (See the following note.)
  Permissions denied

\textbf{Note:} Windows NT 4.0 supports DENY access control entries but cannot edit security information that uses them. Therefore, this access type is not recommended until Windows NT 5.0 or later.

**permissions** is one or more of the following, separated by ‘+’:

- Rj (Receive Journal)
- Rq (Receive Message)
- Pq (Peek Message)
- Sq (Send Message)
- Sp (Set Properties)
- Gp (Get Properties)
- D (Delete Queue)
- Pg (Get Permissions)
- Ps (Set Permissions)
- O (Take Ownership)
Example

This example sets the queue security properties to allow NTDOMAIN\User6 to Receive Messages (Rq), Get Properties (Gp), and Get Permissions (Pg).

```
length msg $ 200;
rc=0;
CALL MSMQSETQSEC(qid, rc, "", "NTDOMAIN\User6:ALLOW:Rq+Gp+Pg");
if rc ^= 0 then do;
   put 'MSMQSetQSec: failed';
   msg = sysmsg();
   put msg;
end;
else put 'MSMQSetQSec: succeeded';
```
# Using the SAS Common Messaging Interface

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Common Messaging Interface</td>
<td>166</td>
</tr>
<tr>
<td>Writing Applications Using the Common Messaging Interface</td>
<td>166</td>
</tr>
<tr>
<td>Introduction to Writing Applications with the Common Messaging Interface</td>
<td>166</td>
</tr>
<tr>
<td>Administrator Programs</td>
<td>167</td>
</tr>
<tr>
<td>User Programs</td>
<td>168</td>
</tr>
<tr>
<td>Using TIBCO Rendezvous with the SAS Common Messaging Interface</td>
<td>169</td>
</tr>
<tr>
<td>Overview of Using TIBCO Rendezvous with the SAS Common Messaging Interface</td>
<td>169</td>
</tr>
<tr>
<td>Rendezvous Certified Message Delivery (Rendezvous-CM)</td>
<td>169</td>
</tr>
<tr>
<td>TIBCO Rendezvous Coding Example</td>
<td>170</td>
</tr>
<tr>
<td>TIBCO Rendezvous Certified Messaging Coding Examples</td>
<td>173</td>
</tr>
<tr>
<td>Example 1: Sending and Receiving Messages in the Same DATA Step</td>
<td>173</td>
</tr>
<tr>
<td>Example 2: Sending and Receiving Messages in Separate DATA Steps</td>
<td>176</td>
</tr>
<tr>
<td>Using a Repository with Application Messaging</td>
<td>180</td>
</tr>
<tr>
<td>Using the SAS Registry with the Common Messaging Interface</td>
<td>181</td>
</tr>
<tr>
<td>Overview of Using the SAS Registry</td>
<td>181</td>
</tr>
<tr>
<td>Using the SAS Registry Editor</td>
<td>181</td>
</tr>
<tr>
<td>Writing Applications to Access the SAS Registry</td>
<td>181</td>
</tr>
<tr>
<td>Attachment Layout for WebSphere MQ and MSMQ</td>
<td>184</td>
</tr>
<tr>
<td>Attachment Layout for TIBCO Rendezvous</td>
<td>188</td>
</tr>
<tr>
<td>Overview of Attachment Layout for TIBCO Rendezvous</td>
<td>188</td>
</tr>
<tr>
<td>Data Message Layout</td>
<td>188</td>
</tr>
<tr>
<td>Data Set Attachment Layout</td>
<td>189</td>
</tr>
<tr>
<td>External File Attachment Layout</td>
<td>189</td>
</tr>
<tr>
<td>Message Data - &quot;MSG&quot; or &quot;DATA&quot;</td>
<td>190</td>
</tr>
<tr>
<td>Attachment Header - &quot;HDR&quot;</td>
<td>191</td>
</tr>
<tr>
<td>Data Set Definition - &quot;DAT&quot;</td>
<td>193</td>
</tr>
<tr>
<td>Variable Definition - &quot;VAR&quot;</td>
<td>194</td>
</tr>
<tr>
<td>Data Set Observations - &quot;ATO&quot;</td>
<td>195</td>
</tr>
<tr>
<td>Data Set Index - &quot;ATI&quot;</td>
<td>196</td>
</tr>
<tr>
<td>Data Set Integrity Constraints - &quot;ATC&quot;</td>
<td>197</td>
</tr>
<tr>
<td>External File Descriptor - &quot;FDC&quot;</td>
<td>199</td>
</tr>
<tr>
<td>Text File Attachment - &quot;ATX&quot;</td>
<td>200</td>
</tr>
<tr>
<td>Binary File Attachment - &quot;ATB&quot;</td>
<td>200</td>
</tr>
<tr>
<td>Last Message of Attachment - &quot;LST&quot;</td>
<td>200</td>
</tr>
<tr>
<td>Attachment Error Handling</td>
<td>201</td>
</tr>
</tbody>
</table>
Common Messaging Interface

The SAS Common Messaging Interface provides the following:

- a seamless environment for writing applications that access message queues of the IBM WebSphere MQ (previously named MQSeries), Microsoft MSMQ, and TIBCO Rendezvous transports
- a way to use the local SAS registry to store and retrieve messaging information

The common interface to WebSphere MQ, MSMQ, and Rendezvous enables your application programs to interact in a consistent manner that is independent of your transport.

This section describes the use of the interface and provides reference information for each SAS CALL routine. For the CALL routine reference, see “Common Message Queueing CALL Routines” on page 205.

Writing Applications Using the Common Messaging Interface

Introduction to Writing Applications with the Common Messaging Interface

Two general types of programs can use the common messaging interface. One uses the interface to administer information about the message transports. Another uses the interface to send and receive messages between applications. These two types of programs are discussed in the sections below.

Note: The SAS®9 Common Messaging Interface uses the SAS®9 data set format by default. In order to send and receive SAS Release 8 data sets, you must include the "ATTACH_VERSION=VERSION_8" option in the data set option list on the SENDMESSAGE call. If you do not use the "ATTACH_VERSION=VERSION_8" option on the SENDMESSAGE call, then received data sets are stored in the SAS®9 format. If you might be sending data sets to another SAS session that is running SAS Release 8.2 or earlier, then use the ATTACH_VERSION= option to exchange data sets in a format that can be interpreted by both applications.
Administrator Programs

SAS programs can use the common messaging interface in order to administer the information in the repository for the queues. The goal of such an administrator program is to encapsulate all information about the queues so that all other programs in the application can focus on using the queues rather than configuring them. This not only simplifies the other programs, but also makes the queues easier to administer by having all of this information in one location.

Administrator programs perform general functions, such as the following:

- defining the transport-specific details that are required by the queue. The available transports are MQSeries (WebSphere MQ), MSMQ, Rendezvous, or Rendezvous-CM.
- setting aliases for new transports and queues and retrieving aliases for existing ones.
- retrieving the properties of a queue.
- defining and retrieving maps to data descriptors that identify the data type, offset, and length.
- setting and retrieving dynamic creation queue models for the MSMQ transport.
- setting and retrieving transport definition models for Rendezvous (optional) and Rendezvous-CM (required).

The following SAS CALL routines are used to administer the information repository:

- CALL SETALIAS routine
- CALL SETMAP routine
- CALL SETMODEL routine
- CALL GETALIAS routine
- CALL GETMAP routine
- CALL GETMODEL routine
- CALL GETQUEUEPROPS routine

Other functions of the administration process include removing any unneeded information in the repository. This encompasses functions such as the following:

- deleting a transport or queue alias definition
- deleting a data descriptor definition map
- deleting a dynamic or transport model definition

The following SAS CALL routines are used to administer these aspects of the information repository:

- CALL DELETALIAS routine
- CALL DELETEMAP routine
- CALL DELETEMODEL routine
User Programs

This section describes how a SAS program can use the common messaging interface in order to access message queues to send and receive messages to other programs. The common interface alleviates the need for these user programs to use transport-specific code. This makes the user programs less vulnerable to changes in the queue's attributes. The programs interact with each queue in a consistent matter, independent of the transport.

User programs perform general functions such as the following:

- initializing the type of transport and obtaining a unique identifier
- opening an existing queue by using a known transport identifier
- sending messages to a queue by using a unique queue identifier
- receiving messages (and possibly attachments) from a queue
- parsing the message
- getting attachments that are associated with a message (if necessary)
- copying any desired attachments to local storage
- closing all queues upon completion of the program tasks
- terminating transports that are initialized by the program

The following SAS CALL routines are the basis for initializing or terminating a transport, opening or closing a queue, and sending or receiving messages and attachments:

- CALL INIT routine
- CALL TERM routine
- CALL OPENQUEUE routine
- CALL CLOSEQUEUE routine
- CALL SENDMESSAGE routine
- CALL RECEIVEMESSAGE routine
- CALL PARSEMESSAGE routine
- CALL GETATTACHMENT routine
- CALL ACCEPTATTACHMENT routine

In addition, user programs can perform transaction processing on transaction queues. Such functions include the following:

- creating a transaction object in order to begin progressing
- committing or canceling work that is performed by using a transaction object
- releasing a transaction object and any resource that is associated with it

The following SAS CALL routines are provided for applications that require transaction processing:

- CALL BEGINTRANSMISSION routine
- CALL COMMIT routine
CALL ABORT routine

CALL FREETRANSACTION routine

Using TIBCO Rendezvous with the SAS Common Messaging Interface

Overview of Using TIBCO Rendezvous with the SAS Common Messaging Interface

SAS Integration Technologies supports the message delivery features of TIBCO Rendezvous Release 7.5.4 and later.

TIBCO Rendezvous is a leading messaging middleware product from TIBCO Software, Inc. Like IBM WebSphere MQ (previously named MQSeries) and Microsoft MSMQ, TIBCO Rendezvous makes it easy to create distributed applications across heterogeneous systems.

The SAS Common Messaging Interface includes messaging functions that are common to WebSphere MQ, MSMQ, and Rendezvous. However, the TIBCO Rendezvous message delivery system differs from the other transports in some important ways. Developers must take these differences into account when using the Common Messaging Interface to support Rendezvous-based applications. The main differences are as follows:

- Rendezvous uses an approach called subject-based addressing. While both WebSphere MQ and MSMQ deliver messages to specific destination queues using queue names, Rendezvous broadcasts messages that have been labeled with user-defined subject names. Data consumer applications listen for particular subject names and receive messages only when the subject name matches a name being listened for. The communicating programs must agree in advance on the subject names to be used and the forms of messages to be exchanged.

- Because messages are broadcast to subject names instead of specific destination queues, a message can be received only by stations that are online and actively listening for the subject name associated with the message.

“Common Message Queueing CALL Routines” on page 205 explains how to use the SAS Common Messaging Interface to access the unique features of TIBCO Rendezvous. “TIBCO Rendezvous Coding Example” on page 170 shows how to use the SAS Common Messaging Interface with TIBCO Rendezvous. For additional information, please consult the TIBCO documentation.

Rendezvous Certified Message Delivery (Rendezvous-CM)

Certified message delivery features offer a stronger assurance of delivery than reliable message delivery. Certified message delivery protocols also offer the following:
tighter control

greater flexibility

fine-grained reporting

To determine whether you should use Rendezvous certified message delivery, please consult the TIBCO documentation.

"Common Message Queueing CALL Routines" on page 205 explains how to use the SAS Common Messaging Interface to access the features of TIBCO Rendezvous Certified Message Delivery. "TIBCO Rendezvous Certified Messaging Coding Examples" on page 173 shows how to use the SAS Common Messaging Interface with TIBCO Rendezvous Certified Message Delivery. For additional information, please consult the TIBCO documentation.

TIBCO Rendezvous Coding Example

The following example of a SAS DATA step shows how to use the SAS Common Messaging Interface with the TIBCO Rendezvous transport to send and receive messages using subject-based addressing.

```sas
data _null_;

length msg $ 200;
length qid qid2 tid rc attchflg 8;
length parm1 parm2 parm3 recv1 recv2 recv3 8;
length parm4 recv4 $50;
length map $ 80;
length event $ 10;

tid=0;
rc=0;
put '----';
put 'Call INIT';
CALL INIT(tid, 'RENDEZVOUS', rc);
if rc ^= 0 then do;
   put 'INIT: failed';
   msg = sysmsg();
   put msg;
end;
else put 'INIT: succeeded';

rc=0;
qid=0;
put '----';
put 'Call OPENQUEUE for queue1 to listen for and receive messages';
CALL OPENQUEUE(qid, tid, 'test.subject', 'FETCH', rc, "POLL(Timeout=15)");
if rc ^= 0 then do;
   put 'OPENQUEUE: failed';
   msg = sysmsg();
   put msg;
end;
```
else put 'OPENQUEUE: succeeded';

rc=0;
qid2=0;
put '----';
put 'Call OPENQUEUE for queue2 to send messages';
CALL OPENQUEUE(qid2, tid, 'test.subject',
    'DELIVERY', rc);
if rc ^= 0 then do;
    put 'OPENQUEUE: failed';
    msg = sysmsg();
    put msg;
end;
else put 'OPENQUEUE: succeeded';

rc=0;
put '----';
put 'Call SETMAP';
CALL SETMAP('mymap', 'REGISTRY', rc,
    'SHORT;LONG;DOUBLE;CHAR,,50');
if rc ^= 0 then do;
    put 'SETMAP: failed';
    msg = sysmsg();
    put msg;
end;
else put 'SETMAP: succeeded';

parm1=100;
parm2=9999;
parm3=9999.1234;
parm4="ABCDEFGHIJKLMNOPQRSTUVWXYZ";
put '----';
put 'Call SENDMESSAGE';
call sendmessage(qid2,rc,"map","mymap",
    parm1,parm2,parm3,parm4);
if rc ^= 0 then do;
    put 'send message failed: ';
    msg=sysmsg();
    put msg;
end;
else put 'send message succeeded';

rc = 0;
put '----';
put 'Call RECEIVEMESSAGE';

map = "mymap";
call receivemessage(qid, rc, event,
    attchflg,"map",map,recv1,recv2,recv3,recv4);
put 'qid = ' qid;
put 'event = ' event;
put 'attchflg = ' attchflg;
if rc ^= 0 then do;
    put 'receive message failed: ';
    msg=sysmsg();
put msg;
end;
else do;
   put 'receive message succeeded';
   put map;
end;
if event eq 'DELIVERY' then do;
   put 'Message has been delivered';
   put 'recv1 = ' recv1;
   put 'recv2 = ' recv2;
   put 'recv3 = ' recv3;
   put 'recv4 = ' recv4;
end;
rc=0;
put '----';
put 'Call CLOSEQUEUE for queue2';
CALL CLOSEQUEUE(qid2, rc);
if rc ^= 0 then do;
   put 'CLOSEQUEUE: failed';
   msg = sysmsg();
   put msg;
end;
else put 'CLOSEQUEUE: succeeded';
rc=0;
put '----';
put 'Call CLOSEQUEUE for queue1';
CALL CLOSEQUEUE(qid, rc);
if rc ^= 0 then do;
   put 'CLOSEQUEUE: failed';
   msg = sysmsg();
   put msg;
end;
else put 'CLOSEQUEUE: succeeded';
rc=0;
put '----';
put 'Call TERM';
CALL TERM(tid, rc);
if rc ^= 0 then do;
   put 'TERM: failed';
   msg = sysmsg();
   put msg;
end;
else put 'TERM: succeeded';

run;
Example 1: Sending and Receiving Messages in the Same DATA Step

In this example, the sender and listener use the same DATA step.

data _null_;

    length msg $ 200;
    length gid gid2 tid rc 8;
    length map $80;
    length recv1 recv2 recv3 8;
    length recv4 $50;
    length event $10;

    tid=0;
    rc=0;
    put '----';
    put 'Call INIT';
    CALL INIT(tid, 'RENDEZVOUS-CM', rc);
    if rc ^= 0 then do;
        put 'INIT: failed';
        msg = sysmsg();
        put msg;
    end;
    else put 'INIT: succeeded';

    call setmodel("RENDEZVOUS-CM", "RENDCMSENDER",
                  "REGISTRY", rc, "CMNAME, LEDGER",
                  "cmsender", "c:\cmsendledger.txt");
    if rc ^= 0 then do;
        put 'SETMODEL: failed';
        msg = sysmsg();
        put msg;
    end;
    else put 'SETMODEL: succeeded';

    call setmodel("RENDEZVOUS-CM", "RENDCMRECEIVE",
                  "REGISTRY", rc, "CMNAME, LEDGER, REQUESTOLD,
                  SYNCLEDGER", "cmreceive", "c:\cmrcvledger.txt",
                  "YES", "NO");
    if rc ^= 0 then do;
        put 'SETMODEL: failed';
        msg = sysmsg();
        put msg;
end;
else put 'SETMODEL: succeeded';

rc=0;
put '----';
put 'Call SETMAP';
CALL SETMAP('rendmap', 'REGISTRY', rc,
'SHORT;LONG;DOUBLE;CHAR,,50');
if rc ^= 0 then do;
   put 'SETMAP: failed';
   msg = sysmsg();
   put msg;
end;
else put 'SETMAP: succeeded';

rc=0;
qid2=0;
put '----';
put 'Call OPENQUEUE';
CALL OPENQUEUE(qid2, tid, 'testcm.subject',
'DELIVERY', rc, "DYNAMIC(Model=rendcmserver)");
if rc ^= 0 then do;
   put 'OPENQUEUE: failed';
   msg = sysmsg();
   put msg;
end;
else put 'OPENQUEUE: succeeded';
put "qid2= " qid2;

rc=0;
qid=0;
put '----';
put 'Call OPENQUEUE';
CALL OPENQUEUE(qid, tid, 'testcm.subject', 'FETCH', rc,
"DYNAMIC(Model=rendcmreceive)", "POLL(Timeout=15)" );
if rc ^= 0 then do;
   put 'OPENQUEUE: failed';
   msg = sysmsg();
   put msg;
end;
else put 'OPENQUEUE: succeeded';
put "qid= " qid;

/* send a message */
parm1=100;
parm2=9999;
parm3=9999.1234;
parm4="Demonstrating the rendezvous message api.";

put '----';
put 'Call SENDMESSAGE';
call sendmessage(qid2,rc,"map","rendmap", parm1,parm2,parm3,parm4);
if rc ^= 0 then do;
   put 'send message failed: ';

msg=sysmsg();
put msg;
end;
else put 'send message succeeded';

rc = 0;
put '-----';
put 'Call RECEIVEMESSAGE';

map = *rendmap*;
call receivemessage(qid, rc, event,
                  attchflg,"map",map,recv1,recv2,recv3,recv4);
put 'qid = ' qid;
put 'event = ' event;
put 'attchflg = ' attchflg;
if rc ^= 0 then do;
   put 'receive message failed: ';
   msg=sysmsg();
   put msg;
end;
else do;
   put 'receive message succeeded';
   put map;
end;
if event eq 'DELIVERY' then do;
   put 'Message has been delivered';
   if attchflg eq 1 then do;
      put 'Attachments are associated with this message';
      /* process attachments...*/
   end;
   put 'recv1 = ' recv1;
   put 'recv2 = ' recv2;
   put 'recv3 = ' recv3;
   put 'recv4 = ' recv4;
end;

rc=0;
put '-----';
put 'Call CLOSEQUEUE for sender';
put "qid2= " qid2;
CALL CLOSEQUEUE(qid2, rc, "DELETE_PURGE");
if rc ^= 0 then do;
   put 'CLOSEQUEUE: failed';
   msg = sysmsg();
   put msg;
end;
else put 'CLOSEQUEUE: succeeded';

rc=0;
put '-----';
put 'Call CLOSEQUEUE for receiver';
put "qid= " qid;
CALL CLOSEQUEUE(qid, rc, "DELETE_PURGE");
if rc ^= 0 then do;
   put 'CLOSEQUEUE: failed';
   msg = sysmsg();
   put msg;
end;
else put 'CLOSEQUEUE: succeeded';

rc=0;
picture '-----';
picture 'Call TERM';
CALL TERM(tid, rc);
if rc ^= 0 then do;
   put 'TERM: failed';
   msg = sysmsg();
   put msg;
end;
else put 'TERM: succeeded';

run;

Example 2: Sending and Receiving Messages in Separate DATA Steps

Overview

In this example, the sender and listener use separate DATA steps. Each DATA step is run in a separate SAS session. The receiving DATA step needs to start running before the sending DATA step ends.

Sending DATA Step

/* SAS DATA step to send a certified message */

data _null_
length msg $ 200;
length qid2 tid rc 8;
length map $80;
length recv4 $50;
length event $10;
length queue $ 80;

tid=0;
rc=0;
call setmodel("RENDZEVOUS-CM", "RENDCMSENDER",
   "REGISTRY", rc, "CMNAME, LEDGER", "cmsender",
   "c:\sendledger.txt");
if rc ^= 0 then do;
   put 'SETMODEL: failed';

msg = sysmsg();
put msg;
end;
else put 'SETMODEL: succeeded';
rc=0;
put '----';
put 'Call SETMAP';
CALL SETMAP('rendmap', 'REGISTRY', rc,
  'SHORT;LONG;DOUBLE;CHAR,,50');
if rc ^= 0 then do;
  put 'SETMAP: failed';
  msg = sysmsg();
  put msg;
end;
else put 'SETMAP: succeeded';
call setalias("queue", "tibcmalias", "REGISTRY",
  rc, "RENDZVOUS-CM", "send.cmmsg");
if rc ^= 0 then do;
  put 'set_alias failed: ';
  msg=sysmsg();
  put msg;
end;
else put 'set_alias succeeded';
put ' this should be next';
rc=0;
qname = "tibcmalias";
qid2=0;
put '----';
put 'Call OPENQUEUE for queue2';
CALL OPENQUEUE(qid2, tid, qname, 'DELIVERY',
  rc, "DYNAMIC(Model=rendcmsender)");
if rc ^= 0 then do;
  put 'OPENQUEUE: failed';
  msg = sysmsg();
  put msg;
end;
else put 'OPENQUEUE: succeeded';
/* send a message */
parm1=100;
parm2=9999;
parm3=9999.1234;
parm4="Demonstrating the rendezvous message api.";
put '----';
put 'Call SENDMESSAGE';
call sendmessage(qid2,rc,"map, addlistener","rendmap",
  "cmreceive",parm1,parm2,parm3,parm4);
if rc ^= 0 then do;
  put 'send message failed: ';
  msg=sysmsg();
Receiving DATA Step

/* SAS DATA step to receive certified messages */

data _null_
length msg $ 200;
length gid tid rc 8;
length map $80;
length event $10;
length queue $ 80;
length token $300;
length attach $10;
length recv1 recv2 recv3 8;
length recv4 $50;
length certified $8;
length sendername $50;
rc=0;
call setmodel("RENDZVOUS-CM", "RENDCMreceive",
"REGISTRY", rc, "CMNAME, LEDGER, REQUESTOLD",
"cmreceive", "c:\recvledger.txt", "YES");

if rc ^= 0 then do;
  put 'SETMODEL: failed';
  msg = sysmsg();
  put msg;
end;
else put 'SETMODEL: succeeded';

call setalias("queue", "tibcmalias", "REGISTRY",
rc, "RENDEVOUS-CM", "send.cmmsg");
if rc ^= 0 then do;
    put 'set_alias failed: ';
    msg=sysmsg();
    put msg;
end;
else put 'set_alias succeeded';
rc=0;
qid=0;
tid = 0;
qname = "tibcmalias";
put '----';
put 'Call OPENQUEUE';
CALL OPENQUEUE(qid, tid, qname, 'FETCH', rc,
    "DYNAMIC(Model=rendcmreceive)", "POLL(TIMEOUT=30)"");
if rc ^= 0 then do;
    put 'OPENQUEUE: failed';
    msg = sysmsg();
    put msg;
end;
else put 'OPENQUEUE: succeeded';
put "qid= " qid;

put "CALL receivemessage";
map = "rendmap";
call receivemessage(qid, rc, event,
    attchflg,"map,certified,sendername",map, certified,
    sendername, recv1,recv2,recv3,recv4);
put 'qid = ' qid;
put 'event = ' event;
put 'attchflg = ' attchflg;
put 'certified = ' certified;
put 'sendername = ' sendername;
if rc ^= 0 then do;
    put 'receive message failed: ';
    msg=sysmsg();
    put msg;
end;
else do;
    put 'receive message succeeded';
    put map;
end;
if event eq 'DELIVERY' then
do;
    put 'Message has been delivered';
    if attchflg eq 1 then do;
        put 'Attachments are associated with this message';
        /* process attachments...*/
    end;
    put 'recv1 = ' recv1;
    put 'recv2 = ' recv2;
    put 'recv3 = ' recv3;
    put 'recv4 = ' recv4;
Using a Repository with Application Messaging

The common messaging interface enables you to store information about message queues in the local SAS registry. The information that can be stored and retrieved include the following:

Transport alias
is an alias name that describes a transport (MQSeries [refers to WebSphere MQ], MSMQ, Rendezvous, or Rendezvous-CM)

Queue alias
is an alias name that describes a transport and queue

Dynamic queue model
is a model name that describes a queue's properties

Transport model
is a model name that describes a Rendezvous or Rendezvous-CM transport

Data map description
is a map name that describes the format of data within a message

Placing this type of information in storage provides both reusability and encapsulation. A repository can contain all queue definitions, thereby enabling you to focus on the application usage rather than the specific definition of a queue.

The SAS registry provides methods for defining your own queues or overriding globally defined queues. It provides you with complete control and flexibility over a queue.

To bypass the SAS registry altogether, specify the following macro variable:

```sas
%let REGISTRY_BYPASS=1.
```

For more information about using a repository with application messaging, see “Using the SAS Registry with the Common Messaging Interface” on page 181.
Using the SAS Registry with the Common Messaging Interface

Overview of Using the SAS Registry

The SAS registry can be used to store information about objects used for application messaging. This document provides information about using the SAS registry editor to view registry entries. It also provides a sample program for managing registry objects under program control.

Using the SAS Registry Editor

The SAS Registry Editor can be used to verify that values set programmatically for application messaging objects were set properly. To invoke the Registry Editor, select Solutions ➔ Accessories ➔ Registry Editor in the Base SAS menu.

The SAS registry has the following hierarchy for application messaging objects:

Products
  Base
    SAS Messaging
      Maps
      Models
      Queues
      Transports

Writing Applications to Access the SAS Registry

A typical program would configure information such as the following:

- Map data descriptor
- Queue and transport aliases
- Dynamic model for transport processing.

The following code illustrates how to set and retrieve information within the SAS registry.

```sas
data _null_;
length rc 8 msg $ 200;
length descriptor transport queue label $ 80;
length type $ 32;
length auth journal priv trans $ 10;
length basep journalq quota 8;
```
put 'Registry Map creation...';
call setmap('mymap', 'registry', rc,
   'char,0,80;double;');
if rc ne 0 then do;
   put 'Setmap failed';
   msg = sysmsg();
   put msg;
end;
else put 'Setmap was successful';

put 'Registry Map retrieval...';
call getmap('mymap', 'registry', rc, descriptor);
if rc ne 0 then do;
   put 'Getmap failed';
   msg = sysmsg();
   put msg;
end;
else do;
   put 'Getmap was successful';
   put 'descriptor = ' descriptor;
end;

put 'Registry Map deletion...';
call deleteMap('mymap', 'registry', rc);
if rc ne 0 then do;
   put 'Deletemap failed';
   msg = sysmsg();
   put msg;
end;
else put 'Deletemap was successful';

put '-----------------------------';
put 'Registry Queue creation...';
call setalias('queue', 'myqueue', 'registry',
   rc, 'msmq', 'machine_name\queue_name');
if rc ne 0 then do;
   put 'Setalias failed';
   msg = sysmsg();
   put msg;
end;
else put 'Setalias succeeded';

put 'Registry Queue retrieval...';
call getalias('queue', 'myqueue', 'registry',
   rc, transport, queue);
if rc ne 0 then do;
   put 'Getalias failed';
   msg = sysmsg();
   put msg;
end;
else do;
   put 'Getalias succeeded';
   put 'transport = ' transport;
   put 'queue = ' queue;
end;

put '-------------------------------';

put 'Registry Transport creation...';
call setalias('transport', 'mytransport', 'registry', rc, 'MSMQ');
if rc ne 0 then do;
   put 'Setalias failed';
   msg = sysmsg();
   put msg;
end;
else put 'Setalias succeeded';

put 'Registry Transport retrieval...';
call getalias('transport', 'mytransport', 'registry', rc, transport);
if rc ne 0 then do;
   put 'Getalias failed';
   msg = sysmsg();
   put msg;
end;
else do;
   put 'Getalias succeeded';
   put 'transport = ' transport;
   put 'queue = ' queue;
end;

put '-------------------------------';

put 'Registry Model creation...';
call setmodel('msmq', 'mymodel', 'registry', rc, 'authenticate, label', 'always', 'Test Queue of MyModel');
if rc ne 0 then do;
   put 'Setmodel failed';
   msg = sysmsg();
   put msg;
end;
else put 'Setmodel succeeded';

put 'Registry Model retrieval...';
call getmodel('msmq', 'mymodel', 'registry', rc, 'authenticate,basepriority,journal,journalquota,label,privlevel,quota,transaction,type', auth, basep, journal, journalq, label, priv, quota, trans, type);
if rc ne 0 then do;
   put 'Getmodel failed';
   msg = sysmsg();
   put msg;
end;
else do;
   put 'Getmodel succeeded';
Attachment Layout for WebSphere MQ and MSMQ

Attachments consist of multiple physical messages. The beginning of an attachment is recognized by having a message type of 100000. To identify this message, it will be referred to as the attachment header.

Layout of an attachment header message:

Note: All character strings are null terminated.

```
byte[24] - header correlid
(correlationid of this header
message)
long     - original msg type (msg type
provided by the
sending application)
byte[24] - original msg correlid (msg correlationid
provided by the sending application)
byte[24] - message correlid (generated correlationid for
the msg)
int      - number of attachments

int      - attachment type
  1 - SAS data set
  2 - External text file
  3 - External binary file
byte[24] - attachment correlid (correlationid associated
with this attachment)
int      - length of qualifier 1
char[]   - qualifier 1
  external files: designates the sending file
  specification "FILENAME" or "FILEREF"
  dataset: designates the sending library name
int      - length of qualifier 2
char[]   - qualifier 2
```
Other physical messages are also needed to make up a complete attachment. These messages will be called subordinated messages, and they all have a message type of 100001.

The subordinate message that usually follows after the attachment header message is the application message. It can be filtered by using the message correlation ID located in the attachment header message. It contains the actual application-generated message.

The attachment (external file or SAS data set) subordinate messages follow next. They contain the necessary information to re-create the file or data set.

To locate the subordinate message that contains the number of physical messages that are associated with this attachment, filter it by using the attachment correlation ID that is located in the attachment header message. The content of this message is a single numeric integer that corresponds to the number of messages that are associated with this attachment, excluding this message. To filter the rest of the messages that are associated with this attachment, use the same attachment correlation ID that is located in the attachment header message (16 bytes) with a sequence number (4 bytes) added to the end of it. For example, if the attachment correlation ID was 000102030405060708090A0B0C0D0E0F, you would filter this message to find out how many more messages are associated with this attachment. For example, if three more messages make up this attachment, then you can locate these messages by filtering a correlation ID of 000102030405060708090A0B0C0D0E0F00000001, 000102030405060708090A0B0C0D0E0F00000002, and 000102030405060708090A0B0C0D0E0F00000003, respectively. The sequenced attachment correlation ID messages are actually sent to the queue before the non-sequenced attachment correlation ID message. Therefore, if you are able to receive the non-sequenced attachment correlation ID message (that is, a message that tells you how many messages make up this attachment), then you can make sure that the complete attachment has been queued.

At this point, attachment processing differs depending on the attachment type.

For external files, the first sequenced attachment correlation ID message (attachment_correlid+00000001) contains two numeric integers that correspond to the file's logical record length and size, respectively. The rest of the attachment correlation ID messages make up the file itself. The contents of these messages are as follows:

```
long    - size of logical record
char[]  - actual record

```
These messages are limited to 32K. If a file is too large to fit, then it spans multiple physical messages.

Here is an example of an external file attachment residing on a queue:

```
msg type
msg correlid msg contents
-------- ------------
100000
1111111111111111111111111111111100000000
00000001
0000000000000000000000000000000000000000
22222222222222222222222200000000
00000001
00000003
3333333333333333333333333333333300000000
00000008
"FILENAME"
0000000D
"d:\mytext.txt"
0000000C
"Text file..."
00000000
00000000
100001
22222222222222222222222200000000
00000003
"This is the actual application message."
100001
3333333333333333333333333333333300000001
lrecl|filesize
100001
3333333333333333333333333333333300000002
len|record|len|record|len|record...
100001
3333333333333333333333333333333300000003
len|record|len|record|len|record...
100001
3333333333333333333333333333333300000000
00000003
```

For data sets, the sequenced attachment correlation ID messages begin with a type identifier. This identifier signifies the type of information that is in this message. A type identifier of one signifies data set definitions. A type identifier of two signifies variable definitions. A type identifier of three signifies actual observations. Type identifiers four (indexes) and five (integrity constraints) usually have no use and can be ignored.

Note: All character strings are null terminated.

Layout of a data set definition message:

```
int   - type (data set definition=1)
int   - version (future)
long  - data set type length
char[] - data set type
long  - data set label length
char[] - data set label
long - number of observations
long - number of variables
long - observation length
long - length of compress
char[] - compress
char - reuse
long - length of encrypt
char[] - encrypt
long - number of variables in sort key
long - length of sort collating sequence
char[] - sort collating sequence
short - sort flags
int - read password flag
byte[4] - read password (encrypted)
int - write password flag
byte[4] - write password (encrypted)
int - alter password flag

Layout of a variable definition message:

int - type (variable definition=2)
-------------------------------
long - length of variable name
char[] - variable name
long - length of format name
char[] - format name
long - length of informat name
char[] - informat name
long - variable label length
char[] - variable label
char - variable type (1=double, otherwise character)
long - variable length
long - format field length
long - format decimal
long - informat field length
long - informat decimal
char - nsort
-------------------------------
.
.
. repeat for each variable

Note: Variable definitions might span multiple physical
messages if definitions are larger than 32K.

Layout of an observation message:

int - type (observation=3)
data - the layout of data is defined by the variable
definition above
Note: Observations might span multiple physical messages if they are larger than 32K.

Layout of an index message:

- int - type (index=4)
- long - upercmx
- long - length of index/key name
- char[] - index/key name
- long - flags
- long - number of variables in the index/key
- long - variable lengths added together
- char[] - all variables null terminated

- repeat for each index

---

Attachment Layout for TIBCO Rendezvous

Overview of Attachment Layout for TIBCO Rendezvous

An attachment consists of multiple physical messages. Each physical message has a specific message type. The field name of the first field in each message specifies the message type. Subsequent fields in the same message should use the same field name.

Data Message Layout

The following table shows the field name and purpose of the "MSG," or "DATA," type.

Note: The message type "MSG," or "DATA," can be retrieved without a field ID. All other message types must use a field ID.

<table>
<thead>
<tr>
<th>Field Name</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;MSG&quot; or &quot;DATA&quot;</td>
<td>message data sent using a map</td>
</tr>
</tbody>
</table>
Data Set Attachment Layout

All attachments are required to have an attachment header and a "LST" message. However, not all messages are required. For example, many data sets do not use integrity constraints or indexes. If a data set does not contain the information that is contained in a message type, then the message is not required to be sent. The following table shows the field name, the purpose of each message type, and the order in which messages should be sent for a data set.

Table 7.2  Fields for the Data Set Attachment Layout

<table>
<thead>
<tr>
<th>Field Name</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;HDR&quot;</td>
<td>attachment header</td>
</tr>
<tr>
<td>&quot;MSG&quot; or &quot;DATA&quot;</td>
<td>message data sent using a map</td>
</tr>
<tr>
<td>&quot;DAT&quot;</td>
<td>data set descriptor</td>
</tr>
<tr>
<td>&quot;VAR&quot;</td>
<td>variable definition for data set</td>
</tr>
<tr>
<td>&quot;ATO&quot;</td>
<td>data set observations</td>
</tr>
<tr>
<td>&quot;ATI&quot;</td>
<td>data set index</td>
</tr>
<tr>
<td>&quot;ATC&quot;</td>
<td>data set integrity constraints</td>
</tr>
<tr>
<td>&quot;LST&quot;</td>
<td>last message of attachment</td>
</tr>
</tbody>
</table>

External File Attachment Layout

All attachments are required to have an attachment header and a "LST" message. However, not all messages are required. For each "FDC" record, send either a text file or a binary file. You can send more than one file in an attachment. Each file must have an "FDC" message and then one of the following:

- one or more "ATX" messages for the text files
- one or more "ATB" messages for the binary files

The following table shows the field name, the purpose of each message type, and the order in which messages should be sent for an external file.
Table 7.3 Fields for the External File Attachment Layout

<table>
<thead>
<tr>
<th>Field Name</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;HDR&quot;</td>
<td>attachment header</td>
</tr>
<tr>
<td>&quot;FDC&quot;</td>
<td>external file descriptor</td>
</tr>
<tr>
<td>&quot;ATX&quot;</td>
<td>text file attachment body</td>
</tr>
<tr>
<td>&quot;ATB&quot;</td>
<td>binary file attachment body</td>
</tr>
<tr>
<td>&quot;LST&quot;</td>
<td>last message of attachment</td>
</tr>
</tbody>
</table>

The following sections contain the description and required format for each message type.

**Message Data - "MSG" or "DATA"**

*Note:* The message type "MSG," or "DATA," can be retrieved without a field ID. All other message types must use a field ID.

If any message data is to be sent along with an attachment, that message is sent following the attachment header. The field name for this type of message is either "MSG" or "DATA." The following sample is based on the map that is used in the code example provided on the Common Messaging Interface documentation.

The map for this message is described as: 'SHORT;LONG;DOUBLE;CHAR,,50'.

The following table shows the data values for the message data.

Table 7.4 Data Values for the Message Data

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>parm1</td>
<td>100;</td>
</tr>
<tr>
<td>parm2</td>
<td>9999;</td>
</tr>
<tr>
<td>parm3</td>
<td>9999.1234;</td>
</tr>
<tr>
<td>parm4</td>
<td>&quot;ABCDEFGHIJKLMNOPQRSTUVWXYZ&quot;; (blank padded to 50)</td>
</tr>
</tbody>
</table>

The following table shows the data type values for the message data.
Table 7.5  Data Types for the Message Data

<table>
<thead>
<tr>
<th>Data Type</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>short</td>
<td>1</td>
<td>add with tibrvMsg_AddI16()</td>
</tr>
<tr>
<td>long</td>
<td>2</td>
<td>add with tibrvMsg_AddI32() as appropriate</td>
</tr>
<tr>
<td>double</td>
<td>3</td>
<td>add with tibrvMsg_AddF64()</td>
</tr>
<tr>
<td>string(char)</td>
<td>4</td>
<td>add with tibrvMsg_AddString()</td>
</tr>
</tbody>
</table>

The following table shows the layout of the message data.

Table 7.6  Fields for the Message Data

<table>
<thead>
<tr>
<th>Field ID</th>
<th>Field Type</th>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>int</td>
<td>tibrvMsg_AddI32()</td>
<td>The number of data pieces to follow. For this example, the value of the field is &quot;4&quot;.</td>
</tr>
<tr>
<td>2</td>
<td>int</td>
<td>tibrvMsg_AddI32()</td>
<td>The data type of the first data item. Because this data item is a short, the value for this field is &quot;1&quot;.</td>
</tr>
<tr>
<td>3</td>
<td>short</td>
<td>tibrvMsg_AddI16()</td>
<td>The actual value of the first parameter being sent. In this case, because it is a short, the value is added to the message by using tibrvMsg_AddI16(). The value for this field is &quot;100&quot;.</td>
</tr>
</tbody>
</table>

For each parameter that is sent, repeat fields 2 and 3 in the previous table, setting the appropriate values and incrementing the field IDs.

Attachment Header - "HDR"

The beginning of an attachment is recognized by processing the attachment header message. This message type is recognized by the "HDR" field name in all fields.

The following table shows the layout of the attachment header.

Note:  All character strings are null terminated.
<table>
<thead>
<tr>
<th>Field ID</th>
<th>Field Type</th>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>byte[24]</td>
<td>tibrvMsg_AddString()</td>
<td>header correlation ID: can be set to all blanks</td>
</tr>
<tr>
<td>2</td>
<td>unsigned long</td>
<td>tibrvMsg_AddU32()</td>
<td>reserved: set to 0</td>
</tr>
<tr>
<td>3</td>
<td>byte[24]</td>
<td>tibrvMsg_AddString()</td>
<td>reserved: set to all blanks</td>
</tr>
<tr>
<td>4</td>
<td>byte[24]</td>
<td>tibrvMsg_AddString()</td>
<td>message correlation ID: can be set to all blanks</td>
</tr>
<tr>
<td>5</td>
<td>integer</td>
<td>tibrvMsg_AddI32()</td>
<td>number of attachments in message (1 per data set)</td>
</tr>
<tr>
<td>6</td>
<td>integer</td>
<td>tibrvMsg_AddI32()</td>
<td>attachment type: value is</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>■ &quot;1&quot; for SAS data set.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>■ &quot;2&quot; for an external text file</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>■ &quot;3&quot; for an external binary file</td>
</tr>
<tr>
<td>7</td>
<td>byte[24]</td>
<td>tibrvMsg_AddString()</td>
<td>attachment correlation ID: can be set to all blanks</td>
</tr>
<tr>
<td>8</td>
<td>int</td>
<td>tibrvMsg_AddI32()</td>
<td>length of qualifier 1 in field 9</td>
</tr>
<tr>
<td>9</td>
<td>char[]</td>
<td>tibrvMsg_AddString()</td>
<td>qualifier 1:</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>■ external files: designates the sending file specification &quot;FILENAME&quot; or &quot;FILEREF&quot;</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>■ data set: designates the sending library name</td>
</tr>
<tr>
<td>10</td>
<td>int</td>
<td>tibrvMsg_AddI32()</td>
<td>length of qualifier 2 in field 11</td>
</tr>
<tr>
<td>11</td>
<td>char[]</td>
<td>tibrvMsg_AddString()</td>
<td>qualifier 2:</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>■ external files: designates the sending filename or fileref</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>■ data set: designates the sending member name</td>
</tr>
<tr>
<td>12</td>
<td>int</td>
<td>tibrvMsg_AddI32()</td>
<td>length of attachment description</td>
</tr>
<tr>
<td>13</td>
<td>char[]</td>
<td>tibrvMsg_AddString()</td>
<td>attachment description</td>
</tr>
<tr>
<td>14</td>
<td>int</td>
<td>tibrvMsg_AddI32()</td>
<td>user-specified minor version number</td>
</tr>
</tbody>
</table>
For each attachment in the list, repeat fields 6-15 in the previous table, incrementing the field ID each time.

The attachment header is usually followed by the subordinate messages that contain the information necessary to re-create the data set or the external file.

## Data Set Definition - "DAT"

The data set definition message is sent following the message data. This message type is recognized by the "DAT" field name in all fields.

The following table shows the layout of the data set definition.

**Note:** All character strings are null terminated.

<table>
<thead>
<tr>
<th>Field ID</th>
<th>Field Type</th>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>int</td>
<td>tibrvMsg_AddI32()</td>
<td>type of record is data set definition= 1</td>
</tr>
<tr>
<td>2</td>
<td>int</td>
<td>tibrvMsg_AddI32()</td>
<td>version information or 0</td>
</tr>
<tr>
<td>3</td>
<td>long</td>
<td>tibrvMsg_AddI32()</td>
<td>data set type length</td>
</tr>
<tr>
<td>4</td>
<td>char[]</td>
<td>tibrvMsg_AddString()</td>
<td>data set type</td>
</tr>
<tr>
<td>5</td>
<td>long</td>
<td>tibrvMsg_AddI32()</td>
<td>data set label length</td>
</tr>
<tr>
<td>6</td>
<td>char[]</td>
<td>tibrvMsg_AddString()</td>
<td>data set label</td>
</tr>
<tr>
<td>7</td>
<td>long</td>
<td>tibrvMsg_AddI32()</td>
<td>number of observations</td>
</tr>
<tr>
<td>8</td>
<td>long</td>
<td>tibrvMsg_AddI32()</td>
<td>number of variables</td>
</tr>
<tr>
<td>9</td>
<td>long</td>
<td>tibrvMsg_AddI32()</td>
<td>observation length</td>
</tr>
<tr>
<td>10</td>
<td>long</td>
<td>tibrvMsg_AddI32()</td>
<td>length of compress</td>
</tr>
<tr>
<td>11</td>
<td>char[]</td>
<td>tibrvMsg_AddString()</td>
<td>compress</td>
</tr>
<tr>
<td>12</td>
<td>char</td>
<td>tibrvMsg_AddString()</td>
<td>reuse (&quot;R&quot; or &quot;E&quot;)</td>
</tr>
</tbody>
</table>
### Variable Definition - "VAR"

The variable definition message is sent following the data set definition message. This message type is recognized by the "VAR" field name in all fields.

The following table shows the layout of the variable definition.

#### Table 7.9 Fields for the Variable Definition

<table>
<thead>
<tr>
<th>Field ID</th>
<th>Field Type</th>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>int</td>
<td>tibrvMsg_AddI32()</td>
<td>number of variables</td>
</tr>
<tr>
<td>2</td>
<td>int</td>
<td>tibrvMsg_AddI32()</td>
<td>type of record is variable definition=2</td>
</tr>
</tbody>
</table>

Note: All character strings are null terminated.
<table>
<thead>
<tr>
<th>Field ID</th>
<th>Field Type</th>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>long</td>
<td>tibrvMsg_AddI32()</td>
<td>length of variable name</td>
</tr>
<tr>
<td>4</td>
<td>char[]</td>
<td>tibrvMsg_AddString()</td>
<td>name of variable</td>
</tr>
<tr>
<td>5</td>
<td>long</td>
<td>tibrvMsg_AddI32()</td>
<td>length of format name</td>
</tr>
<tr>
<td>6</td>
<td>char[]</td>
<td>tibrvMsg_AddString()</td>
<td>format name</td>
</tr>
<tr>
<td>7</td>
<td>long</td>
<td>tibrvMsg_AddI32()</td>
<td>length of informat name</td>
</tr>
<tr>
<td>8</td>
<td>char[]</td>
<td>tibrvMsg_AddString()</td>
<td>informat name</td>
</tr>
<tr>
<td>9</td>
<td>long</td>
<td>tibrvMsg_AddI32()</td>
<td>length of variable label</td>
</tr>
<tr>
<td>10</td>
<td>char[]</td>
<td>tibrvMsg_AddString()</td>
<td>variable label</td>
</tr>
<tr>
<td>11</td>
<td>char</td>
<td>tibrvMsg_AddString()</td>
<td>type of variable (1=numeric, 2=char)</td>
</tr>
<tr>
<td>12</td>
<td>long</td>
<td>tibrvMsg_AddI32()</td>
<td>length of variable</td>
</tr>
<tr>
<td>13</td>
<td>long</td>
<td>tibrvMsg_AddI32()</td>
<td>format field length</td>
</tr>
<tr>
<td>14</td>
<td>long</td>
<td>tibrvMsg_AddI32()</td>
<td>format decimal</td>
</tr>
<tr>
<td>15</td>
<td>long</td>
<td>tibrvMsg_AddI32()</td>
<td>informat field length</td>
</tr>
<tr>
<td>16</td>
<td>long</td>
<td>tibrvMsg_AddI32()</td>
<td>informat decimal</td>
</tr>
<tr>
<td>17</td>
<td>char</td>
<td>tibrvMsg_AddString()</td>
<td>nsort information</td>
</tr>
</tbody>
</table>

For each variable, repeat the fields in the previous table.

**Note:** If definitions are larger than 32K, then variable messages might span multiple physical messages.

---

**Data Set Observations - "ATO"**

The data set observations message is sent following the variable definition message. This message type is recognized by the "ATO" field name in all fields.

The following table shows the layout of the data set observations.

**Note:** All character strings are null terminated.
Table 7.10  Fields for Data Set Observations

<table>
<thead>
<tr>
<th>Field ID</th>
<th>Field Type</th>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>int</td>
<td>tibrvMsg_AddI32()</td>
<td>number of observations</td>
</tr>
<tr>
<td>2</td>
<td>int</td>
<td>tibrvMsg_AddI32()</td>
<td>type of record is observation = 3</td>
</tr>
<tr>
<td>3</td>
<td>int</td>
<td>tibrvMsg_AddI32()</td>
<td>observation type (vtype)</td>
</tr>
<tr>
<td>4</td>
<td>double-observation</td>
<td>tibrvMsg_AddF64()</td>
<td>if observation type in field 3 is numeric</td>
</tr>
<tr>
<td>4</td>
<td>char[] - observation</td>
<td>tibrvMsg_AddString()</td>
<td>if observation type in field 3 is character</td>
</tr>
</tbody>
</table>

For each observation, repeat the fields in the previous table.

Note: If observations are larger than 32K, then they might span multiple physical messages.

Data Set Index - "ATI"

If the data set index message is needed, the data set index message is sent following the data set observations message. This message type is recognized by the "ATI" field name in all fields.

The following table shows the layout of the index definition.

Note: All character strings are null terminated.

Table 7.11  Fields for the Data Set Index

<table>
<thead>
<tr>
<th>Field ID</th>
<th>Field Type</th>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>int</td>
<td>tibrvMsg_AddI32()</td>
<td>type of record is index = 4</td>
</tr>
<tr>
<td>2</td>
<td>int</td>
<td>tibrvMsg_AddI32()</td>
<td>number of records in this message</td>
</tr>
<tr>
<td>3</td>
<td>long</td>
<td>tibrvMsg_AddI32()</td>
<td>upercmx</td>
</tr>
<tr>
<td>4</td>
<td>long</td>
<td>tibrvMsg_AddI32()</td>
<td>length of index or key name</td>
</tr>
<tr>
<td>5</td>
<td>char[]</td>
<td>tibrvMsg_AddString()</td>
<td>index or key name</td>
</tr>
</tbody>
</table>
Data Set Integrity Constraints - "ATC"

If the data set integrity constraints message is needed, then the data set integrity constraints message is sent following the data set index message. This message type is recognized by the "ATC" field name in all fields.

The following table shows the layout of the integrity constraints definition.

---

**Note**: All character strings are null terminated.

Table 7.12  Fields for the Data Set Integrity Constraints

<table>
<thead>
<tr>
<th>Field ID</th>
<th>Field Type</th>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>int</td>
<td>tibrvMsg_AddI32()</td>
<td>type of record is integrity constraint = 5</td>
</tr>
<tr>
<td>2</td>
<td>int</td>
<td>tibrvMsg_AddI32()</td>
<td>number of records in this message</td>
</tr>
<tr>
<td>3</td>
<td>long</td>
<td>tibrvMsg_AddI32()</td>
<td>IC type</td>
</tr>
</tbody>
</table>

Based on the value of field 3 in the previous table, use the following tables.

- If the field type is CHECK for field 3, then use the fields in the following table. For each buffer, repeat field 11 in the previous table, incrementing the field ID each time.

Table 7.13  Fields for the CHECK Field Type

<table>
<thead>
<tr>
<th>Field ID</th>
<th>Field Type</th>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>long</td>
<td>tibrvMsg_AddI32()</td>
<td>max length for this IC</td>
</tr>
<tr>
<td>5</td>
<td>char[]</td>
<td>tibrvMsg_AddString()</td>
<td>name of IC</td>
</tr>
</tbody>
</table>
If the field type is not CHECK for field 3, then use the fields in the following table.

<table>
<thead>
<tr>
<th>Field ID</th>
<th>Field Type</th>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>long</td>
<td>tibrvMsg_AddI32()</td>
<td>retval</td>
</tr>
<tr>
<td>7</td>
<td>long</td>
<td>tibrvMsg_AddI32()</td>
<td>total length</td>
</tr>
<tr>
<td>8</td>
<td>char[]</td>
<td>tibrvMsg_AddString()</td>
<td>list of wtnames</td>
</tr>
<tr>
<td>9</td>
<td>long</td>
<td>tibrvMsg_AddI32()</td>
<td>whlen</td>
</tr>
<tr>
<td>10</td>
<td>long</td>
<td>tibrvMsg_AddI32()</td>
<td>number of members in tree</td>
</tr>
<tr>
<td>11</td>
<td>byte[]</td>
<td>tibrvMsg_AddOpaque()</td>
<td>whbuf buffer</td>
</tr>
</tbody>
</table>

For each NNAME value, repeat field 8 in the previous table, incrementing the field ID each time. Subsequent field IDs will increase from here.

If the field type is not CHECK or FOREIGN KEY for field 3, then use the following table for field 9.

<table>
<thead>
<tr>
<th>Field ID</th>
<th>Field Type</th>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>9</td>
<td>long</td>
<td>tibrvMsg_AddI32()</td>
<td>filler value = 1</td>
</tr>
</tbody>
</table>

If the field type is not CHECK but it is FOREIGN KEY for field 3, then use the fields in the following table.
**Table 7.16  Fields for the FOREIGN KEY Field Type**

<table>
<thead>
<tr>
<th>Field ID</th>
<th>Field Type</th>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>9</td>
<td>long</td>
<td>tibrvMsg_AddI32()</td>
<td>total length of following fields</td>
</tr>
<tr>
<td>10</td>
<td>long</td>
<td>tibrvMsg_AddI32()</td>
<td>fkdelt</td>
</tr>
<tr>
<td>11</td>
<td>long</td>
<td>tibrvMsg_AddI32()</td>
<td>fkupd</td>
</tr>
<tr>
<td>12</td>
<td>long</td>
<td>tibrvMsg_AddI32()</td>
<td>pkling + 1</td>
</tr>
<tr>
<td>13</td>
<td>char[]</td>
<td>tibrvMsg_AddString()</td>
<td>pkname</td>
</tr>
<tr>
<td>14</td>
<td>char[8]</td>
<td>tibrvMsg_AddString()</td>
<td>pkfname libref</td>
</tr>
<tr>
<td>15</td>
<td>long</td>
<td>tibrvMsg_AddI32()</td>
<td>length of member name</td>
</tr>
<tr>
<td>16</td>
<td>char[]</td>
<td>tibrvMsg_AddString()</td>
<td>member name.</td>
</tr>
<tr>
<td>17</td>
<td>long</td>
<td>tibrvMsg_AddI32()</td>
<td>ICP attributes</td>
</tr>
</tbody>
</table>

For each record in the message, repeat field 3 and all subsequent fields in the previous tables.

**External File Descriptor - "FDC"**

This message type is recognized by the "FDC" field name in all fields. For each "FDC" record, send either a text file or a binary file. You can send more than one file in an attachment but the files must be either all text files or all binary files. Each file must have an "FDC" message and then one of the following:

- one or more "ATX" messages for the text files
- one or more "ATB" messages for the binary files

The following table shows the layout of the external file descriptor.

**Table 7.17  Fields for the External File Descriptor**

<table>
<thead>
<tr>
<th>Field ID</th>
<th>Field Type</th>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>int</td>
<td>tibrvMsg_AddI32()</td>
<td>size of logical record</td>
</tr>
<tr>
<td>2</td>
<td>int</td>
<td>tibrvMsg_AddI32()</td>
<td>file size</td>
</tr>
</tbody>
</table>
Text File Attachment - "ATX"

This message type is recognized by the "ATX" field name in all fields. The following table shows the layout of the text file attachment body.

<table>
<thead>
<tr>
<th>Field ID</th>
<th>Field Type</th>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>int</td>
<td>tibrvMsg_AddI32()</td>
<td>number of records in this message</td>
</tr>
<tr>
<td>2</td>
<td>long</td>
<td>tibrvMsg_AddI32()</td>
<td>length of data in field 3</td>
</tr>
<tr>
<td>3</td>
<td>char[]</td>
<td>tibrvMsg_AddString</td>
<td>file data</td>
</tr>
</tbody>
</table>

For each record in the message, repeat fields 2 and 3.

Binary File Attachment - "ATB"

This message type is recognized by the "ATB" field name in all fields. The following table shows the layout of the binary file attachment body.

<table>
<thead>
<tr>
<th>Field ID</th>
<th>Field Type</th>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>int</td>
<td>tibrvMsg_AddI32()</td>
<td>number of records in this message</td>
</tr>
<tr>
<td>2</td>
<td>long</td>
<td>tibrvMsg_AddI32()</td>
<td>length of data in field 3</td>
</tr>
<tr>
<td>3</td>
<td>tibrv_u8</td>
<td>tibrvMsg_AddOpaque</td>
<td>file data</td>
</tr>
</tbody>
</table>

For each record in the message, repeat fields 2 and 3.

Last Message of Attachment - "LST"

All attachments must end with an "LST" message. This message type is recognized by the "LST" field name in all fields. This message type contains a count of the number of messages sent for the attachment, not including itself.

The following table shows the layout of the last message.
Attachment Error Handling

Transfer Errors: Queue versus Point-To-Point

When sending a message to a message queue, all of the attachments (along with the message) are transferred to the queue when the _SEND_ or _SENDLIST_ is invoked. The attachments are stored at the domain server until they are fetched by a user. If an error occurs while you send the attachments to the queue, then neither the message nor the attachments are delivered to the queue. In this scenario, the return code from _SEND_ or _SENDLIST_ is set to _SEATTXF. This error indicates that neither the message nor the attachments were delivered because one or more errors occurred during attachment transfer.

When a message is sent using point-to-point messaging, only the attachment list, along with the message, is sent to the receiving side initially. The receiver is then responsible for determining which, if any, attachments should actually be transferred. Because the message is delivered to the receiver before any attachments are actually transferred, an error encountered during attachment transfer will not cause the _SEND_ to terminate. If an error is encountered, then the current attachment transfer is terminated, but the remaining attachments selected to be received are sent to the receiving side. If any errors are encountered during attachment transfer, the return code from _SEND_ or _SENDLIST_ is set to _SWATTXF. This is only a warning indicating that the message was successfully sent, but one or more errors occurred during attachment transfer.

Accept Errors

When a message includes attachments, the receiver has the responsibility to determine which attachments are ultimately transferred, via the _ACCEPT_ATTACHMENT_ method. If an error is encountered during attachment transfer, then the current attachment transfer is terminated, but the transfer continues with the next attachment in the attachlist. If any errors are encountered, then the return code from _ACCEPT_ATTACHMENT_ is set to _SWATTXF. This is only a warning indicating that one or more errors occurred during attachment transfer.
Attachment Error Codes

List of Attachment Error Codes

To review what was mentioned above, a specific return code is set if an error is encountered during attachment transfer:

- When sending on a Cnction instance, _SWATTXF is returned.
- When sending on a Queue instance, _SEATTXF is returned.
- When accepting attachments on either a Queue or Cnction instance, _SWATTXF is returned.

When one of these scenarios occurs, the attachlist parameter passed to these methods is updated. An additional named item, RC, is added to each separate attachment list. The value of RC will be a numeric return code that can be used to determine what caused the error for this particular attachment transfer. The defined return codes include the following:

<table>
<thead>
<tr>
<th>Input File Errors (error occurred on input file):</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>20</td>
<td>general I/O error</td>
</tr>
<tr>
<td>21</td>
<td>libname does not exist</td>
</tr>
<tr>
<td>22</td>
<td>memname does not exist</td>
</tr>
<tr>
<td>23</td>
<td>invalid or missing password</td>
</tr>
<tr>
<td>24</td>
<td>invalid data set option value</td>
</tr>
<tr>
<td>25</td>
<td>invalid data set option name</td>
</tr>
<tr>
<td>26</td>
<td>general error parsing data set options</td>
</tr>
<tr>
<td>27</td>
<td>error parsing where stmt</td>
</tr>
<tr>
<td>28</td>
<td>bad physical filename</td>
</tr>
<tr>
<td>29</td>
<td>file in use</td>
</tr>
<tr>
<td>30</td>
<td>file does not exist</td>
</tr>
<tr>
<td>31</td>
<td>invalid authorization for external file</td>
</tr>
<tr>
<td>32</td>
<td>open failed for some reason other than mentioned above</td>
</tr>
<tr>
<td>33</td>
<td>error obtaining Integrity Constraints information</td>
</tr>
<tr>
<td>34</td>
<td>variable contains unsupported characters or is too long</td>
</tr>
<tr>
<td>35</td>
<td>key name contains unsupported characters or is too long</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Output File Errors (error occurred on output file):</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>80</td>
<td>general I/O error</td>
</tr>
<tr>
<td>81</td>
<td>libname does not exist</td>
</tr>
<tr>
<td>82</td>
<td>invalid or missing password</td>
</tr>
<tr>
<td>83</td>
<td>bad physical filename</td>
</tr>
<tr>
<td>84</td>
<td>file in use</td>
</tr>
<tr>
<td>85</td>
<td>file does not exist</td>
</tr>
</tbody>
</table>
invalid authorization for external file
open failed for some reason other than
  mentioned above
file already exists
engine does not support read passwords
engine does not support encryption

General/Misc. Errors:

<table>
<thead>
<tr>
<th>Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Out of memory error</td>
</tr>
<tr>
<td>2</td>
<td>Open of catalog by queue manager failed</td>
</tr>
<tr>
<td>3</td>
<td>Read error (of catalog) encountered by queue manager</td>
</tr>
<tr>
<td>4</td>
<td>Write error (of catalog) encountered by queue manager</td>
</tr>
<tr>
<td>5</td>
<td>Index create failure</td>
</tr>
<tr>
<td>6</td>
<td>Backwards compatibility error</td>
</tr>
<tr>
<td>7</td>
<td>Only SQL views supported</td>
</tr>
</tbody>
</table>

Example

In the following example, one attachment is accepted into a non-existent library name:

```c
/* build one attachment list, att1 */
att1 = makelist();
rc = setnitemc(att1, 1, "ATTACH_ID");
rc = setnitemc(att1, "NOEXIST", "OUTLIB");
rc = setnitemc(att1, "A", "OUT");

/* insert att1 into the main attachment list, alist */
alist = makelist();
alist = insertl(alist, att1, -1);```

```c
/* accept the attachment */
call send(obj, "_ACCEPT_ATTACHMENT_", alist, rc);

/* if error, dump out attachment list to view rc */
if (rc NE 0) then
  call putlist(alist, "Attachment list after accept:", 1);
```

After the accept method call, the attachment list alist has the following named items:

- Named item ATTACH_ID has a value of 1.
- Named item OUTLIB has a value of "NOEXIST."
- Named item OUT has a value of "A."
- Named item RC has a value of 81.

The error code list maps the return code of 81 into output library is nonexistent. Similarly, when the sender returns from the _SEND_ or _SENDLIST_, the attachlist parameter is updated with the RC named item to reflect that the attachment transfer failed.
att1 = makelist();
rc = setnitemc(att1, "SASUSER", "LIBNAME");
rc = setnitemc(att1, "NAMES", "MEMNAME");
rc = setnitemc(att1, "DATASET", "TYPE");
attachlist = makelist();
attachlist = insertl(attachlist, att1, -1);
call send(cnctionObj, "_SEND_", msgtype, attachlist,
   rc, "Message One");
if (%sysrc(_SWATTXF) = rc) then do;
   call putlist(attachlist, "attachlist after send", -1);
end;

Assuming that the attachment was accepted by the receiving side as shown above,
the attachment list, attachlist is updated with the RC named item to reflect that the
attachment transfer failed.

- Named item LIBNAME has a value of "SASUSER."
- Named item MEMNAME has a value of "NAMES."
- Named item TYPE has a value of "DATASET."
- Named item RC has a value of 81.

Again, the error code list maps the return code of 81 into output library is
nonexistent.
SAS CALL Routines for the Common Messaging Interface

This section documents all of the available CALL routines within the common messaging interface. The beginning of the documentation for each CALL indicates which transports are supported. Within the CALL routines and CALL documentation, the term MQSeries is used to refer to WebSphere MQ. When support for MQSeries (now known as WebSphere MQ) is noted, this includes both MQSeries Base/Server and MQSeries Client.
CALL ABORT Routine

Cancels prior work that has been done via a transaction object.

Supports: MQSeries, MQSeries C, MSMQ

Syntax

CALL ABORT(transid, rc);

Required Arguments

**transid**

Specifies the handle to a transaction object that is obtained from the CALL BEGINTRANSACTION routine.

Type: Numeric, Input

**rc**

Provides the return code from the CALL routine. If an error occurs, then the return code is nonzero. You can use the SAS function SYSMSG() in order to obtain a textual description of the return code.

Type: Numeric, Output

Details

For MQSeries, all transactions are associated with a particular queue manager. So when you cancel a unit of work that is associated with a particular queue manager, all work performed by that particular queue manager under synchpoint control is canceled at once. You can associate more than one transaction object with the same queue manager, but it is not a good practice. Under MSMQ, all transaction objects are autonomous.

Example

The following example cancels the processing of a transactional unit of work:

```
length msg $ 200;
length transid rc 8;
rc=0;
call abort(transid, rc);
if rc ^= 0 then do;
   put 'ABORT: failed';
   msg = sysmsg();
   put msg;
```
CALL ACCEPTATTACHMENT Routine

Accepts an attachment by re-creating it on the local machine.

Supports: MQSeries, MQSeries C, MSMQ, Rendezvous, Rendezvous-CM

Syntax

CALL ACCEPTATTACHMENT(qid, attachid, qual1, qual2, rc);

Required Arguments

qid
  Specifies the handle of an open queue that is obtained from a previous CALL OPENQUEUE routine.
  Type  Numeric, Input

attachid
  Specifies an attachment identifier that is obtained from a previous CALL GETATTACHMENT routine.
  Type  Numeric, Input

qual1
  Specifies the first attachment qualifier. If this is an external file attachment, then this qualifier designates the file specification that is used to receive it (either FILENAME or FILEREF). Otherwise, this qualifier designates the receiving library name.
  Type  Character, Input

qual2
  Specifies the second attachment qualifier. If this is an external file attachment, then this qualifier designates the receiving filename or fileref. Otherwise, this qualifier designates the receiving member name.
  Type  Character, Input

rc
  Provides the return code from the CALL routine. If an error occurs, then the return code is nonzero. You can use the SAS function SYSMSG() in order to obtain a textual description of the return code.
  Type  Numeric, Output

Details

For information about exception processing when you use attachments, see "Attachment Error Handling" on page 201.
Example

This example accepts attachments from a message and stores them in the file d:\myexternalfilename.tmp.

```plaintext
length msg $ 200;
length qid lastflag attachid rc 8;
length type $ 13;
length qual1 qual2 $ 80;
length desc $ 80;
length minor major 8;
next:
    rc=0;
    lastflag=0;
    attachid=0;
    type='';
    qual1='';
    qual2='';
    desc='';
    minor=0;
    major=0;
    call getattachment(qid, lastflag, attachid, type,
        qual1, qual2, rc, desc, minor, major);
if rc ^= 0 then do;
    put 'GETATTACHMENT: failed';
    msg = sysmsg();
    put msg;
end;
else do;
    put 'GETATTACHMENT: succeeded';
    put 'Attachment type is ' type;
    if type eq 'EXTERNAL_TEXT' OR type eq
        'EXTERNAL_BIN' then do;
        put "Sender's " qual1 " was " qual2;
        /* accept/receive the external attachment */
        call acceptattachment(qid, attachid, 'filename',
            'd:\myexternalfilename.tmp', rc);
    if rc ^= 0 then do;
        put 'ACCEPTATTACHMENT: failed';
        msg = sysmsg();
        put msg;
    end;
else
    put 'ACCEPTATTACHMENT: succeeded';
end;
else do;
    put "Sender's library name was ' qual1;
    put "Sender's member name was ' qual2;
    /* accept/receive the library/member */
    libname tmp 'd:\tmp';
    call acceptattachment(qid, attachid,
        'tmp', 'test', rc);
end;
if lastflag eq 0 then goto
next;
```
CALL BEGINTRANSACTION Routine

Begins transaction processing by creating a transaction object.

Supports: MQSeries, MQSeries C, MSMQ

Syntax

CALL BEGINTRANSACTION(transid, tid, rc);

Required Arguments

transid

Returns a handle to a transaction object that is generated for committing and canceling transactional processing, as well as freeing the resources that are associated with the transaction object.

Type Numeric, Output

tid

Specifies the transport handle that is obtained from the CALL INIT routine.

Type Numeric, Input

rc

Provides the return code from the CALL routine. If an error occurs, then the return code is nonzero. You can use the SAS function SYSMSG() in order to obtain a textual description of the return code.

Type Numeric, Output

Details

The created transaction object is used to commit or cancel prior processing (CALL SENDMESSAGE and CALL RECEIVEMESSAGE routines) that use the transaction object as a message property. Transaction processing is supported only by the MQSeries, MQSeries C, and MSMQ transports.

Example

The following example begins a transaction:

```sas
length msg $ 200;
length transid tid rc 8;
rc=0;
transid=0;
call begintransaction(transid, tid, rc);
if rc ^= 0 then do;
   put 'BEGINTRANSACTION: failed';
   msg = sysmsg();
   put msg;
end;
```
CALL CLOSEQUEUE Routine
Closes a message queue.

Supports: MQSeries, MQSeries C, MSMQ, Rendezvous, Rendezvous-CM

Syntax

CALL CLOSEQUEUE(qid, rc<, attr<, attr2>>);

Required Arguments

qid
Specifies the handle of a queue that is obtained from a previous CALL OPENQUEUE routine.
Type Numeric, Input

rc
Provides the return code from the CALL routine. If an error occurs, then the return code is nonzero. You can use the SAS function SYSMSG() in order to obtain a textual description of the return code.
Type Numeric, Output

Optional Arguments

attr
Specifies a delete attribute. The following attributes are valid:

DELETE
Specifies that the queue is to be deleted after it successfully closes, but only if there are no messages on the queue. This attribute is supported with MQSeries only. It is not supported with MSMQ because there is no way to programmatically determine the depth of the queue. It is not supported with Rendezvous because Rendezvous handles this function internally.

DELETE_PURGE
Causes the queue to be deleted, even if the queue depth is greater than zero. This attribute is supported with MQSeries, MQSeries C, MSMQ, and Rendezvous-CM. It is not supported with Rendezvous because Rendezvous handles this function internally. If you are using Rendezvous Certified Message Delivery, when you close a listener queue the default setting is for the sender to save messages for persistent messaging. If you do not want messages to be saved by the sender or do not want persistent messaging, specify the DELETE_PURGE attribute when you close the queue. Setting the DELETE_PURGE attribute is the same as setting the cancelAgreements argument on TIBRVCM_CANCEL(TRUE).

Type Character, Input
**attr2**
Specifies an attribute to recognize or report errors that can occur when using the DELETE_PURGE attribute with MSMQ. The following attributes are valid:

**RECOGNIZE_MSMQ_DELETE_ERRORS**
Recognizes MSMQ queue deletion errors by checking the return code for the CALL CLOSEQUEUE routine. When the CALL MQDELETEQUEUE routine encounters a failure, the return code output argument is set and you can use `sysmsg()` to obtain the error message.

**LOG_MSMQ_DELETE_ERRORS**
Logs MSMQ queue deletion errors to the App.Program logger. The return code argument is not set, which means that the program flow is not disrupted if there is an error.

**Type** Character, Input

---

**Example**
The following example closes a queue:

```plaintext
length msg $ 200;
length qid rc 8;
rc=0;
call closequeue(qid, rc);
if rc ^= 0 then do;
    put 'CLOSEQUEUE: failed';
    msg = sysmsg();
    put msg;
end;
else put 'CLOSEQUEUE: succeeded';
```

---

**CALL COMMIT Routine**
Commits prior work that has been done via a transaction object.

**Supports:** MQSeries, MQSeries C, MSMQ

**Syntax**

```plaintext
CALL COMMIT(transid, rc);
```

**Required Arguments**

**transid**
Specifies the handle to a transaction object that is obtained from the CALL BEGINTRANSACTION routine.

**Type** Numeric, Input
\textit{rc} provides the return code from the CALL routine. If an error occurs, then the return code is nonzero. You can use the SAS function \texttt{SYSMSG()} in order to obtain a textual description of the return code.

**Type**  
\texttt{Numeric, Output}

**Details**

For MQSeries, all transactions are associated with a particular queue manager. So when you commit a unit of work that is associated with a particular queue manager, all work that is performed by that particular queue manager under synchpoint control is committed at once. You can associate more than one transaction object with the same queue manager, but it is not a good practice. Under MSMQ, all transaction objects are autonomous.

**Example**

The following example commits a transactional unit of work for processing:

```sas
length msg $ 200;
length transid rc 8;
rc=0;
call commit(transid, rc);
if rc ^= 0 then do;
    put 'COMMIT: failed';
    msg = sysmsg();
    put msg;
end;
else put 'COMMIT: succeeded';
```

---

**CALL DELETEALIAS Routine**

Deletes a transport or queue alias definition from the information repository.

**Supports:** MQSeries, MQSeries C, MSMQ, Rendezvous, Rendezvous-CM

**Syntax**

\texttt{CALL DELETEALIAS(type, name, storage, rc)};

**Required Arguments**

\textbf{type}

Specifies the type of alias that is to be deleted. The following types are valid:

- \texttt{TRANSPORT}
- \texttt{QUEUE}

**Type**  
\texttt{Character, Input}
**name**
Identifies the transport alias or queue alias that is to be deleted.

*Type* Character, Input

**storage**
Specifies the location of the alias definition. The REGISTRY location is valid.

*Type* Character, Input

**rc**
Provides the return code from the CALL routine. If an error occurs, then the return code is nonzero. You can use the SAS function SYSMSG() in order to obtain a textual description of the return code.

*Type* Numeric, Output

**Example**
The following example deletes a queue alias from the SAS registry:

```sas
length msg $ 200;
length rc 8;
rc=0;
call deletealias('QUEUE', 'MYQUEUE', 'REGISTRY', rc);
if rc ^= 0 then do;
   put 'DELETEALIAS: failed';
   msg = sysmsg();
   put msg;
end;
else put 'DELETEALIAS: succeeded';
```

---

**CALL DELETEMAP Routine**

Deletes a map data descriptor definition from the information repository.

**Supports:** MQSeries, MQSeries C, MSMQ, Rendezvous, Rendezvous-CM

**Syntax**

```
CALL DELETEMAP(name, storage, rc);
```

**Required Arguments**

**name**
Identifies the map data descriptor that is defined by a previous CALL SETMAP routine.

*Type* Character, Input

**storage**
Specifies the location for the map definition. The REGISTRY location is valid.
Type Character, Input

rc
Provides the return code from the CALL routine. If an error occurs, then the return code is nonzero. You can use the SAS function SYSMSG() in order to obtain a textual description of the return code.

Type Numeric, Output

Example
The following example deletes a map data descriptor definition from the SAS registry:

```sas
length msg $ 200;
length rc 8;
rc=0;
call deletemap('MYMAP', 'REGISTRY', rc);
if rc ^= 0 then do;
  put 'DELETEMAP: failed';
  msg = sysmsg();
  put msg;
end;
else put 'DELETEMAP: succeeded';
```

CALL DELETEMODEL Routine
Deletes a dynamic creation queue model from the information repository.

Supports: MSMQ, Rendezvous, Rendezvous-CM

Syntax
```sas
CALL DELETEMODEL(transport, name, storage, rc);
```

Required Arguments

transport
Specifies the transport that is associated with this model. MSMQ, Rendezvous, and Rendezvous-CM are the only valid transports for this CALL routine.

Type Character, Input

name
Identifies the dynamic model.

Type Character, Input

storage
Specifies the location for the model definition. The REGISTRY location is valid.

Type Character, Input
rc
Provides the return code from the CALL routine. If an error occurs, then the return code is nonzero. You can use the SAS function SYSMSG() in order to obtain a textual description of the return code.

Type Numeric, Output

Example
The following example deletes an MSMQ model queue definition from the SAS registry:

```sas
length msg $ 200;
length rc 8;
rc=0;
call deletemodel('MSMQ', 'MYMODEL', 'REGISTRY', rc);
if rc ^= 0 then do;
   put 'DELETEMODEL: failed';
   msg = sysmsg();
   put msg;
end;
else put 'DELETEMODEL: succeeded';
```

CALL FREETRANSACTION Routine
Frees a transaction object and its associated resources.

Supports: MQSeries, MQSeries C, MSMQ

Syntax
CALL FREETRANSACTION(transid, rc);

Required Arguments

transid
Specifies the handle to a transaction object that is obtained from the CALL BEGINTRANSACTION routine.

Type Numeric, Input

rc
Provides the return code from the CALL routine. If an error occurs, then the return code is nonzero. You can use the SAS function SYSMSG() in order to obtain a textual description of the return code.

Type Numeric, Output

Example
The following example frees the resources that are associated with a transaction object:
length msg $ 200;
length transid rc 8;
rc=0;
call freetransaction(transid, rc);
if rc ^= 0 then do;
   put 'FREETRANSACTION: failed';
   msg = sysmsg();
   put msg;
end;
else put 'FREETRANSACTION: succeeded';

CALL GETALIAS Routine

Obtains the current definition of a transport alias or queue alias that is set by the CALL SETALIAS routine in the information repository.

Supports: MQSeries, MQSeries C, MSMQ, Rendezvous, Rendezvous-CM

Syntax

CALL GETALIAS(type, name, storage, rc, transport<, queue>);

Required Arguments

type
   Specifies the type of alias. The following types are valid:
      ■ TRANSPORT
      ■ QUEUE

Type Character, Input

name
   Identifies the transport alias or queue alias that is set by the CALL SETALIAS routine.

Type Character, Input

storage
   Specifies the location for the alias definition. The REGISTRY location is valid.

Type Character, Input

rc
   Provides the return code from the CALL routine. If an error occurs, then the return code is nonzero. You can use the SAS function SYSMSG() in order to obtain a textual description of the return code.

Type Numeric, Output

transport
   Returns the transport name.
CALL GETATTACHMENT Routine

Gets attachment information that is associated with a particular message.

Supports: MQSeries, MQSeries C, MSMQ, Rendezvous, Rendezvous-CM

Syntax

CALL GETATTACHMENT(qid, lastflag, attachid, type, qual1, qual2, rc
<, desc><, minor><, major>);

Required Arguments

qid
Specifies the handle of an opened queue obtained from a previous CALL
OPENQUEUE routine.

Type Numeric, Input
lastflag
Indicates whether you have reached the last attachment in a message. Possible values are as follows:

0
  Specifies that more attachments are to be presented.
1
  Specifies that this is the final attachment.

Type Numeric, Output

attachid
Returns an attachment identifier that is used with the CALL ACCEPTATTACHMENT routine when this attachment is accepted.

Type Numeric, Output

type
Returns the type of attachment. The following types are valid:
- EXTERNAL_TEXT
- EXTERNAL_BIN
- DATASET

Type Character, Output

qual1
Returns the first attachment qualifier. If this is an external attachment, then this qualifier designates the file specification that is used to send it (either FILENAME or FILEREF). Otherwise, this qualifier designates the sending library name.

Type Character, Output

qual2
Returns the second attachment qualifier. If this is an external attachment, then this qualifier designates the sending filename or fileref. Otherwise, this qualifier designates the sending member name.

Type Character, Output

rc
Provides the return code from the CALL routine. If an error occurs, then the return code is nonzero. You can use the SAS function SYSMSG() in order to obtain a textual description of the return code.

Type Numeric, Output

Optional Arguments
desc
Returns a description of the attachment if the sender provides one. This parameter is optional.

Type Character, Output

minor
Returns a user-specified minor version number. This parameter is optional.
Type Numeric, Output

**major**
Returns a user-specified major version number. This parameter is optional.

Type Numeric, Output

**Details**

You can repeatedly call this function until the final attachment has been presented.

---

**Note:** To receive an attachment from outside of the SAS environment, you must know the layout of an attachment. For more information, see the "Attachment Layout for WebSphere MQ and MSMQ" on page 184 and the "Attachment Layout for TIBCO Rendezvous" on page 188.

---

**Example**

The following example gets all of the attachment information from a message:

length msg $ 200;
length qid lastflag attachid rc 8;
length type $ 13;
length qual1 qual2 $ 80;
length desc $ 80;
length minor major 8;
next:
   rc=0;
   lastflag=0;
   attachid=0;
   type='';
   qual1='';
   qual2='';
   desc='';
   minor=0;
   major=0;
call getattachment(qid, lastflag, attachid, type,
   qual1, qual2, rc, desc, minor, major);
if rc ^= 0 then do;
   put 'GETATTACHMENT: failed';
   msg = sysmsg();
   put msg;
end;
else do;
   put 'GETATTACHMENT: succeeded';
   put 'Attachment type is ' type;
   if type eq 'EXTERNAL_TEXT' OR type eq 'EXTERNAL_BIN' then do;
      put "Sender's " qual1 " was " qual2;
      /* process external file... * /
   end;
   else do;
      put "Sender's library name was ' qual1;
      put "Sender's member name was ' qual2;
CALL GETMAP Routine

Obtains the current definition of a map data descriptor in the information repository.

Supports: MQSeries, MQSeries C, MSMQ, Rendezvous, Rendezvous-CM

Syntax

CALL GETMAP(name, storage, rc, descriptor);

Required Arguments

name
Identifies the map data descriptor that is defined by a previous CALL SETMAP routine.

Type: Character, Input

storage
Specifies the location for the map definition. The REGISTRY location is valid.

Type: Character, Input

rc
Provides the return code from the CALL routine. If an error occurs, then the return code is nonzero. You can use the SAS function SYSMSG() in order to obtain a textual description of the return code.

Type: Numeric, Output

descrptor
Returns a string that describes the layout of the data. The format of the descriptor is as follows:

"type,offset,length;type,offset,length;..."

where:

- type is the type of data (SHORT, LONG, DOUBLE, CHAR)
- offset is the offset from the beginning of the message which is the cursor location in the case of the CALL PARSEMESSAGE routine
- length is the length of the data which is valid only for CHAR data type

Type: Character, Output

Example

The following example obtains a map data descriptor definition in the SAS registry:
length msg $ 200;
length rc 8;
length descriptor $ 80;
rc=0;
descriptor='';
call getmap('MYMAP', 'REGISTRY', rc, descriptor);
if rc ^= 0 then do;
   put 'GETMAP: failed';
   msg = sysmsg();
   put msg;
end;
else do;
   put 'GETMAP: succeeded';
   put 'descriptor = ' descriptor;
end;

CALL GETMODEL Routine
For MSMQ, obtains a dynamic creation queue model from the information repository. For Rendezvous and Rendezvous-CM, obtains transport attributes.
Supports: MSMQ, Rendezvous, Rendezvous-CM

Syntax
CALL GETMODEL(transport, name, storage, rc, props, value1<, value2<, value3,
...>>)

Required Arguments
transport
   Specifies the transport that is associated with this model. MSMQ, Rendezvous, and Rendezvous-CM are the only valid transports for this CALL routine.
   Type   Character, Input

name
   Identifies the dynamic model.
   Type   Character, Input

storage
   Specifies the location for the model definition. The REGISTRY location is valid.
   Type   Character, Input

rc
   Provides the return code from the CALL routine. If an error occurs, then the return code is nonzero. You can use the SAS function SYSMSG() in order to obtain a textual description of the return code.
   Type   Numeric, Output
props
Identifies one or more properties to be queried.

Type Character, Input

value
Identifies one or more queue properties to be queried. This parameter is a character string with each applicable output variable separated by a comma.

Type Character or Numeric, Output

Details
You must associate a variable with each property that is identified by props.

For MSMQ, the following properties are valid:

AUTHENTICATE character
BASEPRIORITY numeric
JOURNAL character
JOURNALQUOTA numeric
LABEL character
PRIVLEVEL character
QUOTA numeric
TRANSACTION character
TYPE binary string

For Rendezvous and Rendezvous-CM, the following transport properties are valid:

DAEMON character
NETWORK character
SERVICE character

For Rendezvous-CM only, the following transport properties are valid:

CMNAME character
LEDGER character
RELAYAGENT character
REQUESTOLD character
SYNCLEDGER character

Example
The following example obtains an MSMQ model queue definition in the SAS registry:

length msg $ 200;
length rc 8;
length auth priv $ 10;
length label $ 80;
rc=0;
auth='';
priv='';
label='';
call getmodel('MSMQ', 'MYMODEL', 'REGISTRY', rc,
'AUTHENTICATE,PRIVLEVEL,LABEL', auth, priv, label);
if rc ^= 0 then do;
   put 'GETMODEL: failed';
   msg = sysmsg();
CALL GETQUEUEPROPS Routine

Gets information pertaining to a queue's properties and security.

Supports: MQSeries, MQSeries C, MSMQ, Rendezvous, Rendezvous-CM

Syntax

CALL GETQUEUEPROPS(qid, rc, ttype, pmask, depth, maxdepth, maxmsgl, ctime, desc <, inbox>);

Required Arguments

qid
Specifies the handle to an open queue that is obtained from a previous CALL OPENQUEUE routine.

Type Numeric, Input

rc
Provides the return code from the CALL routine. If an error occurs, then the return code is nonzero. You can use the SAS function SYSMSG() in order to obtain a textual description of the return code.

Type Numeric, Output

ttype
Identifies the transport type of the queue. Possible values are as follows:

- MQSeries
- MQSeries C
- MSMQ
- Rendezvous
- Rendezvous-CM

Type Character, Output

pmask
Returns the property assertion mask that the queue accepts. This property is valid only for the MQSeries, MQSeries C, and MSMQ transports. Possible values are as follows:
bit 0
In MSMQ, specifies that the queue only accepts authenticated messages.

bit 1
In MSMQ, specifies that the queue only accepts private messages.

bit 2
In MSMQ, specifies that the queue only accepts public messages.

bit 4
In MSMQ, specifies that the queue only accepts transactional messages. In MQSeries, bit 4 specifies that the QMgr supports synchpoint.

Type Numeric, Output

**depth**
Returns the current depth of the queue.

Type Numeric, Output

**maxdepth**
Returns the maximum depth that is configured for the queue. This property is valid only for the MQSeries, MQSeries C, and MSMQ transports.

Type Numeric, Output

**maxmsgl**
Returns the maximum length that is configured for the queue. This property is valid only for the MQSeries, MQSeries C, and MSMQ transports.

Type Numeric, Output

**ctime**
Returns the queue creation time stamp. This property is valid only for the MQSeries, MQSeries C, and MSMQ transports.

Type Character, Output

**desc**
Returns a description of the queue. This property is valid only for the MQSeries, MQSeries C, and MSMQ transports.

Type Character, Output

**Optional Argument**

**inbox**
Returns the name of the private inbox created for a session opened with FETCHX. This property is valid only for the Rendezvous transports. This parameter is optional.

Type Character, Output

**Details**
If a transport does not support a particular property, then the routine returns -2 for numeric property values but does not change character property values.
Example

The following example obtains the properties of a queue:

```
length msg $ 200;
length qid rc 8;
length ttype $ 13;
length pmask depth maxdepth maxmsgl 8;
length ctime desc $ 80;
rc=0;
ttype='';
pmask=0;
depth=0;
maxdepth=0;
maxmsgl=0;
ctime='';
desc='';
call getqueueprops(qid, rc, ttype, pmask, depth,
   maxdepth, maxmsgl, ctime, desc);
if rc ^= 0 then do;
   put 'GETQUEUEPROPS: failed';
   msg = sysmsg();
   put msg;
end;
else do;
   put 'GETQUEUEPROPS: succeeded';
   put 'transport type = ' ttype;
   if ttype eq 'MQSERIES' then do;
      if pmask='1...b then put 'Syncpoint is enabled';
      else put 'Syncpoint is disabled';
   end;
   else if ttype eq 'MSMQ' then do;
      if pmask='1'b then put 'Authenticated
         messages are required';
      if pmask='1.'b then put 'Private
         messages are required';
      else if pmask='1..b then put 'Public
         messages are required';
      else put 'Privacy is optional';
      if pmask='1...b then put 'Transactional
         messages are required';
      else put 'Transactional messages
         are not permitted';
   end;
   put 'depth = ' depth;
   put 'maxdepth = ' maxdepth;
   put 'maxmsgl = ' maxmsgl;
   put 'creation time = ' ctime;
   put 'description = ' desc;
end;
```
CALL INIT Routine

Initializes a particular transport. You must use the CALL TERM routine to terminate the transport after you have completed a session.

Supports: MQSeries, MQSeries C, MSMQ, Rendezvous, Rendezvous-CM

Syntax

CALL INIT(tid, tname, rc);

Required Arguments

* tid*

Returns the transport handle that is used to open a queue or to begin transaction processing.

Type Numeric, Output

* tname*

Specifies the name of the transport that is initialized. The following transport names are valid:

- MQSERIES (trantab=SAS_trantab_override)
- MQSeries C (trantab=SAS_trantab_override)
- MSMQ
- RENDEZVOUS
- RENDEZVOUS-CM
- alias that is defined in the information repository

Note: With the MQSeries transport, if you use SAS to perform the conversion instead of using an MQSeries conversion exit, then you can specify which TRANTAB to use for converting the application data.

Type Character, Input

* rc*

Provides the return code from the CALL routine. If an error occurs, then the return code is nonzero. You can use the SAS function SYSMSG() in order to obtain a textual description of the return code.

Type Numeric, Output

Details

The following transports are valid: MQSeries (MQSeries Base/Server), MQSeries C (MQSeries Client), MSMQ (Microsoft Message Queue), RENDEZVOUS (TIBCO Rendezvous), and RENDEZVOUS-CM (TIBCO Rendezvous Certified Message...
Delivery). In addition, you can use a transport alias name that is defined in the information repository to indirectly specify one of the transports.

Example
The following example initializes an MQSeries Base/Server transport:

```plaintext
length msg $ 200;
length tid rc 8;
tid=0;
rc=0;
call init(tid, 'MQSERIES', rc);
if rc ^= 0 then do;
  put 'INIT: failed';
  msg = sysmsg();
  put msg;
end;
else put 'INIT: succeeded';
```

CALL OPENQUEUE Routine
Opens a message queue. You must use the CALL CLOSEQUEUE routine to close the message queue.

Supports: MQSeries, MQSeries C, MSMQ, Rendezvous, Rendezvous-CM

Note: For Rendezvous Certified Message Delivery (Rendezvous-CM), you must define a model definition for certified message delivery. Use the CALL SETMODEL routine to define a model definition.

Syntax

```plaintext
CALL OPENQUEUE(qid, tid, qname, mode, rc<, attr1<, attr2>>);
```

Required Arguments

- **qid**
  - Returns the queue handle for the opened queue. This handle is used in subsequent calls to send, receive, and parse messages and attachments, and close the queue.
  
  **Type**: Numeric, Output

- **tid**
  - Specifies the transport handle that is obtained from the CALL INIT routine. If transport handle is set to 0, then qname is assumed to be a queue alias name that is defined in the information repository, and the transport is initialized (and terminated at close) automatically.

  **Type**: Numeric, Input

- **qname**
  - Specifies the name of the queue to open.
The syntax for an MQSeries transport is:

MQSeries:Qmgr:Queue

The syntax for an MSMQ transport is:

MSMQ:
PathName | FormatName

The following PathName representations are valid:

- `machineName\QueueName` (public queue)
- `machineName\QueueName;Journal` (public queue's journal)
- `machineName\PRIVATE$\QueueName` (private queue)
- `machineName\PRIVATE$\QueueName;Journal` (private queue's journal)
- `machineName\Journal` (machine journal queue)
- `machineName\DeadLetter` (machine deadletter queue)
- `machineName\DeadXACT` (machine transaction deadletter queue)

Note: `machineName` can be substituted with "." to designate the local machine.

The following FormatName representations are valid:

- `PUBLIC=QueueGUID` (public queue)
- `PUBLIC=QueueGUID;Journal` (public queue's journal)
- `PRIVATE=machineGUID\QueueNumber` (private queue)
- `PRIVATE=machineGUID\QueueNumber;Journal` (private queue's journal)
- `DIRECT=AddressSpecification\QueueName` (direct format for public queue)
- `DIRECT=AddressSpecification\PRIVATE$\QueueName` (direct format for private queue)

where `AddressSpecification` is `protocol:address` (for example, `tcp:10.26.1.177`).

You can use direct format in certain situations. Consult MSMQ documentation for details. You can also use a queue alias name that is defined in the information repository as the `qname` parameter.

The syntax for a Rendezvous or Rendezvous-CM transport is:

SubjectName | InboxName

**SubjectName** consists of one or more elements separated by dot characters (periods). The elements can represent a subject name hierarchy. For example:

RUN.HOME
RUN.for.Elected_office.President

**InboxName** is generated by the Rendezvous software. The syntax is the same as **SubjectName**, but must begin with `_INBOX` as the first element.
If an inbox name is specified, the name must have already been created and
returned by another call. For example, a CALL RECEIVEMESSAGE routine
might have returned an inbox name in its respq attribute. When the queue is
being opened for sending, wildcard characters ("*" and ">") are not allowed.

Type   Character, Input

mode
Identifies the operational mode of the queue that is opened. You can use only
one mode to open a queue.

The following modes for the MSMQ and MQSeries transports are valid:

DELIVERY
   Enables messages to be sent to a queue

FETCH
   Enables messages to be destructively retrieved

FETCHX
   The same as FETCH except it ensures exclusive usage

BROWSE
   Enables messages to be nondestructively retrieved.

The following modes for the Rendezvous and Rendezvous-CM transport are
valid:

DELIVERY
   enables messages to be sent to a queue.

FETCH
   enables messages to be retrieved.

FETCHX
   same as FETCH except used for point-to-point or private messages (using
   inboxes) instead of broadcast messages (using subject names). The qname
   property must be left blank ("") on the open call. A private inbox name is
generated and associated with the qid. To access this queue, retrieve the
inbox name by using the CALL GETQUEUEPROPS routine. Use the value
returned as the response queue value on send message calls when notifying
a partner application of the private inbox name to send responses to. For
Rendezvous-CM, if persistent messaging is not required, then you can use
the FETCHX mode. The FETCHX mode should not be used with persistent
messaging because inbox names do not survive transport invalidation.

REQUEST
   enables request messages to be sent to a subject (queue) that is being
   monitored by a remote program that serves as an information supplier. The
   qname parameter should specify the name of the queue to which the request
   message is to be sent. Any responses received arrive on the queue that is
   specified in the respqueue parameter of the CALL SENDMESSAGE routine.

REQUESTX
   same as REQUEST except used for point-to-point or private messages
   (using inboxes) instead of broadcast messages (using subject names). The
   qname parameter should specify the name of the queue on which the request
   message is to be sent. Any responses received use the inbox name
   associated with the qid. This inbox name is created internally by Rendezvous
   when the respqueue parameter is initialized to null. For Rendezvous-CM, if
   persistent messaging is not required, then you can use the REQUESTX
mode. The REQUESTX mode should not be used with persistent messaging because inbox names do not survive transport invalidation.

Note: Before any messages are sent with the Rendezvous transport, the queues that receive the messages must be running and must have a listener (that is, the queues must be opened for FETCH, FETCHX, REQUEST, or REQUESTX). Otherwise, data will be lost. Queues that are opened for REQUEST and REQUESTX automatically have their receiving (response) queues open to listen for incoming messages when the initial request is sent.

**Type** Character, Input

**rc**

Provides the return code from the CALL routine. If an error occurs, then the return code is nonzero. You can use the SAS function SYSMSG() in order to obtain a textual description of the return code.

**Type** Numeric, Output

Optional Argument

**attr**

Specifies one or more attributes to be associated with the queue. Each attribute constitutes a separate parameter in the open call. The following attributes are valid:

**POLL (Timeout=wait_period_in_seconds)**

Enables you to specify how message reception is handled for this queue. By default, the time-out period is set to INFINITE and a receive is blocked until a message arrives. To override the default, specify POLL and the time-out period.

**DYNAMIC (Model=model_name)**

Signifies that the queue is to be dynamically created, and specifies a model name that is defined in the information repository, which specifies how to create the queue. For the MQSeries transport, the model is defined in the MQSeries configuration, not in the SAS information repository.

**CLUSTER (CLUSTER=BIND(bind_type))**

Allows for setting of open options that enables MQSeries to connect to clusters. This attribute is valid only for MQSeries. Values for *bind_type* can be:

- **OPEN**
  translates to MQOO_BIND_ON_OPEN

- **NOT_FIXED**
  translates to MQOO_BIND_NOT_FIXED

- **AS_Q_DEF**
  translates to MQOO_BIND_AS_Q_DEF

**Type** Character, Input

Example

The following example opens a queue for delivery by using an alias name:
length msg $ 200;
length qid tid rc 8;
/* MYQUEUE exists as a queue alias definition
in the SAS information repository. */
rc=0;
qid=0;
tid=0;
call openqueue(qid, tid, 'MYQUEUE',
'DELIVERY', rc, "POLL(Timeout=5)");
if rc ^= 0 then do;
   put 'OPENQUEUE: failed';
   msg = sysmsg();
   put msg;
end;
else put 'OPENQUEUE:
succeeded';

CALL PARSEMESSAGE Routine

Parses a message body that has been received.
Supports: MQSeries, MQSeries C, MSMQ, Rendezvous, Rendezvous-CM

Syntax

CALL PARSEMESSAGE(qid, cursor, rc, map, data);

Required Arguments

qid
   Specifies the handle of an open queue that is obtained from a previous CALL
OPENQUEUE routine.
   Type  Numeric, Input

cursor
   Sets the cursor to zero in order to parse from the beginning. Upon return, the
cursor is positioned at the next data location, according to the specified map.
   Type  Numeric, Input or Output

rc
   Provides the return code from the CALL routine. If an error occurs, then the
return code is nonzero. You can use the SAS function SYSMSG() in order to
obtain a textual description of the return code.
   Type  Numeric, Output

map
   Specifies the map data descriptor that is defined by a previous CALL SETMAP
routine.
   Type  Character, Input
**data**

Identifies the data to be parsed from the internal receive buffer.

**Type** Character or Numeric, Output

**Example**

The following example parses a message:

```plaintext
length msg $ 200;
length qid rc attchflg 8 event $ 10;
length msgtype 8 corrid $ 48 map $ 80;
length employee $ 20 id 8;
rc=0;
map='employeerecord';
/* data descriptor defined in repository...
ie., "char,20;double" */
cursor=0;
call parsemessage(qid, cursor, rc, map, employee, id);
if rc ^= 0 then do;
   put 'PARSEMESSAGE: failed';
   msg = sysmsg();
   put msg;
   end;
else do;
   put 'PARSEMESSAGE: succeeded';
   put 'employee = ' employee;
   put 'id = ' id;
   end;
```

---

**CALL RECEIVEMESSAGE Routine**

Receives a message and optional attachments from a queue.

**Supports:** MQSeries, MQSeries C, MSMQ, Rendezvous, Rendezvous-CM

**Syntax**

```plaintext
CALL RECEIVEMESSAGE(qid, rc, event, attchflg, props<, value1, value2,...<data1, data2,...>>);
```

**Required Arguments**

**qid**

Specifies the handle of an open queue that is obtained from a previous CALL OPENQUEUE routine.

**Type** Numeric, Input
rc
Provides the return code from the CALL routine. If an error occurs, then the
return code is nonzero. You can use the SAS function SYSMSG() in order to
obtain a textual description of the return code.

Type  Numeric, Output

event
Contains a description of the event that occurs as a result of the message being
received. Possible event types are:

DELIVERY
  Specifies that the message was delivered.

NO_MESSAGE
  Specifies that no message is on queue.

ERROR
  Specifies that an error has occurred. This event results in a nonzero value for
  rc.

You need to initialize this parameter to a length of at least 10 before making the
call so that there is room for the value to be placed in the string. Otherwise, the
message might be truncated.

Type  Character, Output

attchflg
Indicates whether an attachment is associated with the received message.
Possible return values are as follows:

0
  Specifies that no attachments are associated with this message.

1
  Specifies that attachments are associated with this message. You can use
  the CALL GETATTACHMENT routine to receive the attachments.

Type  Numeric, Output

props
Identifies one or more message properties that are associated with the message
that is received. This parameter is a character string. Each property is separated
by a comma. The following receive message properties are valid for MQSeries:

- ACCOUNTINGTOKEN
- APPLIDENTITYDATA
- APPLORIGINDATA
- PUTAPPLNAME
- PUTAPPLTYPE

The following receive message properties are valid for MSMQ:

- ADMINQUEUE
- AUTHENTICATE
- DESCRIPTION
- SENDERCERT
The following receive message properties are valid for both MQSeries and MSMQ:

- CORRELATIONID
- FEEDBACK
- MAP
- MSGID
- MSGTYPE
- OPTIONS
- QUEUEDTIME
- RESPQUEUE
- TIMEOUT
- TRANSACTION
- USERID

The following receive message properties are valid for Rendezvous and Rendezvous-CM:

- MAP
- RESPQUEUE
- TIMEOUT

The following receive message properties are valid for Rendezvous-CM only:

- CERTIFIED
- RELAYAGENTACTION
- SENDERNAME

Type  Character, Input

Optional Arguments

**value**

Provides the values that are associated with each property that is specified via the `props` parameter. You must associate a value with each property that is identified with the `props` parameter. The property values can be an input, output, or both.

Descriptions and values for the received message properties are:

**ACCOUNTINGTOKEN**

Specifies an MQSeries accounting token.

Type  Binary string, Output

**ADMINQUEUE**

Specifies an MSMQ administrator queue.

Type  Character, Output

**APPLIDENTITYDATA**

Specifies MQSeries application identity data.
**APPLORIGINDATA**
Specifies MQSeries application origin data.

**AUTHENTICATE**
Indicates MSMQ authentication enablement. Possible authenticate return values are as follows:

**NO**
Specifies that the message was not authenticated.

**YES**
Specifies that the message was authenticated.

**CORRELATIONID**
Specifies a correlation identifier. For MQSeries and MSMQ transports, on input this property can be used for filtering purposes. However, do not try to filter with this property when you are receiving attachment messages. The original CORRELATIONID is not associated with the attachment header message, although the original CORRELATIONID is embedded within the attachment header itself and will be presented accurately. This type of processing is needed because an attachment consists of multiple messages that must be uniquely identified. A CORRELATIONID that is set by the application is not guaranteed to be unique.

**CERTIFIED**
Specifies a Certified Message (CM) indicator. Possible return values are as follows:

**NO**
Specifies that the message was received by the normal transport or the listener has not been certified.

**YES**
Specifies that the message was received within the certified delivery transport.

**DESCRIPTION**
Specifies a message description.

**FEEDBACK**
For MQSeries, specifies a feedback code. For MSMQ, specifies a class.

**MAP**
Specifies a data map name.
Type  Character, Input

**MSGID**
Indicates the message identifier. On input, this property can be used for filtering purposes for both MQSeries and MSMQ transports.

Type  Binary string, Input or Output

**MSGTYPE**
Indicates the message type.

Type  Numeric, Output

**OPTIONS**
Specifies the receive options. The following options are valid:

- **POSITIONFIRST**
  (MQSeries/MSMQ)
  Indicates to reposition to the first message in the queue.

- **CONVERSION_EXIT**
  (MQSeries only)
  Specifies to use the MQSeries conversion exit. Otherwise, SAS performs all necessary data conversion internally.

Type  Character, Input

**PUTAPPLNAME**
Indicates an MQSeries application name.

Type  Character, Output

**PUTAPPLTYPE**
Indicates an MQSeries application type.

Type  Character, Output

**QUEUEDTIME**
Indicates the time at which the message was queued.

Type  Character, Output

**RELAYAGENTACTION**
Specifies connect or disconnect actions for the relay agent. The following values are valid:

- **CONNECT**
  Indicates to connect to the relay agent before receiving messages and attachments.

- **DISCONNECT**
  Indicates to disconnect from the relay agent after all messages associated with the call have been processed. If an attachment is received, the disconnect call is issued after the CALL ACCEPTATTACHMENT routine has processed all of the messages associated with the attachment and before the call returns to the DATA step. If the CALL ACCEPTATTACHMENT routine is not used, then the connection is not
closed. If a connection was made to the relay agent during the call and an error occurs, then the error causes a disconnect from the relay agent.

**BOTH**
Indicates to connect to the relay agent, receive all messages, and then disconnect from the relay agent. If an attachment is received, the disconnect call is issued after the CALL ACCEPTATTACHMENT routine has processed all of the messages associated with the attachment and before the call returns to the DATA step. If CALL ACCEPTATTACHMENT routine is not used, then the connection is not closed. If an error occurs in a call, then if a connection was made to the relay agent during the call, an error causes a disconnect from the relay agent.

**Type** Character, Input

**RESPQUEUE**
Indicates the response queue name.

**Type** Character, Output

**SENDERCERT**
Indicates the subject within received certificate (MSMQ).

**Type** Character, Output

**SENDERNAME**
Indicates the name of the certified message (CM) transport used by the sender.

**Type** Character, Output

**TIMEOUT**
Specifies the number of seconds the CALL RECEIVEMESSAGE routine should wait for a message to arrive before returning. A value of -1 resets the queue to a non-polling state, and the CALL RECEIVEMESSAGE routine will wait indefinitely for a message to arrive. If the POLL attribute was not specified on a CALL OPENQUEUE routine, using this option on a CALL RECEIVEMESSAGE routine turns the queue into a polling queue that does not wait indefinitely for a message to arrive. You can turn a polling queue into a non-polling queue that waits indefinitely by specifying '-1' as the value of the TIMEOUT property on the CALL RECEIVEMESSAGE routine. By setting a TIMEOUT value on the CALL RECEIVEMESSAGE routine, the TIMEOUT value for the current queue ID is set to the new value, and all subsequent CALL RECEIVEMESSAGE routines will wait for the new time-out specified.

**Type** Numeric, Input

**TRANSACTION**
Indicates the transaction object obtained from the CALL BEGINTRANSACTION routine.

**Type** Numeric, Input

**USERID**
Indicates the user identifier who sent the message.

**Type** Character, Output
Type  Character or Numeric

data
When you issue the CALL RECEIVEMESSAGE routine, all data that is associated with a message is placed into an internal buffer. You can parse this data during the CALL RECEIVEMESSAGE routine with these optional parameters, or you can use the CALL PARSEMESSAGE routine at a later time to parse the data.

Type  Character or Numeric, Output

Example
The following example receives a message such as the one sent in the CALL SENDMESSAGE routine example:

    length msg $ 200;
    length qid rc attachflg 8 event $ 10;
    length msgtype 8 corrid $ 48 map $ 80;
    length employee $ 20 id 8;
    rc=0;
    corrid='';
    /* no filtering */
    map='employeerecord';
    /* data descriptor defined in repository...
       for example, "char,,20;double" */
    call receivemessage(qid, rc, event, attachflg,
       'MSGTYPE,CORRELATIONID,MAP', msgtype, corrid,
       map, employee, id);
    if rc ^= 0 then do;
       put 'RECEIVEMESSAGE: failed';
       msg = sysmsg();
       put msg;
    end;
    else do;
       put 'RECEIVEMESSAGE: succeeded';
       put 'Event = ' event;
       if event eq 'DELIVERY' then do;
          put 'Message has been delivered';
          if attachflg eq 1 then do;
             put 'Attachment(s) are associated with this message';
             /* process attachments...*/
          end;
          put 'employee = ' employee;
          put 'id = ' id;
       end;
    end;

CALL SENDMESSAGE Routine
Sends a message and optional attachments to a queue.
CALL SENDMESSAGE Routine

Synta

CALL SENDMESSAGE(qid, rc, props<, value1, value2,…<, data1, data2,…>>);

Required Arguments

qid
Specifies the handle of an open queue that is obtained from a previous CALL OPENQUEUE routine.

Type Numeric, Input

rc
Provides the return code from the CALL routine. If an error occurs, then the return code is nonzero. You can use the SAS function SYSMSG() in order to obtain a textual description of the return code.

Type Numeric, Output

props
Identifies one or more message properties that affect the message being sent. This parameter is a character string with each applicable property separated by a comma. All values except MSGID are input to the CALL SENDMESSAGE routine.

The following are valid send message properties for MQSeries:

- ACCOUNTINGTOKEN
- APPLIDENTITYDATA
- APPLORIGINDATA
- CODEDCHARSETID
- ENCODING
- FEEDBACK
- FORMAT
- PUTAPPLNAME
- PUTAPPLTYPE
- PUTDATE
- PUTTIME
- REPORT
- USERID

The following are valid send message properties for MSMQ:

- ACKNOWLEDGE
- ADMINQUEUE
- AUTHENTICATE
- DESCRIPTION
The following are valid send message properties for both MQSeries and MSMQ:

- ALLOWREADPROTECT
- ATTACHLIST
- CORRELATIONID
- MAP
- MSGID
- MSGTYPE
- PERSIST
- PRIORITY
- RESPQUEUE
- TIMEOUT
- TRANSACTION

The following are valid send message properties for Rendezvous and Rendezvous-CM:

- ATTACHLIST
- ALLOWREADPROTECT
- MAP
- RESPQUEUE

The following are valid send message properties for Rendezvous-CM only:

- ADDLISTENER
- ALLOWLISTENER
- DISALLOWLISTENER
- RELAYAGENTACTION
- TIMEOUT

**Optional Arguments**

**value**

Provides values that are associated with the properties specified via the `props` parameter. You must associate a value with each property that is specified by `props`. All values except MSGID are input to the routine. For the MQSeries transport, MSGID is input and output. For the MSMQ transport, MSGID is only output. Descriptions and values for the send message properties are listed by transport, and the following `values` are valid:
ACCOUNTINGTOKEN
MQSeries accounting token.

Type  Binary string

ACKNOWLEDGE
MSMQ acknowledgment types. Possible acknowledge types are as follows:

NONE (Default)
Specifies that no acknowledgment messages are posted.

FULL_REACH_QUEUE
Specifies that positive or negative acknowledgments are posted, depending on whether the message reaches the queue.

FULL_RECEIVE
Specifies that positive or negative acknowledgments are posted, depending on whether the message is retrieved from the queue.

NACK_REACH_QUEUE
Specifies that negative acknowledgments are posted when a message cannot reach the queue.

NACK_RECEIVE
Specifies that negative acknowledgments are posted when a message cannot be retrieved from the queue.

Type  Character

ADDLISTENER
Identifies one or more certified message names (CMNAMEs) of the listeners. This parameter is a character string with each CMNAME separated by a comma.

Anticipates a listener (or listeners) for certified delivery agreement.

Note: If a listener is added, then this feature applies to all future messages within the session.

Type  Character

ADMINQUEUE
Specifies the MSMQ administrator queue.

Type  Character

ALLOWLISTENER
Identifies one or more certified message names (CMNAMEs) of the listeners. This parameter is a character string with each CMNAME separated by a comma.

Allows listeners on the specified CMNAME to reinstate certified delivery. This feature overrides any DISALLOWLISTENER for listener CMNAME.

Note: If a listener is allowed, this feature applies to all future messages within the session.

Type  Character
ALLOWREADPROTECT
Specifies the value "YES". You must assert this property on read-protected data sets in order for that data set to be sent as an attachment. This ensures that the user realizes that the Read password and encryption attributes are not preserved when this data set is sent as a message attachment. If this property is not applied, then the CALL SENDMESSAGE routine fails when the user tries to send a Read protected data set, and an error is returned.

Note: This property is supported in SAS 8.1 and later. The password and encryption attributes are not preserved in the intermediate message format when the attachment is on a message queue. Because of this exposure, take care when sending password-protected or encrypted data sets as message attachments.

Type      Character
APPLIDENTITYDATA
Specifies the MQSeries application identity data.

Type      Character
APPLORIGINDATA
Specifies the MQSeries application origin data.

Type      Character
ATTACHLIST
Specifies that a list of attachments is included with message. The format of the list is as follows:

"type,qual1,qual2,options;
type,qual1,qual2,options;...

where the parameters are defined as follows:

type
  Is the attachment type, which can be one of the following:

  EXTERNAL_TEXT
    Is an external text file.

  EXTERNAL_BIN
    Is an external binary file.

  DATASET
    Is a SAS data set.

qual1
  Is a qualifier. For EXTERNAL_TEXT and EXTERNAL_BIN attachment types, this qualifier specifies the file specification type which can be one of the following:

    ■   FILENAME
    ■   FILEREF

For the DATASET attachment type, this qualifier specifies the library name.
qual2
Is a qualifier. For EXTERNAL_TEXT and EXTERNAL_BIN attachment
types, this qualifier specifies the actual filename or fileref. For the
DATASET attachment type, this qualifier specifies the member name.

options
Specifies optional attachment specifications. Multiple options must be
separated by spaces. The following options are valid for all attachment
types:
- DESC=attachment description
- MINOR=user specified minor version
- MAJOR=user specified major version

The following options are valid for the DATASET attachment type:
- DATASET_OPTIONS=data set options
- WHERE=WHERE clause
- INDEX=yes|no (default is yes so that indexes are sent)
- IC=yes|no (default is yes so that integrity constraints are sent)
- ATTACH_VERSION=VERSION_8

If the ATTACH_VERSION option is specified and value=VERSION_8,
then the data set is sent using the column types available in the data sets
before SAS®9. Use this option if you might be sending data sets to
another SAS session that is running SAS 8.2 or earlier.

If the ATTACH_VERSION option is omitted or if any other value is
specified, then the full data set, including all new types, is sent.

Type Character

AUTHENTICATE
Specifies MSMQ authentication enablement. Possible authenticate types are
as follows:
- NO (default)
  Specifies that no authentication is necessary. The message is not signed.
- YES
  Specifies that the message is signed and authenticated by the destination
  queue manager.

Type Character

CODEDCHARSETID
Specifies the MQSeries coded character set.

Type Numeric

CORRELATIONID
Specifies the correlation identifier.

Type Binary string

DESCRIPTION
Specifies the Message description.
DISALLOWLISTENER
Specifies one or more certified message names (CMNAMEs) of the listeners. This parameter is a character string with each CMNAME separated by a comma.
It cancels certified delivery to listeners with the specified CMNAME.

Note: If a listener is disallowed, this feature applies to all future messages within the session.

ENCODING
Specifies MQSeries data encoding.

ENCRYPT
Specifies MSMQ encryption enablement. Possible encryption types are as follows:

NO (Default)
Specifies that the message is to be sent as clear-text.

YES
Specifies end-to-end encryption of the message body.

ENCRYPTALG
Specifies the MSMQ encryption algorithms. The following choices are valid:

- RC2 (default)
- RC4

FEEDBACK
Specifies MQSeries feedback code.

FORMAT
Specifies MQSeries format name.

HASHALG
Specifies MSMQ hash algorithms. Possible hash types are as follows:

- MD2
- MD4
- MD5 (default)
Type Character

JOURNAL
  Specifies MSMQ journaling. Possible journal types are as follows:
  NO (default)
    Specifies that the message is not kept in the originating machine's journal queue.
  YES
    Specifies that the message is kept in the originating machine’s journal queue.
  DEADLETTER
    Specifies that the message is kept in a dead letter queue if it cannot be delivered.

Type Character

MAP
  Specifies the data map name.

Type Character

MSGID
  Specifies the message identifier.

Type Binary string

MSGTYPE
  Specifies the message type.

Type Numeric

PERSIST
  Specifies message persistence. Possible persist types are as follows:
  NO
    Indicates that the message is not persistent (default).
  YES
    Indicates that the message is persistent.

Type Character

PRIORITY
  Specifies message priority.

Type Numeric

PUTAPPLNAME
  Specifies MQSeries application name.

Type Character

PUTAPPLTYPE
  Specifies MQSeries application type.

Type Numeric
PUTDATE
  Specifies MQSeries put date.
  Type  Character

PUTTIME
  Specifies MQSeries put time.
  Type  Character

RELAYAGENTACTION
  Specifies the connect and disconnect actions for the relay agent. The following values are valid:

  CONNECT
  Indicates to connect to the relay agent before sending messages and attachments.

  DISCONNECT
  Indicates to disconnect from the relay agent after all messages associated with the call have been processed. The disconnect happens at the end of the call before the call returns to the DATA step.

  BOTH
  Indicates to connect to the relay agent, send all messages, and then disconnect from the relay agent. The disconnect happens at the end of the call before the call returns to the DATA step.

  Type  Character

REPORT
  Specifies the MQSeries reporting types. Possible report types are as follows:

  NONE
  Specifies that no reports are required.

  PASS_CORREL_ID
  Specifies to pass a correlation identifier.

  PASS_MSG_ID
  Specifies to pass a message identifier.

  COA
  Specifies that confirmation-on-arrival reports are required.

  COA_WITH_DATA
  Specifies that confirmation-on-arrival reports with data are required.

  COA_WITH_FULL_DATA
  Specifies that confirmation-on-arrival reports with full data are required.

  COD
  Specifies that confirmation-on-delivery reports are required.

  COD_WITH_DATA
  Specifies that confirmation-on-delivery reports with data are required.

  COD_WITH_FULL_DATA
  Specifies that confirmation-on-delivery reports with full data are required.

  EXPIRATION
  Specifies that expiration reports are required.
EXPIRATION_WITH_DATA
  Specifies that expiration reports with data are required.

EXPIRATION_WITH_FULL_DATA
  Specifies that expiration reports with full data are required.

EXCEPTION
  Specifies that exception reports are required.

EXCEPTION_WITH_DATA
  Specifies that exception reports with data are required.

EXCEPTION_WITH_FULL_DATA
  Specifies that exception reports with full data required.

DISCARD_MSG
  Specifies to discard message if it is undeliverable.

  Type  Character

RESPQUEUE
  Specifies the response queue name.

  Note: If this attribute is specified with an empty string value ("") when using a
Rendezvous or Rendezvous-CM queue that was opened using REQUESTX
mode, then the generated inbox name will be sent. If another name is
specified, it will be used instead.

  Type  Character

SENDERCERT
  Specifies the MSMQ certificate store name that is used in order to search for
external certificates. "MY" is typically specified. This results in a search of the
current user's certificates with their associated private keys. For example, if
"MY" is used, the corresponding registry entry is

  HKEY_CURRENT_USER\Software\Microsoft\SystemCertificates\MY

  Type  Character

TIMEOUT
  Specifies the time-out value in seconds.
  For Rendezvous-CM, specify this time-out as the length of time this message
is to be sent using certified message delivery.

  Type  Numeric

TRANSACTION
  Specifies the transaction object that is obtained from the CALL
BEGINTRANSACTION routine.

  Type  Numeric

USERID
  Specifies the MQSeries user identifier.

  Type  Character
Type Character or Numeric, Input or Output

data
  Specifies the individual pieces of data that are sent with the message.

Type Character or Numeric, Input

Details

If you intend to send attachments, use a queue that supports transactional processing. In this way, all messages associated with a failed attachment can be backed out if any part of the attachment processing fails. The IBM MQSeries queue manager supports the synchpoint function. An MSMQ queue is a transactional queue. For information about exception processing when using attachments, see "Attachment Error Handling" on page 201.

Before any messages are sent with the TIBCO Rendezvous transport, the queues that receive the messages must be running and must have a listener (that is, the queues must be opened for FETCH, FETCHX, REQUEST, or REQUESTX). Otherwise, data will be lost. Queues that are opened for REQUEST and REQUESTX automatically have their receiving (response) queues open to listen for incoming messages.

Note: If you are sending certified messages by using Rendezvous-CM, and plan to close the sending queue immediately after sending the message, then you might want to put a sleep() call in to sleep for a couple of seconds. This delay allows the Certified Delivery Agreement to be established between the sending transport and the receiving transport. This delay can also occur when a listener is first opened to receive certified messages.

Example

The following example sends an employee name and ID with records attached:

```plaintext
length msg $ 200;
length qid rc 8;
length msgtype 8 corrid $ 48 alist $ 80;
length employee $ 20 id 8;
rc=0;
  /* message properties */
  msgtype=1;
  corrid='0102030405060708090A0B0C0D0E0F';
  alist='DATASET,EMPLOYEE,RECORDS,
       DESC=employee records for John Doe';
  /* message data */
  employee='John Doe           ';
  id=9999;
call sendmessage(qid, rc,
                  'MSGTYPE,CORRELATIONID,ATTACHLIST',
                  msgtype, corrid, alist, employee, id);
if rc ^= 0 then do;
  put 'SENDMESSAGE: failed';
  msg = sysmsg();
  put msg;
end;
```
CALL SETALIAS Routine

Defines a transport or queue alias in the information repository.

Supports: MQSeries, MQSeries C, MSMQ, Rendezvous, Rendezvous-CM

Syntax

CALL SETALIAS(type, name, storage, rc, transport<, queue>);

Required Arguments

**type**

Specifies the type of alias to be defined. The following types are valid:

- TRANSPORT
- QUEUE

*Type Character, Input*

**name**

Identifies the transport alias or queue alias that is assigned.

*Type Character, Input*

**storage**

Specifies the location for the alias definition. The REGISTRY location is valid.

*Type Character, Input*

**rc**

Provides the return code from the CALL routine. If an error occurs, then the return code is nonzero. You can use the SAS function SYSMSG() in order to obtain a textual description of the return code.

*Type Numeric, Output*

**transport**

Identifies the name of the transport. The following transports are valid:

- MQSERIES (trantab=SAS_trantab_override)
- MQSeries C (trantab=SAS_trantab_override)
- MSMQ
- RENDEZVOUS
- RENDEZVOUS-CM

Note: With the MQSeries transport, if you use SAS to perform the conversion instead of using an MQSeries conversion exit, then you can specify which
TRANTAB to use for converting the application data. If the TRANTAB is not specified, SAS will use the session encoding information to convert the data.

**Type**  Character, Input

**Optional Argument**

*queue*

Identifies the name of the queue that is defined. This parameter is optional.

**Note:** This queue is valid only if a queue alias is being defined.

**Type**  Character, Input

**Details**

An alias provides a level of indirection that simplifies the programming interface by encapsulating information for all other programs. For details about administrator programs, see "Administrator Programs" on page 167.

**Example**

This example defines an MSMQ queue alias in the SAS registry.

```sas
length msg $ 200;
length rc 8;
rc=0;
call setalias('QUEUE', 'MYQUEUE', 'REGISTRY', rc,
'MSMQ', 'machine_name\queue_name');
if rc ^= 0 then do;
  put 'SETALIAS: failed';
  msg = sysmsg();
  put msg;
  end;
else put 'SETALIAS: succeeded';
```

**CALL SETMAP Routine**

Defines a map data descriptor in the information repository.

**Supports:** MQSeries, MQSeries C, MSMQ, Rendezvous, Rendezvous-CM

**Syntax**

```
CALL SETMAP(name, storage, rc, descriptor);
```
Required Arguments

**name**
Identifies the map data descriptor that is assigned.

  Type  Character, Input

**storage**
Specifies the location for the map definition. The REGISTRY location is valid.

  Type  Character, Input

**rc**
Provides the return code from the CALL routine. If an error occurs, then the return code is nonzero. You can use the SAS function SYSMSG() in order to obtain a textual description of the return code.

  Type  Numeric, Output

**descriptor**
Describes the layout of the data within a message body. This parameter is a string that contains the data type, the offset (optional), and (for character data) the length of each SAS variable. This data is presented in the order in which it is passed to the CALL SENDMESSAGE routine and returned from the CALL RECEIVEMESSAGE routine. The descriptor has the following format: where:

```
"type,offset,length;type,offset,length;..."
```

- **type** is the type of data (SHORT, LONG, DOUBLE, or CHAR).
- **offset** is the offset from the beginning of the message, which is the cursor location in the case of the CALL PARSEMESSAGE routine. This parameter is optional.
- **length** is the length of the data, which is valid only for the CHAR data type.

  Type  Character, Input

Details

A map specifies the layout of the data within a message body. Maps can be used with the MQSeries, MQSeries C, MSMQ, Rendezvous, or Rendezvous-CM transport when sending and receiving data.

Example

The following example defines a map data descriptor in the SAS registry:

```sas
length msg $ 200;
length rc 8;
rc=0;
call setmap('MYMAP', 'REGISTRY', rc,
'SHORT;LONG,2;SHORT;DOUBLE,6;CHAR,,50');
if rc ^= 0 then do;
  put 'SETMAP: failed';
  msg = sysmsg();
  put msg;
end;
else put 'SETMAP: succeeded';
```
CALL SETMODEL Routine

CALL SETMODEL Routine
For the MSMQ transport, defines a dynamic creation queue model. For the Rendezvous transport, the
CALL SETMODEL routine enables you to change one or more transport attributes from the default values.
For the Rendezvous-CM transport, defines a model definition for certified message delivery.

Supports: MSMQ, Rendezvous, Rendezvous-CM

Syntax

CALL SETMODEL(transport, name, storage, rc, props, value1<, value2, ...>)

Required Arguments

transport
Specifies the transport that is associated with this model. MSMQ, Rendezvous, and Rendezvous-CM are the only valid transports for this CALL routine.

Type Character, Input

name
Identifies the dynamic model or transport model that is assigned.

Type Character, Input

storage
Specifies the location for the model definition. The REGISTRY location is valid.

Type Character, Input

rc
Provides the return code from the CALL routine. If an error occurs, then the return code is nonzero. You can use the SAS function SYMSG() in order to obtain a textual description of the return code.

Type Numeric, Output

props
Identifies one or more properties that the queue exhibits once created. This parameter is a character string. Each applicable property is separated by a comma. You must associate a value with each property that is identified by props.

Type Character, Input

value
Inputs the values for each property that is specified. Use one of the following values for each of the properties listed in the props parameter.

AUTHENTICATE
Specifies whether the queue accepts only authenticated messages. The following values are valid:
NONE (Default)
   Specifies the queue accepts either authenticated or nonauthenticated messages.

ALWAYS
   Specifies the queue always requires authenticated messages.

Type   Character

BASEPRIORITY
   Specifies a single base priority for all messages sent to a public queue. Values range from -32768 to 32767, where 32767 is the highest priority, and 0 is the default priority.

Type   Numeric

JOURNAL
   Specifies whether messages retrieved from the queue are also copied to its journal queue. The following values are valid:

NONE (default)
   Indicates that messages that are removed from the queue are not stored in a journal.

ALWAYS
   Indicates that messages that are removed from the queue are always stored in its journal queue.

Type   Character

JOURNALQUOTA
   Specifies the maximum size (in kilobytes) of the journal queue. The default size is infinite.

Type   Numeric

LABEL
   Specifies a description of the queue. The default is a blank label (""").

Type   Character

PRIVLEVEL
   Specifies the privacy level that is required by the queue. The following values are valid:

NONE
   Specifies that the queue accepts only nonprivate (clear-text) messages.

BODY
   Specifies that the queue accepts only private (encrypted) messages.

OPTIONAL (default)
   Specifies that the queue accepts both private and nonprivate messages.

Type   Character

QUOTA
   Specifies the maximum size (in kilobytes) of the queue. The default size is infinite.
TRANSACTION
Specifies whether the queue is a transactional queue or a nontransactional queue. The following values are valid:

NONE (default)
Indicates that the queue does not accept transactional operations.

ALWAYS
Indicates that all messages that are sent to the queue must be done through an MSMQ transaction.

TYPE
Specifies the type of service that is provided by the queue. The value of the TYPE property is a universal unique identifier (UUID) character string that represents binary data. The default is NULL_GUID.

SERVICE
Specifies the service name or port number. If you specify a null value, the transport creation function looks for the service name "rendezvous" and uses 7500 if "rendezvous" is not found. The TIBCO Rendezvous documentation strongly recommends that administrators define "rendezvous" as a service, especially if UDP port 7500 is already in use. For more information, consult the TIBCO Rendezvous documentation.

NETWORK
Specifies the network name, Host IP, host name, or other identifier of the network. For more information, see the TIBCO Rendezvous documentation.

DAEMON
Specifies the TCP socket number for a local daemon, or the remote host name and socket number for a remote daemon. For more information, consult the TIBCO Rendezvous documentation.

CMNAME
Specifies the reusable name of a certified message (CM) transport. This is the CM Correspondent name, which can be omitted if persistent correspondents are not required.

Note: A model is not required if you are using default Rendezvous values.

For Rendezvous-CM only, the following transport properties are valid:
**LEDGER**
Specifies the name of the file in which to store a file-based ledger. This property can be omitted if persistent correspondents are not required.

**RELAYAGENT**
Specifies the name of the relay agent. If you use this property, then it must be configured by the Rendezvous administrator.

**REQUESTOLD**
Indicates whether a persistent correspondent requires delivery of unacknowledged messages that were sent to a previous certified delivery transport with the same CMNAME. Possible types are as follows:
- **NO** (default)
  Specifies that the new CM transport does not require certified senders to retain unacknowledged messages. Certified senders can delete those messages from their ledgers.
- **YES**
  Specifies that the new CM transport requires certified senders to retain unacknowledged messages sent to this persistent correspondent. When the new CM transport begins listening to the appropriate subjects, the senders can complete delivery. It is an error to specify **YES** when CMNAME is null.

**SYNCLEDGER**
Specifies how to synchronize the ledger to its storage medium. Possible types are as follows:
- **NO** (default)
  Specifies that the operating system writes changes to the storage medium asynchronously.
- **YES**
  Specifies that the operations updating the ledger file do not return until the changes are written to the storage medium.

**Details**
Dynamic models for MQSeries are defined within its own configuration.

**Example**
The following example defines an MSMQ model queue in the SAS registry:

```plaintext
length msg $ 200;
```
length rc 8;
rc=0;
/* private queue model */
call setmodel('MSMQ', 'MYMODEL', 'REGISTRY', rc, 'AUTHENTICATE,PRIVLEVEL,LABEL', 'ALWAYS', 'BODY', 'Private dynamic queue');
if rc ^= 0 then do;
   put 'SETMODEL: failed';
   msg = sysmsg();
   put msg;
   end;
else put 'SETMODEL: succeeded';

CALL TERM Routine
Terminates a particular transport. If you initiate a transport with the CALL INIT routine, you must use the CALL TERM routine to terminate the transport after you have completed the session.
Supports: MQSeries, MQSeries C, MSMQ, Rendezvous, Rendezvous-CM

Syntax
CALL TERM(tid, rc);

Required Arguments

**tid**
Specifies the transport handle that is obtained from the CALL INIT routine.

Type Numeric, Input

**rc**
Provides the return code from the CALL routine. If an error occurs, then the return code is nonzero. You can use the SAS function SYSMSGS() in order to obtain a textual description of the return code.

Type Numeric, Output

Example
The following example terminates a transport:

length msg $ 200;
length tid rc 8;
rc=0;
call term(tid, rc);
if rc ^= 0 then do;
   put 'TERM: failed';
   msg = sysmsg();
   put msg;
   end;
else put 'TERM: succeeded';
Message Queue Polling

Overview of Message Queue Polling

Message Queue Polling Concepts

Message queue polling is a SAS feature that enables you to monitor a message queue and start SAS programs to fulfill requests in the queue. You can configure message queue polling for WebSphere MQ or for JMS, which includes any message-oriented middleware vendor that includes a JMS provider.

Message queue polling is performed by the message queue polling server. The message queue polling server is a specialized SAS server that monitors a queue and performs SAS processing on the messages in the queue. Message queue
polling servers are managed by the Object Spawner. The spawner creates polling server sessions as needed, and balances the workload between the server sessions.

Note: JMS does not support monitoring the depth of a queue. The object spawner always maintains the same number of polling server sessions.

For more information about using message queue polling, see “Configure Your Programs for Message Queue Polling” on page 268.

Overview of Configuring Message Queue Polling

To configure message queue polling, perform the following steps:

1. “Define a Queue Manager for WebSphere MQ”, or “Define a Queue Manager for JMS”.
2. “Define a Message Queue Polling Server”
3. For polling servers that interact with JMS destinations, “Configure Your Java Environment for the Object Spawner and the JMS Access Method”.
4. “Add the Polling Server to the Object Spawner Definition”
5. “Configure Your Programs for Message Queue Polling” on page 268

Define a Queue Manager for WebSphere MQ

To create a queue manager definition in the SAS Metadata Repository, perform the following steps:

1. In SAS Management Console, select the Server Manager and then select Actions ⇒ New Server. The New Server Wizard appears.
2. Select Queue manager for WebSphere MQ, and then click Next.
3. Specify a name and an optional description. Click Next.
4. Define the queues that are managed by the queue manager.
   To create a new queue, perform the following steps:
   a. Click New. The New Queue window appears.
   b. Specify a name and an optional description.
   c. Click OK to create the queue and return to the New Server Wizard.
5. Move the queues that you want to associate with the queue manager from the Available items pane to the Selected items pane. Click Next.
Specify the host name and port number for the queue manager (the Authentication Domain field is not used by the object spawner). Click Next.

Review the information that you have entered, and then click Finish to create the queue manager definition.

---

Define a Queue Manager for JMS

To create a queue manager definition in the SAS Metadata Repository, perform the following steps:

1. In SAS Management Console, select the Server Manager and then select Actions → New Server. The New Server Wizard appears.

2. Select Queue manager for JMS, and then click Next.

3. Specify a name and an optional description. Click Next.

4. Specify the JNDI name for the connection factory, and then define the destinations that are managed by the queue manager.

   To create a new destination, perform the following steps:
   a. Click New. The New Destination window appears.
   b. Specify a name, an optional description, and the JNDI name for the destination. If the destination is a publishing topic, then select Destination is topic.
   c. Click OK to create the destination and return to the New Server Wizard.

5. Move the destinations that you want to associate with the queue manager from the Available items pane to the Selected items pane. Click Next.

6. Specify the JNDI initial context factory and the JNDI provider URL (the Authentication Domain field is not used by the object spawner). For more information about these options, see the descriptions for the JNDICTXFACTORY= and JNDIPROVIDERURL= options for the "FILENAME Statement: JMS Access Method" on page 273.

7. (Optional) Specify advanced connection options. For more information about connection options, see "CONNOPTIONS= "options-list"" on page 275.

8. When you have finished, click Next.

9. Review the information that you have entered, and then click Finish to create the queue manager definition.
Define a Message Queue Polling Server

To create a message queue polling server definition in the SAS Metadata Repository, perform the following steps:

1. In SAS Management Console, select the Server Manager and then select Actions ➔ New Server. The New Server Wizard appears.
2. Select Message Queue Polling Server and then click Next.
3. Specify a name and an optional description. Click Next.
4. Specify your configuration settings for the following fields:
   - **Command**
     - specifies a command that is used to invoke SAS and process messages. You can modify the command to include invocation options.
     - In your SAS command or in the script that you use to invoke SAS, you must specify a SAS program file by using the –SYSIN option. The SAS program that you specify should contain messaging code to read messages from the queue and process the message contents.
   - **Multiuser credentials**
     - select the credentials that are used to start SAS server sessions. The credentials that you specify must have permissions to access the resources, such as data libraries, that your SAS program will access.
     - If you select (None), then the object spawner's credentials are used to start the session.
   - **Server machine list**
     - specifies the machine where the polling server runs. The polling server must run on the same machine as an object spawner that it is associated with.
   - **Queue**
     - specifies the queue (or destination) that the polling server monitors for messages.
5. Click Advanced Options.
   - For JMS polling servers, specify the following options on the Polling tab:
     - **Maximum sessions**
       - specifies the maximum number of concurrent server sessions.
     - **Minimum sessions**
       - specifies the number of server sessions that are kept running. If you specify 0, then one server session is running.
       - **Note:** If the value of Minimum sessions is greater than the value of Maximum sessions, then the value of Maximum sessions is used.
     - **Queue polling process timeout**
       - specifies the time (in seconds) to wait for the server sessions to end when the object spawner is shutting down. If you specify a value that is greater
than zero, and any server sessions are still running after the time has elapsed, then the spawner terminates the sessions. If you specify 0, then there is no time limit for the server sessions to end. The default value is 0.

It is recommended that you use the default value of 0. Make sure that the code that is run by your server sessions checks for stop messages. See “Checking for Stop Messages” on page 269.

**Note:** If you specify a value that is greater than zero, then the spawner log might contain an error message, "Failed to locate the server indicated in the kill request" for each server session that ended normally. These messages do not indicate a problem.

**Note:** If you specify a value that is greater than zero, then the spawner always waits the full time-out period when shutting down. For example, if you specify 30 seconds as the time-out value, then the spawner always waits 30 seconds to shut down, even if all of the server sessions end before 30 seconds. If a spawner manages multiple polling servers, then the polling servers are shut down sequentially. The time delay for shutting down the spawner is cumulative.

For WebSphere MQ polling servers, specify the following options on the **Polling** tab:

**Message threshold**
specifies the maximum ratio of messages to server sessions. If the message threshold is exceeded, then the object spawner creates a new server session. The default value is 10 (10 messages to 1 server session).

For example, a polling server is configured with a message threshold value of 10. The message queue contains 21 messages, and two server sessions are running. Because the ratio of messages per server session (10.5) is greater than the threshold value (10), the object spawner creates a new server session.

**Queue polling timeout**
specifies the interval (in seconds) at which the server checks the depth of the message queue. The default value is 10.

**Maximum sessions**
specifies the maximum number of concurrent server sessions. If you specify 0, then an unlimited number of server sessions can be created. The default value is 1.

**Minimum sessions**
specifies the minimum number of server sessions that are running. The default value is 0.

**Queue polling process timeout**
specifies the time (in seconds) to wait for the server sessions to end when the object spawner is shutting down. If you specify a value that is greater than zero, and any server sessions are still running after the time has
elapsed, then the spawner terminates the sessions. If you specify 0, then there is no time limit for the server sessions to end. The default value is 0.

It is recommended that you use the default value of 0. Make sure that the code that is run by your server sessions checks for stop messages. See “Checking for Stop Messages” on page 269.

Note: If you specify a value that is greater than zero, then the spawner log might contain an error message, "Failed to locate the server indicated in the kill request" for each server session that ended normally. These messages do not indicate a problem.

Note: If you specify a value that is greater than zero, then the spawner always waits the full time-out period when shutting down. For example, if you specify 30 seconds as the time-out value, then the spawner always waits 30 seconds to shut down, even if all of the server sessions end before 30 seconds. If a spawner manages multiple polling servers, then the polling servers are shut down sequentially. The time delay for shutting down the spawner is cumulative.

6 For WebSphere MQ polling servers, select the WebSphere Options tab and specify whether the MQ Server interface is used for monitoring the queue depth. If you choose to use the MQ Server interface, then the object spawner and the queue manager must be on the same machine.

Note: If you do not choose the Server interface, then the connection to the remote queue manager must be defined on the object spawner machine. For more information, see “Define the Queue Manager Connection on the Client Machine” on page 8.

7 Click OK to return to the New Server Wizard, and then click Next.

8 Review your server settings, and then click Finish to create the server definition.

Add the Polling Server to the Object Spawner Definition

To assign a polling server to the object spawner definition, perform the following steps:

1 In SAS Management Console, expand the Server Manager, and then locate the object spawner that you want to modify.

2 Select File ⇒ Properties to open the Properties dialog box for the spawner.

3 On the Servers tab, move the polling server from the Available servers pane to the Selected servers pane.

4 Click OK to save your changes and return to the Server Manager.
If the spawner is running, then refresh the spawner metadata by performing the following steps:

- Expand the spawner definition and select the host name.
- Select **Actions ➔ Refresh Spawner** to refresh the spawner metadata.

## Configure Your Java Environment for the Object Spawner and the JMS Access Method

### Overview of Configuring Your Java Environment

For polling servers that interact with JMS queue managers, you must configure the Java environment for the object spawner and for the JMS access method. Perform the following tasks to configure your Java environment:

1. **“Add the JAR Files for Your JMS Provider to Your CLASSPATH= Environment Variable”**
2. **“Specify the JRE Options for the Object Spawner (Second Maintenance Release of SAS 9.3 and Later)”**
3. **“Set RACF Program Control for the JPROXY Load Module (<fo:inline keep-together.within-line="always">z/OS</fo:inline> Only)”**

### Add the JAR Files for Your JMS Provider to Your CLASSPATH= Environment Variable

The JAR files for your JMS provider must be included in your CLASSPATH= environment variable.

For more information about this step, see the documentation for your JMS provider.

For z/OS, specify your classpath in the `tkmvsev_usermods.cfg` file in the `/Levnn/ObjectSpawner` subdirectory of your SAS configuration directory. Add the following command:

```
SST TKJNI_OPT_CLASSPATH=JAR-files-list
```
Specify the JRE Options for the Object Spawner (Second Maintenance Release of SAS 9.3 and Later)

The object spawner that launches your polling server must specify JRE options as part of its start-up command. To specify the JRE options:

1. Edit the object spawner user modifications file, ObjectSpawner_usermods.bat (or ObjectSpawner_usermods.sh on UNIX or z/OS).

   The object spawner user modifications file should be located in the /Levn/ObjectSpawner subdirectory of your SAS configuration directory.

2. Uncomment the line of the script that defines the JREOPTIONS parameter.

3. If your JMS provider requires native libraries, add the following property to the JREOPTIONS in your ObjectSpawner_usermods file:

   -Djava.library.path=path-to-libraries

   The value for the -Djava.library.path= parameter should specify the path where your native libraries are located.

   For example, if you use the WebSphere MQ JMS provider and the connection factory uses the bindings transport, then the provider will need to load the mqjbnd library. On Windows, your library path parameter might be similar to the following:

   -Djava.library.path=c:\program files\ibm\websphere mq\java\lib

4. For z/OS, if your object spawner runs as a started task, copy the -jreoptions parameter from your ObjectSpawner_usermods.sh file into the objectspawner.parm file.

   objectspawner.parm is located in the /Levn/ObjectSpawner subdirectory of your SAS configuration directory.

   Also, change the region size in the JCL of your started task to 1024M.

5. For Windows, update the object spawner service by using the following command:

   ObjectSpawner.bat -reinstall

   For UNIX and z/OS, restart the object spawner. For details about restarting the object spawner, see the SAS Intelligence Platform: Application Server Administration Guide.

Specify the JRE Options for the Object Spawner (Before the Second Maintenance Release of SAS 9.3 )

The object spawner that launches your polling server must specify JRE options as part of its start-up command. To specify the JRE options:

1. Obtain the JRE options from SAS by using the following code in a SAS session:
filename f 'ObjectSpawner_usermods.bat' lrecl=32767;
data _null_;  
length usermods $32767;
file f;
usermods = "-jreoptions " || '"' || getoption('jreoptions') || '"';
if ( "&sysscp" = 'WIN' ) then
  usermods = 'Set USERMODS=' || usermods;
else
  usermods = "USERMODS='" || trim(usermods) || '"';
put usermods;
run;

2 Copy the ObjectSpawner_usermods.bat file to the directory where your object spawner start-up script is located.

The object spawner start-up script should be located in the /Lev
ObjectSpawner subdirectory of your SAS configuration directory.

3 For UNIX and z/OS, rename the file to ObjectSpawner_usermods.sh.

4 If your JMS provider requires native libraries, add the following property to the JREOPTIONS in your ObjectSpawner_usermods file:

   -Djava.library.path=path-to-libraries

   The value for the -Djava.library.path= parameter should specify the path where your native libraries are located.

   For example, if you use the WebSphere MQ JMS provider and the connection factory uses the bindings transport, then the provider will need to load the mqjbnd library. On Windows, your library path parameter might be similar to the following:

   -Djava.library.path=c:\program files\ibm\websphere mq\java\lib

5 Edit your object spawner start-up script:

Note: For z/OS, if your spawner runs as a started task, then skip to step 6.

a After the call to the level_env script, add a call to the ObjectSpawner_usermods file.

   For example, in Windows add the command

   call ObjectSpawner_usermods.bat

   In UNIX or z/OS, add the command

   . ObjectSpawner_usermods.sh

b In the declaration for the CMD_OPTIONS variable, add a reference to the USERMODS environment variable.

   For example, in Windows add "%USERMODS%" to the set CMD_OPTIONS= command.

   In UNIX or z/OS, add "${USERMODS}" to the CMD_OPTIONS= command.

c For UNIX and z/OS, add eval to the start of the spawner invocation command, escape all of the quotation mark characters in the command, and then enclose the entire command after eval in double quotation marks.

   For example:
eval "nohup $COMMAND $CMD_OPTIONS -sasSpawnerCh "$SPWNNAME" < ... >"

Note: If you apply a SAS maintenance release, then you might need to repeat the changes to your object spawner start-up script.

For z/OS, if your object spawner runs as a started task, copy the -jreoptions parameter from your objectSpawner_usermods.sh file into the objectspawner.parm file.

objectspawner.parm is located in the /Lev/ObjectSpawner subdirectory of your SAS configuration directory.

---

Set RACF Program Control for the JPROXY Load Module (z/OS Only)

On z/OS, if the spawner uses an identity other than its own to start a server, then all of modules that are loaded into the address space must be RACF program controlled, including JPROXY. The location of the JPROXY load module is specified by the TKJNI_OPT_PROXYPATH record in your TKMVSENV data set.

For details about the TKMVSENV data set, see SAS Companion for z/OS. For details about setting RACF program control for a load module, see the z/OS Security Server RACF Security Administrator’s Guide at publibz.boulder.ibm.com/epubs/pdf/ichza780.pdf.

---

Configure Your Programs for Message Queue Polling

Environment Variables for the Polling Server

Environment Variables That Are Set Automatically

When a polling server session is started, the object spawner automatically creates the following environment variables for the session:

- **SASQSID**
  - specifies the unique identifier for the object spawner.

- **SASQUEUE**
  - specifies the name of the queue for the polling server.

- **SASQMQGR**
  - specifies the name of the queue manager that the polling server uses to access the queue.
Specifying Environment Variables on the SAS Command

For WebSphere MQ, you can also set environment variables on the server by using the –SET option on the SAS command for the server session. For example, you might want to specify the queue model by using the MQMODEL variable. The following SAS command sets the queue model to client and sets the MQSERVER variable to enable remote access:

```
sas -sysin "myfile.sas" -set MQMODEL client
     -set MQSERVER "CHANNEL1/TCP/192.168.0.10(1414)"
```

Retrieving Environment Variable Values

You can retrieve the values of the environment variables by using the SYSGET( ) function. For example, the following code retrieves the SASQMGR value and stores it in the QMGR variable:

```
qmgr= sysget('SASQMGR');
```

You can also use the %SYSGET macro function. Here is an example:

```
%let qmgr=%sysget(SASQMGR);
```

On z/OS, if you use a UNIX shell script to invoke SAS, then you must use the -SET invocation option to retrieve the environment variables within the script and pass them to the SAS session. For example:

```
-set "SASQSID ${SASQSID}" -set "SASQMGR ${SASQMGR}"
```

Checking for Stop Messages

Overview of Checking for Stop Messages

The message queue polling server uses SAS sessions to perform processing. These sessions are managed by the object spawner. When the object spawner is stopped, it puts high-priority stop messages on the message queue for each server session that it started. Each stop message contains a unique identifier string that identifies the spawner. The DATA step program must check for this, perform cleanup, and close immediately upon receiving the stop message.

The SASQSID value is passed to the polling server by the object spawner as an environment variable. For more information, see “Environment Variables for the Polling Server” on page 268.

Checking for Stop Messages with the JMS Access Method

To check for stop messages when you use the JMS file access method:

1. Get the value of the SASQSID variable by using the following code:

   ```
   %let sid_var= %sysget(SASQSID);
   ```

2. In the FILENAME or INFILE statement that uses the JMS access method, specify the &SID_VAR macro variable as the value for EOFCHECK=.  

Checking for Stop Messages with WebSphere MQ

If you set the SASQSID value in the get message options (on the CALL MQGMO routine), then the MQGET call will check for a stop message. If the message is found, and the SASQSID value matches the identifier in the message, then MQGET returns a completion code of 2 and a reason code of -2.

For queues that are monitored by the message queue polling server, the MsgDeliverySequence property must be set to Priority.

The following code fragment shows an MQGET call that checks for a stop message:

```plaintext
call mqget(hconn, hobj, hmd, hgmo, msglen, cc, reason);
if cc ^= 0 then do;
   if reason = 2033 then do;
      put 'No message available';
   end;
else do;
   if reason = -2 then do;
      put "MQGET: received stop message from object spawner";
      goto exit;
   end;
   else put 'MQGET: failed with reason= ' reason;
   end;
else put 'MQGET: message received: ';
```

Message Queue Polling Example for WebSphere MQ

The following code is a sample application that you can run with message queue polling.

```plaintext
data _null_;
length hconn hobj cc reason 8;
length rc hod hgmo hmd hmap msglen 8;
length parms $ 200 options $ 200 action $ 3 msg $ 200;
length desc $ 50;

msglen=0;
hconn=0;
hobj=0;
hod=0;
hgmo=0;
hmd=0;
hmap=0;

/* Get the variables set by the object spawner for this session */
sid = sysget('SASQSID');
qmgr= sysget('SASQMGR');
qname= sysget('SASQUEUE');
```
put "Spawner job started."
put "sid = " sid
put "qmgr = " qmgr
put "qname = " qname;

call mqconn(qmgr, hconn, cc, reason);

action = "GEN"
parms="OBJECTNAME"
objname=qname;
call mqod(hod, action, rc, parms, objname);
if rc ^= 0 then do;
    put 'MQOD: failed with rc= ' rc;
    msg = sysmsg();
    put msg;
    goto exit;
end;

options="INPUT_SHARED"
call mqopen(hconn, hod, options, hobj, cc, reason);
if cc ^= 0 then do;
    put 'MQOPEN: failed with reason= ' reason;
    goto exit;
end;

parms= "SASQSID"
call mqgmo(hgmo, action, rc, parms, sid);
if rc ^= 0 then do;
    put 'MQGMO: failed with rc= ' rc;
    msg = sysmsg();
    put msg;
    goto exit;
end;

desc="CHAR,,100"
call mqmap(hmap, rc, desc);
if rc ^= 0 then do;
    put 'MQMAP: failed with rc= ' rc;
    msg = sysmsg();
    put msg;
    goto exit;
end;

reason = 0;
do until (reason = 2033);

    action = "GEN"
call mqmd(hmd, action, rc);
if rc ^= 0 then do;
    put 'MQMD: failed with rc= ' rc;
    msg = sysmsg();
    put msg;
end;

call mqget(hconn, hobj, hmd, hgmo, msglen, cc, reason);
if cc ^= 0 then do;
   if reason = 2033 then do;
      put 'No message available';
   end;
   else do;
      if reason = -2 then do;
         /* -2 indicates that a session-specific stop message has */
         /* been received from the object spawner queue monitor */
         /* application. We should clean up and shutdown */
         /* immediately. */
         put "MQGET: received stop message from object spawner";
         goto exit;
      end;
      else put 'MQGET: failed with reason= ' reason;
   end;
else put 'MQGET: message received: ';

/* Do any application-specific processing of messages here. */

if hmd ^= 0 then do;
   call mqfree(hmd);
end;

exit: /* end do loop */

if hobj ^= 0 then do;
   options="NONE";
   call mqclose(hconn, hobj, options, cc, reason);
   if cc ^= 0 then do;
      put 'MQCLOSE: failed with reason= ' reason;
   end;
end;

if hconn ^= 0 then do;
   call mqdisc(hconn, cc, reason);
   if cc ^= 0 then do;
      put 'MQDISC: failed with reason= ' reason;
   end;
end;

if hod ^= 0 then do;
   call mqfree(hod);
end;
if hgmo ^= 0 then do;
   call mqfree(hgmo);
end;
if hmd ^= 0 then do;
   call mqfree(hmd);
end;
if hmap ^= 0 then do;
   call mqfree(hmap);
end;

run;
FILENAME Statement: JMS Access Method

Assigns a fileref that enables you to access a JMS destination by using the JMS access method to send and receive messages of the type TextMessage.

Valid in: Anywhere
Category: Data Access
Requirement: The JAR files for your JMS provider must be included in your CLASSPATH=
environment variable.
Tip: For the ActiveMQ JMS provider, you must either specify the TIMEOUT= option or configure your jms.prefetchPolicy.queuePrefetch property to 0.

Syntax

FILENAME fileref JMS <"destination-name" > <access-method-options>

Summary of Optional Arguments

"destination-name"
  specifies the Java Naming and Directory Interface (JNDI) name for the JMS destination.

AUTHDOMAIN= "domain"
  specifies a metadata authentication domain that is used to retrieve user credentials.

CONNFACCTOR= "JNDI-connection-name"
  specifies the JNDI name of the JMS connection factory.

CONNOPTIONS= "options-list"
specifies a list of options that are added to the environment that is used to create the initial JNDI context.

**CORRELATIONID=correlId-variable**

specifies the correlation ID for outgoing messages, or stores the correlation ID for incoming messages to a variable.

**DELIVERYMODE=PERSISTENT | NONPERSISTENT**

specifies the JMS delivery mode.

**DESTINATION= "destination-name"**

specifies the JNDI name for the JMS destination.

**EOFCHECK= "check-string"**

specifies a string that indicates the end of the file.

**JNDIICXTFACTORY="factory-class-name"**

specifies the fully qualified class name of the JNDI provider initial context factory.

**JNDIPROVIDERURL="JNDI-provider-URL"**

specifies server information that is used to create the JNDI initial context.

**LRECL=n | nk**

specifies the record length in bytes.

**MATCHOPTIONS= CORRELATIONID | MESSAGEID | BOTHIDS**

adds the correlation ID, the message ID, or both to your message selector string.

**MESSAGEID= variable**

assigns message ID values to the DATA step variable that you specify.

**MSGSELECTOR= "filter-string"**

specifies a filter string that restricts the messages that are received by the access method.

**PASSWORD= "password" | _PROMPT_**

specifies the password that is used to connect to the JMS provider.

**PRIORITY=n**

specifies the message priority for outgoing messages, where a larger priority number indicates a higher priority.

**TIMEOUT=n**

specifies a time limit in milliseconds to wait for a message to arrive.

**TIMETOLIVE=n**

specifies the default length of time in milliseconds that a message is retained by the message system.

**USER= "user-ID" | _PROMPT_**

specifies the user ID that is used to connect to the JMS provider.

**Required Arguments**

**fileref**

is a valid file reference.

**JMS**

specifies the access method that enables you to access Java Messaging Service (JMS) destinations.

**Optional Argument**

**"destination-name"**

specifies the Java Naming and Directory Interface (JNDI) name for the JMS destination.
You can specify a destination by using either the destination-name argument, the DESTINATION= option, or aggregate syntax.

Access Method Options

**AUTHDOMAIN= "domain"**

specifies a metadata authentication domain that is used to retrieve user credentials. The credentials that are associated with the authentication domain are used to connect to the ActiveMQ broker.

**Requirement**

To use the AUTHDOMAIN= option, your SAS session must be able to connect to the SAS Metadata Server. For information about specifying a metadata server connection, see SAS Language Interfaces to Metadata.

**Tip**

The value does not require quotation marks if it is a valid SAS name.

**CONNFACTORY= "JNDI-connection-name"**

specifies the JNDI name of the JMS connection factory. The connection factory contains a set of configuration parameters that are used to connect to the JMS provider.

**Alias**

CONNFACT=

**Tip**

The value does not require quotation marks if it is a valid SAS name.

**CONNOPTIONS= "options-list"**

specifies a list of options that are added to the environment that is used to create the initial JNDI context. Specify the options as space-delimited name-value pairs. For example,

CONNOPTIONS="option1=value option2=value"

**Aliases**

CONNECTIONOPTS=

CONNOPTS=

**CORRELATIONID=correlId-variable**

specifies the correlation ID for outgoing messages, or stores the correlation ID for incoming messages to a variable. For the FILE statement, specifies a DATA step variable whose value is used as the correlation ID of the message.

For the INFILE statement:

- the correlation ID of the incoming message is stored in the variable that you specify.

- the value of the variable that you specify is used as the correlation ID for message selector when you specify MATCHOPTS=CORRELID or MATCHOPTS=BOTHIDS.

**Note:** The CORRELATIONID= option is valid in the FILE and INFILE statements only.

**Alias**

CORRELID=
DELIVERYMODE=PERSISTENT | NONPERSISTENT
specifies the JMS delivery mode. Specify one of the following:

PERSISTENT
specifies that messages are stored in a stable storage location. Messages can be recovered if the JMS provider fails to deliver the message.

NONPERSISTENT
specifies that messages are not stored in a stable storage location. The NONPERSISTENT mode offers better performance than the PERSISTENT mode, but messages might be lost if the JMS provider fails.

Default  PERSISTENT

DESTINATION= "destination-name"
specifies the JNDI name for the JMS destination.

Alias  DEST=

Interaction You can specify a destination by using either the destination-name argument, the DESTINATION= option, or aggregate syntax.

Tip The value does not require quotation marks if it is a valid SAS name.

EOFCHECK= "check-string"
specifies a string that indicates the end of the file. The JMS access method returns end of file when it encounters a message that matches the EOFCHECK= value or the EOFCHECK= value prefixed with the string "OBJMQ_STOP:"

For example, if you specify EOFCHECK="STOPMSG", then a message value of either STOPMSG or OBJMQ_STOP:STOPMSG causes the JMS access method to return end of file.

JNDICTXFACTORY="factory-class-name"
specifies the fully qualified class name of the JNDI provider initial context factory. The class that you specify is used to create the JNDI initial context and provides the starting point for the resolution of names.

Alias  JNDICTXTFAC=

JNDIPROVIDERURL="JNDI-provider-URL"
specifies server information that is used to create the JNDI initial context. The server information that you specify has the basic format of a URL string. However, the specific formatting of the URL string depends on the class that you use to create the JNDI initial context. See the class documentation for your initial context class for details about how to specify the provider URL.

Alias  JNDIPROVURL=

LRECL=n | nk
specifies the record length in bytes. Add k to the number to specify a multiple of 1024.

Default  32767

MATCHOPTIONS= CORRELATIONID | MESSAGEID | BOTHIDS
adds the correlation ID, the message ID, or both to your message selector string. Specify one of the following values:
FILENAMe Statement: JMS Access Method

CORRELATIONID
adds the correlation ID to the selector string.

Alias CORRELID

MESSAGEID
adds the message ID to the selector string.

Alias MSGID

BOTHIDS
adds both the correlation ID and the message ID to the selector string.

Alias MATCHOPTS=

MESSAGEID= variable
assigns message ID values to the DATA step variable that you specify. You can use the MESSAGEID= option to obtain the message ID of an incoming or outgoing message. For outgoing messages, each Write operation writes a new message ID value to the variable.

Note: The MESSAGEID= option is valid in the FILE and INFILE statements only.

Alias MSGID=

MSGSELECTOR= "filter-string"
specifies a filter string that restricts the messages that are received by the access method. The filter string uses SQL92 conditional expression syntax. A message selector matches a message if the selector expression evaluates to true for that message.

For example, if you specify msgselector="JMSPriority = 9", then the access method only receives messages where the priority is 9.

PASSWORD= "password" | _PROMPT_
specifies the password that is used to connect to the JMS provider. Specify a password or specify _PROMPT_ to be prompted to enter your password interactively.

Note: You can submit a password as clear text or as an encoded string from the PWENCODE procedure. For more information, see Encryption in SAS.

Alias PWD=

Tips You can specify the AUTHDOMAIN= option instead of the USER= and PASSWORD= options.

The value does not require quotation marks if it is a valid SAS name.

PRIORITY=n
specifies the message priority for outgoing messages, where a larger priority number indicates a higher priority. The value must be between 0 and 9.
PRIORITY= has no effect for incoming messages. You can filter incoming messages based on priority by using the value of JMSPriority in a message selector. For example, the following message selector specifies a priority of 5.

```
msgselector="JMSPriority=5"
```

Default 4

**TIMEOUT=**

specifies a time limit in milliseconds to wait for a message to arrive. If the time expires before a message arrives, then the access method returns end of file. A value of –1 indicates that the access method returns end of file immediately if there are no messages. A value of 0 indicates that the access method will wait an unlimited amount of time for a message to arrive.

Default -1

**TIMETOLIVE=**

specifies the default length of time in milliseconds that a message is retained by the message system.

Default 0 (no limit)

**USER=** "user-ID" | _PROMPT_

specifies the user ID that is used to connect to the JMS provider. Specify a user ID or specify _PROMPT_ to be prompted to enter your user ID interactively.

Alias USERID=

Tips You can specify the AUTHDOMAIN= option instead of the USER= and PASSWORD= options.

The value does not require quotation marks if it is a valid SAS name.

Details

The SAS JMS file access method allows SAS programs to read and write records to and from any message-oriented middleware that supports a JMS provider. The SAS JMS file access method starts a Java Virtual Machine (JVM) if the SAS session has not already done so. The access method uses the Java Native Interface (JNI) to interact with the JMS provider in the JVM.

You can use the JMS file access method to send and receive JMS messages of the type TextMessage. By using SAS formats and informats, you can send and receive character data and numeric data through the JMS access method as long as the data can be represented as text.

Filerefs assigned with the JMS file access method can be used in the same ways as other external file access methods can be used, including with the following statements: FILENAME, FILE, INFILE, and %INCLUDE. The CORRELATIONID= and MESSAGEID= options are valid only in the FILE and INFILE statements. If you specify a fileref on both the FILENAME statement and the FILE or INFILE
statement, then the option values from the FILE or INFILE statement will override
the values from the FILENAME statement.

If your JMS provider requires native libraries, then use the JREOPTIONS= system
option to add the native library path to your -Djava.library.path property.

Note: The JMS file access method uses the MessageConsumer interface to
receive messages from the JMS destination. This means that the messages that
you receive with the JMS access method are removed from the JMS destination.

Note: In order to use the JMS access method, you must specify the JAR files for
your JMS provider in your CLASSPATH= environment variable. For more
information, see the documentation for your JMS provider. On z/OS, add the JAR
files to the TKJNI_OPT_CLASSPATH record in the TKMVSENV data set. For more
information, see the SAS Companion for z/OS.

For more information about JMS, see the following resources:
- the JMS Tutorial at download.oracle.com/javaee/1.3/jms/tutorial/index.html
- the Java Message Service specification at www.oracle.com/technetwork/
  java/jms/index.html
- the JMS API at download.oracle.com/javaee/6/api/javax/jms/package-
  summary.html

Examples:

Example 1: Basic Send and Receive Using WebSphere MQ

This example shows a simple send and receive using WebSphere MQ as the JMS
provider. The first DATA step puts a message on a queue, and the second DATA
step receives the message and writes it to the SAS log. This example assumes that
the JMS administrator created administrative objects for a connection factory named
myqmgr and a destination named myq, and that the JNDI name space is located at \mynode\public\jndi.

filename myref jms 'myq'
  jndiictxtfactory="com.sun.jndi.fscontext.RefFSContextFactory"
  jndiprovdecor="file:///mynode/public/jndi"
  connfac=myqmgr;

data _null_;
  file myref;
  put 'This is a test message';
run;

data _null_;
  infile myref;
  input mymsg $22.;
  put mymsg=;
run;
Example 2: Using ActiveMQ

This example shows a simple send and receive using ActiveMQ as the JMS provider. In this example, the FILE and INFILE assign filerefs rather than using a FILENAME statement. Note that you can install Active MQ by using "Example 7: Installing and Testing ActiveMQ from a SAS Program" on page 281.

No administrative tasks are required for this example. The CONNOPTS= option specifies a name-value pair that is used by the ActiveMQ provider to create the administrative object for the myq destination. The TIMEOUT= option causes the underlying receive call to wait up to 100 milliseconds for a message to arrive.

data _null_;  
  file 'myq' jms  
    jndiiCtxtFactory='org.apache.activemq.jndi.ActiveMQInitialContextFactory'  
    jndiProviderURL='tcp://localhost:61616'  
    connfactory='ConnectionFactory'  
    connopts='queue.myq=myq';  
  put 'This is a test message';  
run;

data _null_;  
  infile 'myq' jms timeout=100  
    jndiiCtxtFactory='org.apache.activemq.jndi.ActiveMQInitialContextFactory'  
    jndiProviderURL='tcp://localhost:61616'  
    connfactory='ConnectionFactory'  
    connopts='queue.myq=myq';  
  input mymsg $22.;  
  put mymsg=;  
run;

Example 3: Using WebSphere Application Server

This example fragment assigns a fileref using WebSphere Application Server as the JMS provider.

    filename myref jms 'MyJMSClientQueue'  
      jndiCtxtFactory='com.ibm.websphere.naming.WsnInitialContextFactory'  
      jndiProviderURL='iiop://mynode:9816'  
      connfactory='MyJMSQueueConnectionFactory';

Example 4: Using JBoss Application Server

This example fragment assigns a fileref using JBoss Application Server as the JMS provider.

    filename myref jms 'myorg/jms/myq'  
      jndiiCtxtFactory='org.jnp.interfaces.NamingContextFactory'  
      jndiproducerurl='jnp://mynode:1099'  
      connfactory='/ConnectionFactory';

Example 5: Using WebLogic Application Server

This example fragment assigns a fileref using WebLogic Application Server as the JMS provider.

    filename myref jms 'myorg/jms/myq'  
      jndiiCtxtFactory='weblogic.jndi.WLInitialContextFactory'
Example 6: Using TIBCO EMS Server

This example fragment assigns a fileref using TIBCO EMS server as the JMS provider.

```sql
jndiProviderURL='t3://mynode:7001'
connfactory='weblogic.jms.ConnectionFactory';
```

Example 7: Installing and Testing ActiveMQ from a SAS Program

This example downloads and installs ActiveMQ and then performs a simple send and receive test using the local ActiveMQ installation.

Prerequisite

In order to use ActiveMQ, you must add the JAR files that are required by your JMS provider to your Java CLASSPATH. You can use the `--SET CLASSPATH` option on your SAS start-up command to add the JAR files.

For this example, your CLASSPATH should include `install-Dir/apache-activemq-5.8.0/activemq-all-5.8.0.jar`, where `install-Dir` is the value of the `installDir=` macro variable in the example code.

For example, using the default `installDir=` value on Windows, add the following to your SAS command:

```sql
-set classpath "c:\apache-activemq-5.8.0\activemq-all-5.8.0.jar"
```

Alternative Download Method

You can also download ActiveMQ manually from activemq.apache.org.

Specify installation parameters. The `tempZipFile=` macro variable specifies the filename for the downloaded installation file. The `installDir=` macro variable specifies the directory where ActiveMQ will be installed (ActiveMQ will be installed in a subdirectory named `apache-activemq-5.6.0`).

```sql
%let tempZipFile=c:\activemq.zip;
%let installDir=c:\;
```

Download the ActiveMQ binary distribution from the web. If necessary, specify the proxy for your internet connection by using the `PROXY=` option in the `FILENAME` statement. If you encounter a connection error, obtain the current recommended URL from ActiveMQ download site:

```sql
http://www.apache.org/dyn/closer.cgi?path=/activemq/apache-activemq/5.8.0/
```

```sql
filename fromweb URL
'http://apache.mirrors.tds.net/activemq/apache-activemq/5.8.0/apache-activemq-5.8.0-bin.zip'
lrecl=8192 recfm=s;
```

```sql
data _null_;
  infile fromweb;
  file "&tempZipFile" recfm=n lrecl=8192;
  input;
  put _infile_;
Extract the ActiveMQ ZIP file into the destination directory.

```groovy
proc groovy;
    add sasjar="ANT";
    submit "&tempZipFile" "&installDir";
    def ant = new AntBuilder();
    ant.unzip( src:args[0], dest:args[1], overwrite:"false" )
    ant.delete( file:args[0] )
endsubmit;
quit;
```

Assign a fileref associated with a third-party message queue. The `JNDIPROVIDERURL=` option value specifies a connection to the embedded ActiveMQ broker through the VM protocol.

```sas
filename myq jms 'dynamicQueues/myqueue'
    jndiiCtxtFactory='org.apache.activemq.jndi.ActiveMQInitialContextFactory'
    jndiProviderURL='vm://broker'
    connfactory='ConnectionFactory'
    timeout=100;
```

Put some test messages on the queue.

```sas
data _null_;
    file myq user=admin pass=admin;
    format a datetime21.1;
    a = datetime(); put a ' Testing. 1 2';
    a = datetime(); put a +1 'This is a test. 3 4';
    a = datetime();
    put a @22 'This is only a test. 5 6';
run;
```

Get the messages back off of the queue and print them in the log.

```sas
data _null_;
    infile myq length=len;
    input a datetime21.2 @;
    len = len - 25;
    input b $varying60. len c d;
    put a= datetime19. b= c= d=;
run;
```

Example 8: Using the XML LIBNAME Engine

This example demonstrates using the XML LIBNAME engine to write and read data from a queue in XML format.

Details

This example uses an XML map file to store metadata for the data set. You can create an XML map file by using the SAS XML Mapper. For more information about XML map files and the XML LIBNAME engine, see the SAS XMLV2 and XML LIBNAME Engines: User's Guide.

Create a sample data set.

```sas
data pointofsale;
    infile cards;
    input saleId date e8601dt22.2 amount customerId storeId $4.;
```
Assign a fileref to a dynamic queue named "myqueue". This example code uses the same ActiveMQ provider as "Example 7: Installing and Testing ActiveMQ from a SAS Program", but you can use any JMS provider.

```sas
filename mymq JMS 'dynamicQueues/myqueue'
jndiCtxtFactory='org.apache.activemq.jndi.ActiveMQInitialContextFactory'
jndiProviderURL='vm://broker'
connFactory='ConnectionFactory'
timeout=100;
```

Assign a libref that uses the XML engine and JMS file access method.

```sas
libname mymq xmlv2;
```

Put the data set on the queue in XML format.

```sas
data mymq.pointofsale;
  set work.pointofsale;
run;
```

Create an XML map for the data set. The XML map for this example is created through a DATA step for demonstration purposes. For your own data sets, use the SAS XML Mapper to create XML map files.

```sas
filename myXMLmap temp;
data _null_;  
  file myXMLmap;
  infile cards;
  input;
  put _infile_;
  cards;

<?xml version="1.0" encoding="UTF-8"?>
<SXLEMAP name="AUTO_GEN" version="2.1">
  <TABLE description="POINTOFSALE" name="POINTOFSALE">
    <TABLE-PATH syntax="XPath">/TABLE/POINTOFSALE</TABLE-PATH>
    <COLUMN name="saleId">
      <PATH syntax="XPath">/TABLE/POINTOFSALE/saleId</PATH>
      <TYPE>numeric</TYPE>
      <DATATYPE>integer</DATATYPE>
    </COLUMN>
    <COLUMN name="date">
      <PATH syntax="XPath">/TABLE/POINTOFSALE/date</PATH>
      <TYPE>numeric</TYPE>
      <DATATYPE>integer</DATATYPE>
    </COLUMN>
  </TABLE>
</SXLEMAP>
```
Assign a libref that uses the XML engine, the XML map file, and the JMS access method.

```
libname mymq xmlv2 xmlmap=myXMLmap;
```

Read XML off the queue and create a data set.

```
data myds;
  set mymq.PointOfSale;
  put _all_;
run;
```

Print the data set.

```
proc print data=myds;
  format date e8601dt22.2 amount dollar6.2;
run;
```

Example 9: Using Request and Response Queues

This example uses separate request and response queues to send messages and then retrieve them by using their message ID.

Details

The MESSAGEID= and CORRELID= options are used to store the message ID values. The MATCHOPTS= option is used to generate a message selector that retrieves the messages based on their ID values.

Although this example runs in a single process, a common request and response scenario uses many asynchronous processes to create requests.

Assign a fileref for the message queue.

```
filename x jms
  jndiiCtxtFactory='com.sun.jndi.fscontext.RefFSContextFactory'
```
Put some data onto a request queue. The MESSAGEID= option in the FILE statement stores message IDs for each message in the mID variable.

data requests;
  length name $40;
  file x(myRequest) messageid=mID deliverymode=nonpersistent;
  loanAmount=1000; creditScore=675; name='Brown, Bob';
  put loanAmount @10 creditScore @15 name;
  output;
  loanAmount=2000; creditScore=500; name='Wilson, Jane';
  put loanAmount @10 creditScore @15 name;
  output;
  loanAmount=3000; creditScore=750; name='White, William';
  put loanAmount @10 creditScore @15 name;
  output;
  loanAmount=5000; creditScore=700; name='Jones, Fred';
  put loanAmount @10 creditScore @15 name;
  output;
run;

Retrieve data from the request queue, process the data, and send it to the response queue. The MESSAGEID= option in the INFILE statement reads the message IDs from the request queue into the mID variable. The CORRELID= option in the FILE statement stores the request queue message IDs as correlation IDs for the response queue.

data _null_;  
  length name $40 mID $51;
  infile x(myRequest) messageid=mID length=len;
  file x(myResponse) correlid=mID deliverymode=nonpersistent;
  input loanAmount creditScore @;
  len = len - 14;
  input +1 name $varying40. len;
  if ( creditScore > 600 and loanAmount < 10000 ) then
    decision = 'approved';
  else decision = 'declined';
  put decision name;
run;

Sort the data set to demonstrate that the messages are read according to the message ID and not simply in sequence.

proc sort data=requests; by name; run;

Read the messages from the response queue and match the correlation IDs to the message IDs. The MATCHOPTS= option in the INFILE statement specifies that the correlation ID is used to select the messages from the queue. The CORRELID= option in the INFILE statement specifies that cID variable contains the correlation ID values.

data responses;
  length msgName $40 cID $51;
  drop msgName mID cID len;
  set requests;
  cID=mID;
  infile x(myResponse) matchopts=correlid correlid=cID length=len;
input @1 answer $8. @@;
len = len - 9;

**Compare the customer names to verify that the data from the request queue and the response queue are a match.**

input +1 msgName $varying40. len;
if ( name ^= msgName ) then put 'Error: match failed.';
run;

**Print the data set.**

proc print data=responses;
run;
The ACTIVEMQ File Access Method

Dictionary

FILENAME Statement: ACTIVEMQ Access Method

Assigns a fileref that enables you to access an ActiveMQ broker by using the ACTIVEMQ access method.

Valid in: Anywhere
Category: Data Access
Requirement: The ActiveMQ broker that you access must be release 5.7.0 or later.

Syntax
FILENAME fileref ACTIVEMQ "destination-name" > access-method-options>

Summary of Optional Arguments

"destination-name" specifies the destination name for the ActiveMQ message broker.

AUTHDOMAIN= "domain"
specifies a metadata authentication domain that is used to retrieve user credentials.

CLIENTID= "client-name" | GENERATE | NONE
specifies an ID string that is used to identify the client.

CONTENTTYPE= "type"
specifies the format of messages or records that are being written.

CORRELATIONID=correlId-variable
specifies the correlation ID for outgoing messages, or stores the correlation ID for incoming messages to a variable.

DELIVERYMODE=PERSISTENT | NONPERSISTENT
specifies the message delivery mode.
DESTINATION= "destination-name"
    specifies the name of the message destination on the ActiveMQ broker.

DESTTYPE= "type"
    specifies whether the destination is a queue or a topic.

ENCODING= "encoding-type"
    transcodes messages or records by using the specified encoding type.

EOFCHECK= "check-string"
    specifies a string that indicates the end of the file.

LRECL=n | nk
    specifies the record length in bytes.

MATCHOPTIONS= CORRELATIONID | MESSAGEID | BOTHIDS
    adds a value to an existing message selector string, or creates a
    message selector string.

MESSAGEID= variable
    assigns message ID values to the DATA step variable that you specify.

MSGBODYPREFIX= "prefix-string"
    specifies a string that is appended to the start of each message that you
    write.

MSGSELECTOR= "filter-string"
    specifies a filter string that restricts the messages that are received by the
    access method.

PASSWORD= "password" | _PROMPT_
    specifies the password that is used to connect to the ActiveMQ broker.

PRIORITY=n
    specifies the message priority for outgoing messages, where a larger
    priority number indicates a higher priority.

PROXYPASSWORD= "password"
    specifies the password that is used to connect to the proxy server.

PROXYURL= "URL-string"
    specifies the URL for a proxy server that is used to connect to the
    ActiveMQ broker.

PROXYUSER= "user-ID"
    specifies the user ID that is used to connect to the proxy server.

TERMSTR= LF | CRLF | CR | NONE | NULL
    specifies the character sequence that indicates the end of a line in
    messages that are being read.

TIMEOUT=n
    specifies a time limit in milliseconds to wait for a message to arrive.

TIMETOLIVE=n
    specifies the default length of time in milliseconds that a message is
    retained by the message system.

URL= "URL-string"
    specifies the URL string that is used to connect to the ActiveMQ broker.

USER= "user-ID" | _PROMPT_
    specifies the user ID that is used to connect to the ActiveMQ broker.

Required Arguments

fileref
    is a valid file reference.
ACTIVEMQ
specifies the access method that enables you to access an ActiveMQ broker.

Optional Argument

"destination-name"
specifies the destination name for the ActiveMQ message broker.

Interaction
You can specify the destination name by using either the destination-name argument, the DESTINATION= option, the URL= option, or aggregate syntax.

Access Method Options

AUTHDOMAIN= "domain"
specifies a metadata authentication domain that is used to retrieve user credentials. The credentials that are associated with the authentication domain are used to connect to the ActiveMQ broker.

Requirement
To use the AUTHDOMAIN= option, your SAS session must be able to connect to the SAS Metadata Server. For information about specifying a metadata server connection, see SAS Language Interfaces to Metadata.

Tip
The value does not require quotation marks if it is a valid SAS name.

CLIENTID= "client-name" | GENERATE | NONE
specifies an ID string that is used to identify the client. The client ID is used by the ActiveMQ broker to clean up resources from a client session. You can specify any string, or you can specify GENERATE to automatically generate a universally unique ID for the client. You can also specify NONE to specify that no client ID is used.

Default
GENERATE

CONTENTTYPE= "type"
specifies the format of messages or records that are being written. You can specify either the text/xml or the application/x-www-form-urlencoded HTTP content types.

Note: The CONTENTTYPE= option is valid in the FILE and FILENAME statements only.

Alias
CTYPE=

Default
text/xml

CORRELATIONID=correlld-variable
specifies the correlation ID for outgoing messages, or stores the correlation ID for incoming messages to a variable. For the FILE statement, specifies a DATA step variable whose value is used as the correlation ID of the message. The correlation ID is added to outgoing messages in the JMSCorrelationID and CorrelationID parameters.

For the INFILE statement:
the correlation ID of the incoming message is stored in the variable that you specify.

- the value of the variable that you specify is used as the correlation ID for the message selector when you specify MATCHOPTS=CORRELID or MATCHOPTS=BOTHIDS.

Note: The CORRELATIONID= option is valid in the FILE and INFILE statements only.

Alias  CORRELID=

**DELYERMODE=** PERSISTENT | NONPERSISTENT

specifies the message delivery mode. Specify one of the following:

- **PERSISTENT**
  specifies that messages are stored in a stable storage location. Messages can be recovered if the ActiveMQ broker fails to deliver the message.

- **NONPERSISTENT**
  specifies that messages are not stored in a stable storage location. The NONPERSISTENT mode offers better performance than the PERSISTENT mode, but messages might be lost if the ActiveMQ broker fails.

Note: The DELIVERYMODE= option is valid in the FILE and FILENAME statements only.

Default  PERSISTENT

**DESTINATION=** "destination-name"

specifies the name of the message destination on the ActiveMQ broker. The destination can also be specified by using the destination-name argument, aggregate syntax, or as a parameter on the URL= option.

Alias  DEST=

Interaction  You can specify a destination by using either the destination-name argument, the DESTINATION= option, or aggregate syntax.

Tip  The value does not require quotation marks if it is a valid SAS name.

**DESTTYPE=** "type"

specifies whether the destination is a queue or a topic. Specify "Q" or "QUEUE" for a queue, or specify "TOPIC" for a topic.

Default  "QUEUE"

Interaction  If the destination is specified on the URL= option, then the DESTTYPE= option is ignored.

**ENCODING=** "encoding-type"

transcodes messages or records by using the specified encoding type. When writing, the encoding that you specify is used to transcode the messages that are written to the message broker. When reading, the encoding that you specify is used to transcode the messages to your SAS session encoding.

Default
When writing, the default encoding is the SAS session encoding. When reading, the default encoding is UTF8.

**EOFCHECK= "check-string"**
specifies a string that indicates the end of the file. The ACTIVEMQ access method returns end of file when it encounters a message that matches the EOFCHECK= value or the EOFCHECK= value prefixed with the string "OBJMQ_STOP:"

For example, if you specify EOFCHECK="STOPMSG", then a message value of either STOPMSG or OBJMQ_STOP:STOPMSG causes the ACTIVEMQ access method to return end of file.

**LRECL=n | nk**
specifies the record length in bytes. Add k to the number to specify a multiple of 1024.

Default 32767

**MATCHOPTIONS= CORRELATIONID | MESSAGEID | BOTHIDS**
adds a value to an existing message selector string, or creates a message selector string. Specify one of the following values:

**CORRELATIONID**
adds the correlation ID to the selector string.

Alias CORRELID

**MESSAGEID**
adds the message ID to the selector string.

Alias MSGID

**BOTHIDS**
adds both the correlation ID and the message ID to the selector string.

Alias MATCHOPTS=

**MESSAGEID= variable**
assigns message ID values to the DATA step variable that you specify. You can use the MESSAGEID= option to obtain the message ID of an incoming or outgoing message. For outgoing messages, each Write operation writes a new message ID value to the variable.

Note: The MESSAGEID= option is valid in the FILE and INFILE statements only.

Alias MSGID=

**MSGBODYPREFIX= "prefix-string"**
specifies a string that is appended to the start of each message that you write. Adding a prefix is useful when you are writing multiple messages at once, such as if you are using the FCOPY function to write messages.
Note: The MSGBODYPREFIX= option is valid in the FILE and FILENAME statements only.

**MSGSELECTOR= "filter-string"**
specifies a filter string that restricts the messages that are received by the access method. The filter string uses SQL92 conditional expression syntax. A message selector matches a message if the selector expression evaluates to true for that message.

For example, if you specify `msgselector="JMSPriority = 9"`, then the access method only receives messages where the priority is 9.

Note: The MSGSELECTOR= option is valid in the FILENAME and INFILE statements only.

**PASSWORD= "password" | _PROMPT_**
specifies the password that is used to connect to the ActiveMQ broker. Specify a password or specify _PROMPT_ to be prompted to enter your password interactively.

Note: You can submit a password as clear text or as an encoded string from the PWENCODE procedure. For more information, see *Encryption in SAS*.

**Aliases**
- PASS=
- PW=
- PWD=

**Tips**
You can specify the AUTHDOMAIN= option instead of the USER= and PASSWORD= options.

The value does not require quotation marks if it is a valid SAS name.

**PRIORITY=n**
specifies the message priority for outgoing messages, where a larger priority number indicates a higher priority. The value must be between 0 and 9.

PRIORITY= has no effect for incoming messages. You can filter incoming messages based on priority by using the value of JMSPriority in a message selector. For example, the following message selector specifies a priority of 5.

`msgselector="JMSPriority=5"`

The default value is determined by the ActiveMQ broker. If the ActiveMQ broker uses a JDBC message store, then the default value is 4. For a KahaDB message store, the default value is 5.

**PROXYPASSWORD= "password"**
specifies the password that is used to connect to the proxy server. Specify a password or specify _PROMPT_ to be prompted to enter your password interactively.

Note: You can submit a password as clear text or as an encoded string from the PWENCODE procedure. For more information, see *Encryption in SAS*. 
**FILENAME Statement: ACTIVEMQ Access Method**

<table>
<thead>
<tr>
<th>Aliases</th>
<th>PROXYPASS=</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>PROXYPASSWD=</td>
</tr>
</tbody>
</table>

**Interaction**

This option has no effect if you do not specify the PROXYURL= option.

**Tip**

The value does not require quotation marks if it is a valid SAS name.

**PROXYURL= "URL-string"**

specifies the URL for a proxy server that is used to connect to the ActiveMQ broker.

**PROXYUSER= "user-ID"**

specifies the user ID that is used to connect to the proxy server. Specify a user ID or specify _PROMPT_ to be prompted to enter your user ID interactively.

<table>
<thead>
<tr>
<th>Alias</th>
<th>PROXYUSERID=</th>
</tr>
</thead>
</table>

**Interaction**

This option has no effect if you do not specify the PROXYURL= option.

**Tip**

The value does not require quotation marks if it is a valid SAS name.

**TERMSTR= LF | CRLF | CR | NONE | NULL**

specifies the character sequence that indicates the end of a line in messages that are being read. Specify one of the following values:

- **LF**
  
  specifies a line feed character.

- **CRLF**
  
  specifies a carriage return character and a line feed character.

- **CR**
  
  specifies a carriage return character.

- **NONE**
  
  specifies that the entire message is read, including any line-ending characters.

**Note:** The TERMSTR= option is valid in the FILENAME and INFILE statements only.

**Default**

LF

**TIMEOUT=n**

specifies a time limit in milliseconds to wait for a message to arrive. If the time expires before a message arrives, then the access method returns end of file. If you do not specify the TIMEOUT= option, then the access method returns end of file after one millisecond if there are no messages. A value of 0 indicates that the access method waits an unlimited amount of time for a message to arrive.

**Default**

1

**TIMETOLIVE=n**

specifies the default length of time in milliseconds that a message is retained by the message system.
The TIMETOLIVE= option is valid in the FILE and FILENAME statements only.

Default 0 (no limit)

URL= "URL-string"
specifies the URL string that is used to connect to the ActiveMQ broker. The format of the URL string is "scheme://hostname:port/webcontext/serverlet/destination?parameter1&parameter2"

The /destination part of the URL string is optional if you specify the destination by using another option or parameter (for example, the DESTINATION= option.)

If you specify the destination as part of the URL string, then you must also specify the "&TYPE=" parameter as part of your URL string. Specify &type=queue to specify that the destination is a queue, or specify &type=topic to specify that the destination is a topic.

Note: The web context for the ActiveMQ broker URL is specified in the jetty.xml file in the /conf subdirectory of your ActiveMQ installation. The name of your context path is specified by the value of the org.eclipse.jetty.webapp.WebAppContext property. For ActiveMQ 5.7.0 and earlier, the default context is demo. For ActiveMQ 5.8.0 and later, the default context is api.

Default http://localhost:8161/demo/message

Interaction If you specify the destination by using the DESTINATION= option, the destination-name argument, or aggregate file syntax, then the destination value from those parameters overrides the destination that is specified on the URL=.

USER= "user-ID" | _PROMPT_
specifies the user ID that is used to connect to the ActiveMQ broker. Specify a user ID or specify _PROMPT_ to be prompted to enter your user ID interactively.

Alias USERID=

Tips You can specify the AUTHDOMAIN= option instead of the USER= and PASSWORD= options.

The value does not require quotation marks if it is a valid SAS name.

Details

Overview

The ActiveMQ file access method enables SAS programs to read and write records to and from an ActiveMQ broker directly through the HTTP protocol. No message-oriented middleware is required on the machine where SAS is running.

Filerefs assigned with the ActiveMQ file access method can be used in the same ways as other external file access methods can be used, including with the following statements: FILENAME, FILE, INFILE, and %INCLUDE. If you specify a fileref on both the FILENAME statement and the FILE or INFILE statement, then the option
values from the FILE or INFILE statement will override the values from the FILENAME statement.

Communication with the ActiveMQ Broker

The ACTIVEMQ access method uses the HTTP protocol to interact with an ActiveMQ broker by using the RESTful web API. The ActiveMQ MessageServlet class implements the integration between HTTP and the ActiveMQ dispatcher. The SAS ActiveMQ access method uses HTTP POST for sending, and HTTP GET for receiving messages.

For details about the MessageServlet class, see http://activemq.apache.org/maven/5.8.0/activemq-web/apidocs/org/apache/activemq/web/MessageServlet.html at the Apache website.

The REST API can connect to secured brokers. The API uses basic authentication header format to get user ID and password information.

Web Application Contexts

The web context for the ActiveMQ broker URL is specified in the jetty.xml file in the /conf subdirectory of your ActiveMQ installation. The name of your context path is specified by the value of the org.eclipse.jetty.webapp.WebAppContext property.

For ActiveMQ 5.7.0 and earlier, the default context is demo. For ActiveMQ 5.8.0 and later, the default context is api.

Examples:

Example 1: Installing and Testing ActiveMQ from a SAS Program

This example downloads and installs ActiveMQ and then performs a simple send and receive test using the local ActiveMQ installation.

Alternative Download Method

You can also download ActiveMQ manually from activemq.apache.org.

Specify installation parameters. The tempZipFile= macro variable specifies the filename for the downloaded installation file. The installDir= macro variable specifies the directory where ActiveMQ will be installed (ActiveMQ will be installed in a subdirectory named apache-activemq-5.8.0.)

```sas
%let tempZipFile=c:\activemq.zip;
%let installDir=c:\;
```

Download the ActiveMQ binary distribution from the web. If necessary, specify the proxy for your internet connection by using the PROXY= option in the FILENAME statement. If you encounter a connection error, obtain a current recommended URL from ActiveMQ download site: http://www.apache.org/dyn/closer.cgi?path=/activemq/apache-activemq/5.8.0/.

```sas
filename fromweb URL
'http://apache.mirrors.tds.net/activemq/apache-activemq/5.8.0/apache-activemq-5.8.0-bin.zip'
lrecl=8192 recfm=s;

data _null_;
  infile fromweb;
```
file "&tempZipFile" recfm=n lrecl=8192;
input;
put _infile_;
run;

Extract the ActiveMQ ZIP file into the destination directory.

proc groovy;
  add sasjar="ANT";
  submit "&tempZipFile" "&installDir";
  def ant = new AntBuilder();
  ant.unzip( src:args[0], dest:args[1], overwrite:"false" )
  ant.delete( file:args[0] )
endsubmit;
quit;

Start an instance of the ActiveMQ broker.

options noxwait;
x "start &installDir.apache-activemq-5.8.0\bin\activemq";

data _null_; call sleep( 8, 1 ); run; * Give broker time to initialize.;

Put some test messages on the queue.

data _null_; file 'myqueue' activemq url='http://localhost:8161/api/message'
  user=admin pass=admin;
  format a datetime21.1;
  a = datetime(); put a ' Testing: 1 2';
  a = datetime(); put a +1 'This is a test. 3 4';
  a = datetime();
  put a @22 'This is only a test. 5 6';
run;

Get the messages back off of the queue and print them in the log.

data _null_; infile 'myqueue' activemq url='http://localhost:8161/api/message'
  user=admin pass=admin length=len timeout=1;
  input a datetime21.2 @;
  len = len - 25;
  input b $varying60. len c d;
  put a= datetime19. b= c= d=;
run;

Log Output

Each test message appears in the log:

```
a=21FEB2013:13:48:48 b=This is a test. c=3 d=4
a=21FEB2013:13:48:48 b=This is only a test. c=5 d=6
```
Example 2: Using a Message Selector String

This example uses a message selector string to categorize data and to prioritize the order in which the messages for those data values are read.

**Initialize the filerefs for the sample.** If necessary, change the URL=, USER=, and PASSWORD= options to match your ActiveMQ deployment.

```sas
%let url='http://localhost:8161/api/message';
%let moreOpts=user=admin password=admin;
filename requestq activemq url=&url &moreOpts destination=requestq;
filename response activemq url=&url &moreOpts destination=response;
```

**Put some sample hotel rating messages on a request queue.**

```sas
data _null_; length name $10 rating $5; file requestq; do k=0 to 9; name='hotel' || put(k,1.); r=floor(mod(10,k+1)); rating=repeat('A',r); put name rating; end; run;
```

**Read messages from the request queue and then put them onto the response queue with a priority property.** The CTYPE= option specifies the content type application/x-www-form-urlencoded in order to write the JMSPriority message header parameter as part of the message. This enables the ActiveMQ access method to write a different priority value for each message according to the number of As in the rating.

```sas
data _null_; length msg $50 name $6 rating $5; infile requestq; file response cType='application/x-www-form-urlencoded'; input name $ rating $; p=length(rating); msg='&JMSPriority=' || put(p,1.) || '&body='; msg=trim(msg) || trim(name) || ' ' || trim(rating); put msg; run;
```

**Read the messages with a priority greater than 3 from the response queue.** The MSGSELECTOR= option specifies a message selector string.

```sas
data _null_; infile response msgSelector="JMSPriority > 3"; input name $ rating $; put name rating=; run;
```

**Read the messages with a priority greater than 2 from the response queue.** The MSGSELECTOR= option specifies a message selector string.

```sas
data _null_; infile response msgSelector="JMSPriority > 2"; input name $ rating $; put name rating=; run;
```

**Read the remaining messages from the queue.**

```sas
data _null_;
 infile response;
 input;
 put _infile_; run;

Log Output

For each message on the queue, the values are written to the log:

<table>
<thead>
<tr>
<th>Hotel</th>
<th>Rating</th>
</tr>
</thead>
<tbody>
<tr>
<td>hotel5</td>
<td>AAAAA</td>
</tr>
<tr>
<td>hotel6</td>
<td>AAA</td>
</tr>
<tr>
<td>hotel3</td>
<td>AAA</td>
</tr>
<tr>
<td>hotel7</td>
<td>AAA</td>
</tr>
<tr>
<td>hotel0</td>
<td>A</td>
</tr>
<tr>
<td>hotel2</td>
<td>AA</td>
</tr>
<tr>
<td>hotel8</td>
<td>AA</td>
</tr>
<tr>
<td>hotel1</td>
<td>A</td>
</tr>
<tr>
<td>hotel4</td>
<td>A</td>
</tr>
<tr>
<td>hotel9</td>
<td>A</td>
</tr>
</tbody>
</table>

Example 3: Using Request and Response Queues

This example uses separate request and response queues to send messages and then retrieve them by using their message ID.

Initialize the filerefs for the sample. If necessary, change the URL=, USER=, and PASSWORD= options to match your ActiveMQ deployment.

```plaintext
%let moreOpts=url='http://localhost:8161/api/message'
    user=admin password=admin;
filename requestq activemq url=&url &moreOpts destination=requestq;
filename response activemq url=&url &moreOpts destination=response;
```

Put four loan requests on a request queue and save the message IDs. The MESSAGEID= option specifies a variable to store the message ID values.

```plaintext
data requests;
    length name $40 mId $45;
    file requestq messageId=mId deliveryMode=nonPersistent;
    loanAmount=1000; creditScore=675; name='Smith, C.';
    put loanAmount @10 creditScore @15 name;
    output;
    loanAmount=2000; creditScore=500; name='Brown, A.';
    put loanAmount @10 creditScore @15 name;
    output;
    loanAmount=3000; creditScore=750; name='White, D.';
    put loanAmount @10 creditScore @15 name;
    output;
    loanAmount=50000; creditScore=700; name='Jones, B.';
    put loanAmount @10 creditScore @15 name;
    output;
run;
```
Write the request message IDs to the log.

```plaintext
data _null_; set requests; put name= @30 mId=; run;
```

Read the messages from the request queue and then write answers to the response queue by using the message ID as the correlation ID. The MESSAGE= option in the INFILE statement stores the message IDs from the request queue in the MID variable. The CORRELID= option in the FILE statement specifies that values from the MID variable are used as the correlation IDs for the messages on the response queue.

```plaintext
data _null_; length name $40 mId $45; infile requestq messageId=mId length=len; file response correlId=mId deliveryMode=nonPersistent; input loanAmount creditScore @; len = len - 14; input @15 name $varying40. len; if ( creditScore > 600 and loanAmount < 10000 ) then decision = 'approved'; else decision = 'declined'; put decision name;
run;
```

Sort the requests to demonstrate that the messages are not simply read in order.

```plaintext
class proc sort data=requests; by name; run;
```

Read the answers from the response queue and write them to the log. In the INFILE statement, the MATCHOPTS= option specifies that the correlation ID is used to select messages. The CORRELID= option specifies that the CID variable is used for the correlation ID.

```plaintext
%macro getAnswers; %local i nobs; data _null_; if 0 then set requests nobs=nobs; call symput('nobs', put(nobs,1.)); stop; run; %do i=1 %to &nobs; data _null_; length msgName $40 cId $51; obsnum=&i; set requests point=obsnum; cId=mId; infile response matchopts=correlid correlid=cId length=len; input @1 answer $8. @; len = len - 9; input +1 msgName $varying40. len; put '------------------'; put 'Request message ID was ' mId; put name 'with a credit score of ' creditScore 'was ' answer 'for a loan of' loanAmount dollar8.; put '------------------'; stop;
run;
```
Log Output

For each observation in the data set, a message appears in the log:

Request message ID was ID:d73274-57938-1360081257920-3:1:1:1:2
Brown, A. with credit score of 500 was declined for loan of $2,000

Request message ID was ID:d73274-61889-1360699056153-3:1:1:1:4
Jones, B. with credit score of 700 was declined for loan of $50,000

Request message ID was ID:d73274-61889-1360699056153-3:1:1:1:1
Smith, C. with credit score of 675 was approved for loan of $1,000

Request message ID was ID:d73274-61889-1360699056153-3:1:1:1:3
White, D. with credit score of 750 was approved for loan of $3,000

Example 4: Using the FCOPY Function

This example demonstrates using the FCOPY function to copy messages from one fileref to another. It also demonstrates the use of message selector strings and the MSGBODYPREFIX= option.

Initialize the filerefs for the sample. If necessary, change the URL=, USER=, and PASSWORD= options to match your ActiveMQ deployment.

```sas
%let url='http://localhost:8161/api/message/testq?type=queue';
%let moreOpts=user=admin password=admin;
filename myq activemq url=&url &moreOpts destination=testq;
```

Put six messages on a queue named TESTQ. The CORRELID= option specifies the variable that is used for the correlation ID.

```sas
data _null_; 
  file myq correlId=corId;
  j=0;
  do i=1 to 3;
    d=datetime(); j=j+1; corId='Unpaid';
    put j @5 'Went unpaid >30 days.' +1 d datetime21.2 +1 'Xyz';
    d=datetime(); j=j+1; corId='Paid';
    put j @5 'Paid on time.' +1 d datetime21.2 +1 'Xyz';
  end;
run;
```
Reassign the MYQ fileref to add a selector. The MSGSELECTOR= option specifies a message selector string.

```
filename myq activemq url=&url &moreOpts
msgSelector="JMSCorrelationID='Unpaid'";
```

Create a new fileref to change the correlation IDs of the messages on the queue. The MSGBODYPREFIX= option specifies a prefix string to each message. The CTYPEn= option specifies the content type application/x-www-form-urlencoded in order to write the message header parameters (through the MSGBODYPREFIX= option) as part of the message.

```
filename sameq activemq url=&url &moreOpts
ctype='application/x-www-form-urlencoded'
msgBodyPrefix='&JMSCorrelationID=Paid&CorrelationID=Paid&body=';
```

Use the FCOPY function to copy the contents of the MYQ fileref to the SAMEQ fileref. The FCOPY function reads each message through the MYQ fileref and then writes it back to the same queue by using the SAMEQ fileref.

```
data _null_
  rc = fcopy( 'myq', 'sameq' );
  if rc ^= 0 then do;
    msg = sysmsg(); put rc= msg=;
  end;
run;
```

Read the messages from the queue and write them to the log. The CORRELID= option stores the correlation ID of each message in the CORID variable.

```
data _null_
  length msg $50;
  file log;
  infile 'testq' activemq url=&url &moreOpts
    length=len correlid=corid;
  corid='';
  input j @a;
  len = len - 30;
  input @5 msg $varying50. len +1 d datetime21.2 + 1 x $3.;
  put corid= j= d= datetime21.2 msg= x=;
run;
```

Log Output

For each message on the queue, the values are written to the log:

<table>
<thead>
<tr>
<th>CORID</th>
<th>Date</th>
<th>Time</th>
<th>Message</th>
<th>x</th>
</tr>
</thead>
<tbody>
<tr>
<td>Paid</td>
<td>21FEB2013:15</td>
<td>15:22:10.22</td>
<td>msg=Went unpaid &gt;30 days.</td>
<td>Xyz</td>
</tr>
<tr>
<td>Paid</td>
<td>21FEB2013:15</td>
<td>15:22:10.24</td>
<td>msg=Went unpaid &gt;30 days.</td>
<td>Xyz</td>
</tr>
<tr>
<td>Paid</td>
<td>21FEB2013:15</td>
<td>15:22:10.24</td>
<td>msg=Went unpaid &gt;30 days.</td>
<td>Xyz</td>
</tr>
</tbody>
</table>
Chapter 11 / The ACTIVEMQ File Access Method