CHAPTER

37

CALL Routine Interface to Indirect Messaging

Introduction

A CALL routine interface has been implemented to provide access to the distributed messaging services, namely message queueing. This interface has been included with the SAS/CONNECT software and can be used within a SAS data step or SAS macro facility. This interface provides the user with CALL routines that can:

- open or close a station.
- open or close a queue.
- set queue parameter values and types.
- query the queue for messages.
- send and receive messages.
- send and receive attachments.

CALL Routines

These CALL routines alter variable values and provide queuing services. The following is a summary of the supported CALL routines:

STATION_OPEN
Open a station interface for distributed messaging collection services.

STATION_CLOSE
Close a station interface.

QUEUE_OPEN
Open message queue.

QUEUE_SETPARM
Define parameters to be sent to a particular queue.

QUEUE_PARMTYPE
Define parameters and explicitly state their type (char or numeric).

QUEUE_SETHDR
Define queue header information.

QUEUE_SETATT
Define attachments.

QUEUE_ATTOPT
Define additional attachment information.

QUEUE_SEND
Send message to a queue.

QUEUE_QUERY
Query on a message queue.

QUEUE_GETHDR
Obtain queue header information.

QUEUE_RECV
Receive message into variables.

QUEUE_GETFLD
Receive one or more parameters at a time.

QUEUE_GETATT
Obtain attachment information.

QUEUE_ACCEPT
Accept attachment.

QUEUE_COMPLETE
Indicate attachment receipt completion.

QUEUE_GETAGENT
Retreives agent header information.

QUEUE_GETPROP
Get queue properties.

QUEUE_SETPROP
Set queue properties.

QUEUE_GETSEC
Get queue security.

QUEUE_SETSEC
Set queue security.

QUEUE_CLOSE
Closes a queue.

Dictionary

STATION_OPEN

Open a station interface for distributed messaging collection services.

Syntax

CALL STATION_OPEN(*stationId, collectionName, rc*

<, *domainName, securityInfo*>);

Where...	Is type...	And represents...
stationId	N	station id is returned
collectionName	C	name of collection at the DOMAIN server
rc	N	return code
domainName	C	optional domain name
securityInfo	C	optional security info

This CALL routine initializes a station interface and enables access to distributed messaging collection services. The *collectionName* parameter identifies either a new collection that will be created dynamically by the DOMAIN server or a collection that already exists in the DOMAIN server. Each collection name must be unique within a SAS session. The collection name (*collectionName*) will be used by the QUEUE_OPEN routine to add a queue to the "collection", thus allowing indirect-messaging.

The *domainName* is an optional parameter that specifies the name of the DOMAIN server used to provide the collection service. This domain name is also used when a queue is opened.

If the *domainName* is specified, the *securityInfo* parameter may also be specified. This supplies the security string that is needed if the DOMAIN server is running secured.

CAUTION:
***domainName* should be provided.** If the *domainName* is not specified, the domain name should be provided using the macro variable _domain. △

STATION_OPEN Example

This example opens a station interface named DMMAPPL.

```
stid=0;
stname ="DMMAPPL";
rc= 0;
call station_open(stid, stname, rc);
```

STATION_CLOSE

Close a station interface.

Syntax

CALL STATION_CLOSE(*stationId*, *rc*);

Where...	Is type...	And represents...
stationId	N	station id is returned
rc	N	return code

When the STATION_CLOSE CALL routine is executed, the station interface is closed. All active queues on the station will be closed at this time.

After the STATION_CLOSE has executed, the *stationId* is no longer valid. A new station must be opened before further distributed messaging can take place on that station interface.

STATION_CLOSE Example

This example closes a station interface.

```
call station_close(stid, rc);
```

QUEUE_OPEN

Open message queue.

Syntax

CALL QUEUE_OPEN(*queueId*, *stationId*, *queueName*, *mode*, *rc*, <, *attribs*>);

Where...	Is type...	And represents...
queueId	N	queue identifier is returned
stationId	N	station identifier
queueName	C	name of message queue to open
mode	C	open mode
rc	N	return code
attribs	C	optional open attributes

When invoked, QUEUE_OPEN opens a message queue that is associated with a specific collection.

Upon successful open of the queue, the *queueId* is updated and returned. The *queueId* parameter will be used by subsequent calls to identify which queue to act upon. After opening, use the queue identifier to send or receive messages on that message queue (depending on the open mode).

stationId identifies the "collection" that the queue will be associated with. The *stationId* parameter should have been obtained from a call to the STATION_OPEN routine.

The *queueName* parameter is the name of the message queue to open. This name must be unique within its associated collection; however, the same queue name may be used within another collection.

The *mode* parameter indicates the open mode. It should be set to one of the following:

FETCH
FETCH mode enables messages to be retrieved from the queue using the QUEUE_RECV CALL routine. The message is removed from the queue once it is retrieved using the QUEUE_QUERY CALL routine.

FETCHX
This mode exactly like FETCH mode except that it ensures that this open instance has exclusive fetching privileges. Anyone can still open the queue for browsing or delivering messages.

BROWSE
This mode enables the message to be retrieved from the queue without removing it from other instances of the queue. That is, the message still exists on other instances of the queue after browsing it.

DELIVERY
This mode enables messages to be sent to the queue.

If an error or a warning condition is encountered during the open, a non-zero return code is returned in the *rc* parameter. Use the SYSMSG() function to print the message that is associated with the non-zero *rc*.

The following *optional* attributes may be specified in the *attribs* parameter(s):

Optional Attributes

dynamic creation attributes:

DYNPERM

DYNTEMP

additional dynamic attributes

- □ DYN_MSGPSIST
- □ DYN_NOTICE
- □ DYN_MAXDEPTH=*n*
- □ DYN_MAXMSGL=*n*
- □ DYN_REQUIRED.

non-dynamic, instance based attributes:

POLL

ENDPOSITIONING

SURVIVE.

DYNPERM, DYNTEMP
Using one of these attributes causes a queue to be dynamically created. If the queue already exists, a queue contention error is reported. There are two ways to dynamically create a queue. Specify DYNPERM to create a permanent queue that will continue to exist after the queue is closed, or specify DYNTEMP to create a temporary queue that will be deleted automatically when closed. In contrast to

dynamic creation is the idea of predefining a queue before it is opened. You can create an administrator pre-defined queue by defining it during PROC DOMAIN start-up using registration syntax or by defining it through a remote procedural command interface that communicates directives to an executing DOMAIN server. See Chapter 39, “The ADMIN Procedure,” on page 419 for more information.

DYN_MSGPSIST
Dynamic creation attribute that enables message persistence. That is, all messages delivered to this queue will persist on the queue indefinitely or until they are explicitly fetched from the queue. By default, messages do not persist.

DYN_NOTICE
Dynamic creation attribute that enables NOTICE message delivery mode. By default, when a query on a queue executes, it not only retrieves the message header information, but it also retrieves the actual message itself from the queue. At this point, QUEUE_RECV is required to receive the message into SAS variables. This attribute can be specified to override the default behavior so that only message header information is returned from the query. Because only the header information (not the message) is retrieved, a QUEUE_RECV is not required to receive the message. See the QUEUE_QUERY method for more information about how to query a NOTICE message delivery mode queue.

DYN_MAXDEPTH=*n*
Dynamic creation attribute that allows you to specify a maximum queue depth restriction to be placed on this queue. The *n* parameter is an integer value. The default value for *n* is -1, which indicates the depth is unlimited.

DYN_MAXMSGL=*n*
Dynamic creation attribute that allows you to specify a maximum message length restriction to be placed on all messages delivered to this queue. This maximum length must account for additional internal bytes needed to represent the data within each message. Attachment lengths are not taken into consideration, only the length of the actual message itself. The *n* parameter is an integer value. The default value for *n* is -1, which indicates the length is unlimited.

DYN_REQUIRED
Dynamic creation attribute that specifies that, to be successful, this open is required. If a queue already exists, it is used; otherwise, the queue is dynamically created as specified. It is important to point out that this attribute overrides default dynamic creation behavior. By default if a same name queue already exists, you get a contention error. With this attribute specified, a same name queue is opened successfully without error. Also, if a same name queue already exists, it will be opened with existing attribute information (dynamic attributes specified on the open call are ignored).

POLL
The default behavior when querying a queue is to block. This means that the query will not return until a message is received. Therefore, the querying application is blocked until a message is received. To override this default behavior, specify this attribute so that a query will return immediately even if there is no message (non-blocking). This is a valid option for FETCH, FETCHX, and BROWSE open modes.

ENDPOSITIONING
When opening a queue, the default behavior is to position it at the beginning. To override the default, specify this attribute so that it will be positioned at the end of the queue. This means that any messages on the queue prior to the open will not be seen by this queue instance.

SURVIVE
Specify this attribute to ensure that the queue outlives the application. That is, an application can dynamically create a temporary queue that has the DYNTEMP and SURVIVE attributes set, and then exit, leaving the queue to "survive" so that others can use it. If SURVIVE was not specified, the queue would be automatically deleted when it is closed.

QUEUE_OPEN Example

This example opens two Queue instances. The first open specifies the DYNTEMP attribute, which indicates that this queue should not already exist and will, therefore, be dynamically created. It is opened in FETCH mode so that messages can be received from the queue named "inventory". In addition, the POLL attribute is set so that the queries will be non-blocking.

The second open does not specify any dynamic attributes. Therefore, the default action will be used to open the queue. The default means that the queue already exists out there with the same name. This second queue instance is opened with the DELIVERY attribute so that messages can be sent to the queue named "inventory".

f_qid and *d_qid* are returned from the open and can then be used to send and receive messages.

```
   /************************************/
   /* open station instance            */
   /************************************/
stname = "DMMAPPL";
stid = 0;
rc = 0;
call station_open(stid, stname, rc);

   /************************************/
   /* open fetch queue                 */
   /************************************/
f_qid = 0;
qname = "inventory";
mode = "FETCH";
attrib1 = "DYNTEMP";
attrib2 = "POLL";
call queue_open(f_qid, stid, qname, mode,
                rc, attrib1, attrib2);

   /************************************/
   /* open delivery queue              */
   /************************************/
mode = "DELIVERY";
call queue_open(d_qid, stid, qname,
                mode, rc);
```

QUEUE_SETPARM

Define parameters to be sent to a particular queue.

Syntax

CALL QUEUE_SETPARM(*queueId*, *rc*, *parm1*, <*parm2*,...,*parmn*>);

Where...	Is type...	And represents...
queueId	N	queue identifier
rc	N	return code
parm1,...*parmn*	N or C	1 or more numeric or character parameters

QUEUE_SETPARM defines parameters to a particular queue so that they can be sent when the QUEUE_SEND CALL routine is invoked.

The *queueId* parameter identifies the queue.

If an error occurs, *rc* is updated and returned as a non-zero value. Use the SYSMSG() function to print the message that is associated with the non-zero *rc*.

The *parm1*,...,*parmn* parameters are the one or more parameters to send to this particular queue. The parameters are not actually sent until the QUEUE_SEND CALL routine is invoked. Parameters may be numeric or character and may appear in any order.

QUEUE_SETPARM Example

This example defines three parameters to be sent to the queue that is identified by *queueId*. The message is not sent until the QUEUE_SEND CALL routine is invoked.

```
name = "John Doe";
age = 35;
company = "SAS";
call queue_setparm(queueId, rc, name, age,
                   company);
```

QUEUE_PARMTYPE

Define parameters and explicitly state their type (char or numeric).

Syntax

CALL QUEUE_PARMTYPE(*queueId*, *rc*, *parm1Type*, *parm1* <, (*parm2Type*, *parm2*) , ..., (*parm4Type*, *parm4*)>);

Where...	Is type...	And represents...
queueId	N	queue identifier
rc	N	return code

Where...	Is type...	And represents...
parm1Type	C	parameter type
parm1	N or C	parameter value

QUEUE_PARMTYPE defines parameters to send to a specific queue so that they can be sent when the QUEUE_SEND CALL routine is invoked. It takes a parameter-type flag so that the type of parameter may be explicitly set.

If QUEUE_SETPARM is called from within a macro, there is no way to send a numeric value as a string. For example, the macro variable set to **123** could never be sent as a string because QUEUE_SETPARM will view it as a numeric. For this reason, QUEUE_PARMTYPE allows the user to explicitly set the parameter type. So in the above example, the parameter flag could be set to **C** and the parameter value of **123** would be converted to a string by QUEUE_PARMTYPE.

The *queueId* parameter identifies the queue.

If an error occurs, *rc* is updated and returned as a non-zero value. Use the SYSMSG() function to print the message that is associated with the non-zero *rc*.

The *parm1Type* parameter indicates the parameter type. It must have a value of **C** if the parameter is character or a value of **N** if the parameter is numeric.

The *parm1* parameter is the actual parameter value. The parameters are sent when the QUEUE_SEND CALL routine is invoked. Parameters may be numeric or character and may appear in any order.

The parameter type and the parameter value must be specified as a pair that has the type preceding the value. A minimum of one pair (parameter type followed by parameter value) up to the maximum of four pairs may be specified. This routine may be called multiple times if more than four parameters need to be defined in this manner.

QUEUE_PARMTYPE Example

This example defines four parameters to the queue that is identified by *queueId*. In this example, the year passes in a parameter type of **C**. This causes QUEUE_PARMTYPE to convert the numeric into a string and send it as a character parameter.

```
ctype = "C";
ntype = "N";
name = "John Doe";
age = 35;
company = "SAS";
year = 1996;
call queue_parmtype(queueId, rc,
                    ctype, name,
                    ntype, age,
                    ctype, company,
                    ctype, year);
```

QUEUE_SETHDR

Define queue header information.

Syntax

CALL QUEUE_SETHDR(*queueId*, *desc*, *respQ*, *datetime*, *corr*, *rc*);

Where...	Is type...	And represents...
queueId	N	queue identifier
desc	C	user-specified description
respQ	C	user-specified response queue's name
datetime	N	datetime time-out value (currently unsupported, value will be ignored)
corr	N	user-specified correlation value
rc	N	return code

QUEUE_SETHDR defines the queue header information that will accompany the message when the QUEUE_SEND CALL routine is invoked on this particular queue. Any information that is supplied by QUEUE_SETHDR is surfaced on the receiving side by using QUEUE_GETHDR. All parameters are required, but numerics may be set to 0 and strings may be set to double quotes (" ") to indicate that no value should be set for this particular parameter.

The *queueId* parameter identifies the queue.

The *desc* parameter is user-supplied, descriptive text.

The *respQ* parameter is the user-supplied response queue name.

The *datetime* parameter is a time-out date-time stamp. However, this parameter is not supported at this time and should be set to 0.

The *corr* parameter is the user-specified correlator value.

If an error occurs, *rc* is updated and returned as a non-zero value. Use the SYSMSG() function to print the message that is associated with the non-zero *rc*.

QUEUE_SETHDR Example

This example defines the header information to be included with the queue that is identified by *queueId*. Only the description and response queue name are specified.

```
desc = "Information concerning
        employee database.";
resp = "Example Queue";
dt = 0;
corr = 0;
```

```
call queue_sethdr(queueId, desc,resp, dt, corr,rc);
```

QUEUE_SETATT

Define attachments.

Syntax

CALL QUEUE_SETATT(*queueId, rc, atype, ainfo, aname* <, *atype, ainfo, aname...*>);

Where...	Is type...	And represents...
queueId	N	queue identifier
rc	N	return code
atype	C	attachment type
ainfo	C	library name or file specification
aname	C	member name or filename

QUEUE_SETATT defines the attachments that are to be included with the next message that is sent by using the QUEUE_SEND CALL routine. For each attachment, three pieces of information must be specified:

- attachment type
- libname or file specification
- filename or member name.

One or more attachments may be defined, but all three pieces of information must be included for each attachment.

If an error occurs, *rc* is updated and returned as a non-zero value. Use the SYSMSG() function to print the message that is associated with the non-zero *rc*.

The *atype* parameter specifies the attachment type. The *ainfo* and *aname* parameters will be set differently depending on the value of *atype*.

The *atype* parameter should be one of the following:

DATASET or CATALOG
If the attachment is a catalog or a data set, the *ainfo* parameter will be the attachment's library name. The *aname* parameter will be the attachment's member name.

When data set attachments are transferred, all data set attributes are cloned by default. These include label, type, password, encryption, index, and sort information.

EXTERNAL_TEXT or EXTERNAL_BIN
If the attachment is an external file (either binary or text), the *ainfo* parameter will be the file specification. It must have a value of either FILENAME or FILEREF. If *ainfo* is set to FILENAME, the *aname* parameter will contain the attachment's physical filename. If *ainfo* is set to FILEREF, the *aname* parameter will contain the name of the fileref that defines the external file.

QUEUE_SETATT Example

This example defines three attachments to the queue that is defined by *queueId*.

```
   /************************************/
   /* Attachment one is the data set   */
   /* SASUSER.EMPLOYEE.                */
   /************************************/
type1 = "DATASET";
lib1 = "SASUSER";
mem1 = "EMPLOYEE";

   /************************************/
   /* Attachment two is the external   */
   /* text file that is defined by the */
   /* fileref RLINK.                   */
   /************************************/
type2 = "EXTERNAL_TEXT";
fspec = "FILEREF";
fref = "RLINK";

   /************************************/
   /* Attachment three is an external  */
   /* binary file.                     */
   /************************************/
type3 = "EXTERNAL_BIN";
fspec3 = "FILENAME";
fname3 = "/tmp/binary.file";

call queue_setatt(queueId, rc,
                  type1, lib1, mem1,
                  type2, fspec, fref,
                  type3, fspec3, fname3);
```

QUEUE_ATTOPT

Define additional attachment information.

Syntax

CALL QUEUE_ATTOPT(*queueId*, *optName*, *optValue*, *rc* <, *desc*, *idx*, *minorVersion*, *majorVersion*, *attachVersion*>);

Where...	Is type...	And represents...
queueId	N	queue identifier
optName	C	option name
optValue	C	value of option

Where...	Is type...	And represents...
rc	N	return code
desc	C	user-specified description
idx	C	index creation override
minorVersion	N	user-specified minor version
majorVersion	N	user-specified major version
attachVersion	C	attachment version

QUEUE_ATTOPT defines additional options that can be used to subset the attachment data and can also be used to provide useful information to the receiving side. QUEUE_ATTOPT applies to the *last* attachment that was defined by means of the QUEUE_SETATT function.

The *optName* parameter must be one of the following:

DATASET_OPTIONS
This option indicates that *optValue* is a character string that represents any valid data set options. This provides a way to subset the data before the transfer. This option is only valid when the attachment is a data set.

WHERE
This option indicates that *optValue* is a character string that represents a valid WHERE statement. This provides a way to subset the data before the transfer. This option is only valid when the attachment is a data set.

EXCLUDE
This option indicates that *optValue* is a character string that represents an EXCLUDE statement to apply to the catalog. This provides a way to exclude specific entries so that unnecessary entries are not transferred. *OptValue* should take the form of "ENTRY1.ENTRYTYPE ENTRY2.ENTRYTYPE..." for each entry that is to be excluded. The SELECT and EXCLUDE items are mutually exclusive so you can only specify one for a given catalog attachment. This option is only valid when the attachment is a catalog.

SELECT
This option indicates that *optValue* is a character string that represents a SELECT statement to apply to the catalog. This provides a way to select specific entries that are to be included without sending the entire catalog. *OptValue* should take the form of ENTRY1.ENTRYTYPE ENTRY2.ENTRYTYPE... for each entry that is to be selected. The SELECT and EXCLUDE items are mutually exclusive, therefore, you can only specify one for a given catalog attachment. This option is only valid when the attachment is a catalog.

If an error occurs, *rc* is updated and returned as a non-zero value. Use the SYSMSG() function to print the message that is associated with the non-zero *rc*.

The remaining parameters are optional and positional. Supported optional parameters include:

The *desc* parameter is user-specified descriptive text to accompany the attachment. This information is surfaced to the user on the receiving side and allows the user to provide specific information about this attachment. This can be specified for any type of attachments.

The *idx* parameter can have a value of NO or N if the user wants to suppress the transfer of index information. Otherwise, double quotes (" ") may be passed in, to indicate that the default will not be overridden. By default, the data set's index is re-created on the output data set. This parameter allows the default to be overridden so that index creation does not occur. This parameter is specific to data sets and will be ignored by all other attachment types. A value of (" ") may be entered to indicate that it should be ignored.

The *minorVersion* parameter represents a numeric, user-specified version that can be presented to the receiver.

The *majorVersion* parameter represents a numeric, user-specified version that can be presented to the receiver.

The *attachVersion* parameter is only valid in Version 7 or later sessions and only applies to data set and catalog attachments. This parameter allows the user to send the catalog or data set to the queue so that a Version 6 client can read the attachments. If a Version 8 application sends a Version 8 data set or catalog to a queue, a Version 6 application cannot accept that attachment. However, the Version 8 application can specify the value of VERSION_612 for this parameter. This forces the catalog or data set to be sent in Version 6 format so that both Version 6 and Version 8 applications can accept them.

QUEUE_ATTOPT Example

This example defines four attachments to the queue defined by *queueId*.

```
   /************************************/
   /* Attachment one is the data set   */
   /* SASUSER.EMPLOYEE.                */
   /************************************/
type1 = "DATASET";
lib1 = "SASUSER";
mem1 = "EMPLOYEE";
call queue_setatt(queueId, rc, type1,
                  lib1, mem1);

   /************************************/
   /* Define data set options and      */
   /* indicate indexes should not be   */
   /* created on output data set.      */
   /************************************/
opt  = "DATASET_OPTIONS";
optval = "KEEP=SMITH";
desc ="";
idx ="NO";
call queue_attopt(queueId, opt, optval,
                  rc, desc, idx);

   /************************************/
   /* Attachment two is an external    */
   /* binary file.                     */
   /************************************/
type2 = "EXTERNAL_BIN";
fspec2 = "FILENAME";
```

```
fname2 = "/tmp/binary.file";

call queue_setatt(queueId, rc, type2,
                  lib2, mem2);

   /************************************/
   /* no options to define for this    */
   /* attachment                       */
   /************************************/

   /************************************/
   /* Attachment three is the catalog  */
   /* SASHELP.CONNECT.                 */
   /************************************/
type3 = "CATALOG";
lib3 = "SASHELP";
mem3 = "CONNECT";
call queue_setatt(queueId, rc, type3,
                  lib3, mem3);

   /************************************/
   /* Specify SELECT statement to subset*/
   /* transfer of connect catalog.     */
   /************************************/
opt  = "SELECT";
optval = "QUEUE.CLASS STATION.CLASS";
desc = 'Queue and station classes';
idx=""
minor=1;
call queue_attopt(queueId, opt, optval,
                  rc, desc, idx, minor);

   /************************************/
   /* Attachment four is the data set  */
   /* SASUSER.SALARY.                  */
   /************************************/
type4 = "DATASET";
lib4 = "SASUSER";
mem4 = "SALARY";
call queue_setatt(queueId, rc, type4,
                  lib4, mem4);

   /************************************/
   /* Define data set options for      */
   /* SASUSER.SALARY.                  */
   /************************************/
opt  = "DATASET_OPTIONS";
optval = "WHERE=(INCOME>50000)
          KEEP=(FNAME LNAME INCOME)";
call queue_attopt(queueId, opt, optval, rc);
```

QUEUE_SEND

Send message to a queue.

Syntax

CALL QUEUE_SEND(*queueId*, *msgtype*, *rc*);

Where...	Is type...	And represents...
queueId	N	queue identifier
msgtype	N	user-specified message type
rc	N	return code

When QUEUE_SEND is invoked:

- All the parameters that are defined by using QUEUE_SETPARM are sent as a message to be stored in the queue.
- All attachments that are defined by using QUEUE_SETATT are sent along with the message to the queue. If an error occurs while transferring any of the attachments, neither the message nor the attachments are delivered to the queue.
- The delivery header information that was defined by using QUEUE_SETHDR is sent with the message so that it can be surfaced to the receiver.

The *msgtype* parameter is set by the user when the message is sent and is surfaced on the receiving side upon return from the query. When surfaced by the query on the receiving side, the message type can be used to determine how many and what type of parameters should be used in receiving the actual message from the QUEUE_RECV CALL routine.

If an error or a warning condition is encountered during the send, a non-zero return code is returned in the *rc* parameter. Use the SYSMSG() function to print the message that is associated with the non-zero *rc*.

QUEUE_SEND Example

This example causes the string "SAS Institute Inc." and the data set attachment SASUSER.A to be sent to the queue that is identified by *queueId*.

```
company="SAS Institute Inc.";
call queue_setparm(queueId, rc, company);

lib="SASUSER";
mem="A";
type="DATASET";
call queue_setatt(queueId, rc, type,
                  lib, mem);

   /***********************************/
   /* Set message type so that receiving */
```

```
   /* side knows how many and what types */
   /* of parameters to receive.          */
   /************************************/
msgtype = 22;

   /************************************/
   /* Sent message and attachment to    */
   /* queue.                            */
   /************************************/
call queue_send(queueId, msgtype, rc);
```

QUEUE_QUERY

Query on a message queue.

Syntax

CALL QUEUE_QUERY(*queueId, etype, msgtype, attachFlag, rc <, deliveryKey>*);

Where...	Is type...	And represents...
queueId	N	queue identifier
etype	C	event type of received message
msgtype	N	message type of received message
attachFlag	N	attachment flag
rc	N	return code
deliveryKey	N	(optional) delivery key

The QUEUE_QUERY CALL routine queries the queue for a message. If the queue was opened with the POLL attribute and there are no messages on the queue, the query will return immediately and set the event type to NO_MESSAGE. If the queue was not opened with the POLL attribute and there is no message on the queue, the query will block until an event is received on the queue.

Etype is the event type and the variable that is passed in should have a length of at least 17, so that it can be updated with any of the supported event types. Upon return, *etype* will have one of the following values:

DELIVERY
Message received.

NO_MESSAGE
No message on the queue.

ERROR
Queue has been closed or deleted.

END_OF_QUEUE
End of queue has been reached.

Anytime a DELIVERY entry type is returned, the user is required to call QUEUE_GETHDR to retrieve the header information before any subsequent sends or queries will be allowed for this particular queue. It should be noted that this behavior differs from the SCL interface because the SCL interface returns the header information on the query call itself.

The *msgtype* parameter is (optionally) set by the user when the message is sent and is surfaced on the query. When surfaced by the query, the message type can be used to determine how many and what type of parameters should be used in receiving the actual message from the QUEUE_RECV CALL routine.

The *attachFlag* parameter is updated upon return from the query. If the event type is DELIVERY, *attachFlag* indicates whether any attachments were included with the message. If attachments were included with the message, *attachFlag* is set to 1. Otherwise, it will be set to 0. If attachments were included with the message, the CALL routine QUEUE_COMPLETE must be called at some point to indicate that the attachment receipt is complete. Subsequent queries will be prohibited until it is called. QUEUE_COMPLETE must be called even if no attachments were actually accepted.

If an error or a warning condition is encountered during the query, a non-zero return code is returned in the *rc* parameter. Use the SYSMSG() function to print the message that is associated with the non-zero *rc*.

If the NOTICE queue attribute is in effect, the *deliveryKey* parameter is required on the query. Set *deliveryKey* to 0 and call QUEUE_QUERY followed by QUEUE_GETHDR to retrieve the header information of the next message on the queue. If there is a message on the queue, the event type will be set to DELIVERY and the header information will be returned, including *msgtype*. In addition, *deliveryKey* will be updated. This key can be used at a later time to retrieve this message from the queue. To retrieve the actual message, QUEUE_QUERY should be called again, this time specifying the *deliveryKey* parameter that was returned on the initial query.

If the queue was opened without the NOTICE attribute, the *deliveryKey* parameter should not be specified.

QUEUE_QUERY Example 1

This example queries a Queue where the queue was opened in FETCH mode that has the POLL attribute set.

```
call queue_query(queueId, etype, msgtype,
                 attachFlag, rc);
if (etype = "DELIVERY") then do;

      /*********************************/
      /* gethdr required if DELIVERY    */
      /*********************************/
   desc = "";
   resp = "";
   dt = 0;
   corr = 0;
   origin = "";
   security = "";
```

```
   call queue_gethdr(queueId, desc, resp,
                     dt, origin, security,
                     corr, rc);

   if (msgtype = 1) then do;
      /*********************************/
      /* receive parameters            */
      /*********************************/
   end;
end;

   /*********************************/
   /* no message                    */
   /*********************************/
else if (etype = "NO_MESSAGE") then do;
end;
```

QUEUE_QUERY Example 2

This example queries on a Queue where the queue was opened with the NOTICE attribute. If the message type is 4, then call query again to retrieve the actual message on the queue.

```
key = 0;
call queue_query(queueId, etype, msgtype,
                 attachFlag, rc, key);

call queue_gethdr(queueId, desc, resp,
                  dt, origin, security,
                  corr, rc);

if (etype eq "DELIVERY") then do;
   if (msgtype eq 4) then do;
         /*******************************/
         /* specify key value returned  */
         /* by the initial query        */
         /*******************************/
      call queue_query(queueId, etype, msgtype,
                       attachFlag, rc, key);
     end;
end;
```

QUEUE_GETHDR

Obtain queue header information.

Syntax

CALL QUEUE_GETHDR(*queueId*, *desc*, *respQ*, *datetime*, *origin*, *security*, *corr*, *rc*);

Where...	Is type...	And represents...
queueId	N	queue identifier
desc	C	user-specified description
respQ	C	response queue name
datetime	N	queued date-time stamp (currently unsupported, value will be ignored)
origin	C	originator's name
security	C	security name of originator
corr	N	correlator
rc	N	return code

When the query CALL routine returns an event type of DELIVERY, QUEUE_GETHDR must be invoked before any subsequent queries or sends are allowed on this queue.

QUEUE_GETHDR retrieves the queue header information specific to the specified queue. If the user-supplied fields were set on the sending side using QUEUE_SETHDR , those values will be surfaced here. Additional fields will also be surfaced. These include the queue date/time stamp and the originator's name and security name.

The *queueId* parameter identifies the queue.

The *desc* parameter is user-supplied, descriptive text. This text will be surfaced if it was set on the sending side.

The *respQ* parameter is the user-supplied response queue name. This parameter will be surfaced if it was set on the sending side.

The *datetime* parameter is the queued date-time stamp, which indicates when the message was queued. At this time, this parameter is not supported, and any value will be ignored.

The *origin* parameter is the originator's name.

The *security* parameter is the originator's security name.

The *corr* parameter is the correlator value.

If an error occurs, *rc* is updated and returned as a non-zero value. Use the SYSMSG() function to print the message that is associated with the non-zero *rc*.

QUEUE_GETHDR Example

This example obtains the header information of the queue identified by *queueId*.

```
desc = "";
resp = "";
dt = 0;
corr = 0;
origin = "";
security = "";
call queue_gethdr(queueId, desc, resp,
```

```
                    dt, origin, security,
                    corr, rc);
```

QUEUE_RECV

Receive message into variables.

Syntax

CALL QUEUE_RECV(*queueId*, *rc* <, *parm1*,...,*parmn*>);

Where...	Is type...	And represents...
queueId	N	queue identifier
rc	N	return code
parm1,...parmn	N or C	parameters in which to receive the message surfaced by the query; consists of 0 or more numerics or characters

When a message is surfaced by a query, it must be received into either data set variables or macro variables, depending on whether it is invoked with a DATA step or as a macro. QUEUE_RECV supports the receipt of both numerics and characters. QUEUE_RECV must be called with the correct parameter types. For example, if a character and a numeric variable were sent to the queue, QUEUE_RECV must be called with a numeric and a character variable in the correct order.

If an error or a warning condition is encountered during the receive, a non-zero return code is returned in the *rc* parameter. Use the SYSMSG() function to print the message that is associated with the non-zero *rc*.

If an unexpected message is received, QUEUE_RECV can be called with no receive parameters in order to throw away the message. A truncation warning is returned, but the message will have successfully been thrown away.

QUEUE_RECV Example 1

This example queries on a fetch queue, and based on the *msgtype* that is returned, receives the message into the appropriate variables.

```
call queue_query(queueId, etype, msgtype,
                 attachFlag, rc);

if (etype = "DELIVERY") then do;

      /*********************************/
      /* will have some meaning to user */
      /*********************************/
   if (msgtype = 1) then do;
```

```
      name = '';
      age = 0;
      race = '';
         /*******************************/
         /* receive 3 parameters        */
         /*******************************/
      call queue_recv(queueId, rc, name,
                      age, race);
   end;
   else if (msgtype = 5) then do;
         /*******************************/
         /* receive 1 parameter         */
         /*******************************/
     task = 0;
      call queue_recv(queueId, rc, task);
   end;
   else do;
      /*********************************/
      /* unknown message type; throw out*/
      /* message by forcing truncation  */
      /*********************************/
      call queue_recv(queueId, rc);
   end;
end;
```

QUEUE_RECV Example 2

This example throws the message away by forcing truncation.

```
call queue_recv(queueId, rc);
```

QUEUE_GETFLD

Receive one or more parameters at a time.

Syntax

CALL QUEUE_GETFLD(*queueId*, *status*, *rc*,
parm1 <, *parm2*,...,*parmn*>);

Where...	Is type...	And represents...
queueId	N	queue identifier
status	N	status of parameter receipt

Where...	Is type...	And represents...
rc	N	return code
parm1,...parmn	N or C	parameters in which to receive the message; consists of 1 or more numeric or character variables

When a message is surfaced by a query, it needs to be received into variables. QUEUE_GETFLD behaves like QUEUE_RECV in that it receives the message into variables. The two CALL routines differ in that QUEUE_RECV requires that you receive the entire message at one time, while QUEUE_GETFLD allows each parameter to be received separately. QUEUE_GETFLD supports the receipt of numeric and character parameters.

The *status* parameter will have a value of 1 if this is the last parameter to receive. Otherwise, it will have a value of 0.

If an error or warning condition is encountered during the receive, a non-zero return code is returned in the *rc* parameter. Use the SYSMSG() function to print the message that is associated with the non-zero *rc*.

QUEUE_GETFLD Example

This example receives one parameter, then two parameters, then the last one.

```
name1 = '';
name2 = '';
name3 = '';
name4 = '';
status = 0;
call queue_getfld(queueId, status, rc,
                  name1);

if (status ne 1) and (rc eq 0) then
   call queue_getfld(queueId, status, rc,
                     name2, name3);

if (status ne 1) and (rc eq 0) then
   call queue_getfld(queueId, status, rc
                     name4);

/*************************************/
/* STATUS should be set to 1 if this is */
/* the last parameter to be received.   */
/*************************************/
```

QUEUE_GETATT

Obtain attachment information.

Syntax

CALL QUEUE_GETATT(*queueId*, *status*, *attachId*, *atype*, *ainfo*, *aname*, *rc* <, *desc*, *minorVersion*, *majorVersion*>);

Where...	Is type...	And represents...
queueId	N	queue identifier
status	N	status of parameter receipt
attachId	N	attachment identifier
atype	C	type of attachment
ainfo	C	library name or file specification
aname	C	member name or filename
rc	N	return code
desc	C	optional user-specified descriptive text
minorVersion	N	optional minor version
majorVersion	N	optional major version

When a message is surfaced by a query, the QUEUE_QUERY CALL routine returns a flag that indicates whether attachments were included with the message. If attachments were included with the message, QUEUE_GETATT can be invoked to obtain specific attachment information. QUEUE_GETATT returns information for one specific attachment.

The *status* parameter will have a value of 1 if this is the last attachment that was included with the message. Otherwise, it will have a value of 0.

The *attachId* parameter is returned and is the identifier that is used to uniquely identify each specific attachment. It can be used at a later time to indicate what attachments to actually accept.

The *atype* parameter specifies the attachment type. The *ainfo* and *aname* parameters will be set differently based on the attachment type.

DATASET or CATALOG
If the attachment is a catalog or a data set, the *ainfo* parameter will be the library name of the attachment. The *aname* parameter is the member name of the attachment.

EXTERNAL_TEXT or EXTERNAL_BIN
If the attachment is an external file (either binary or text), the *ainfo* parameter will be the file specification. It will have a value of either FILENAME or FILEREF. If *ainfo* is set to FILENAME, then the *aname* parameter will contain the attachment's physical filename. If *ainfo* is set to FILEREF, then the *aname* parameter contains the name of the fileref that defines the external file.

If an error or a warning condition is encountered, a non-zero return code is returned in the *rc* parameter. Use the SYSMSG() function to print the message that is associated with the non-zero *rc*.

If the sender specified an attachment description, it is returned in the *desc* parameter.

If the sender specified a user-specifiable minor version number, it is returned in the *minorVersion* parameter.

If the sender specified a user-specifiable major version number, it is returned in the *majorVersion* parameter.

QUEUE_GETATT Example

This example obtains attachment information. Notice the use of the optional parameters on the second and third QUEUE_GETATT invocations.

```
call queue_query(queueId, etype, msgtype,
                 attachFlag, rc);

if (etype = "DELIVERY") and
   (attachFlag = 1) then do;

   status = 0;
   attachId = 0;
   desc='';
   call queue_getatt(queueId,  status,
                     attachId, type,
                     info, mem, rc);

      /*********************************/
      /* more attachment info to obtain */
      /*********************************/
   if (status = 0) then
      call queue_getatt(queueId, status,
                        attachId, type,
                        info, mem, rc, desc);

      /*********************************/
      /* more attachment info to obtain */
      /*********************************/
   if (status = 0) then
      call queue_getatt(queueId, status,
                        attachId, type,
                        info, mem, rc, desc,
                        minor);

end;
```

QUEUE_ACCEPT

Accept attachment.

Syntax

CALL QUEUE_ACCEPT(*queueId*, *attachId*, *ainfo*, *aname*, *rc*);

Where...	Is type...	And represents...
queueId	N	queue identifier
attachId	N	attachment identifier
ainfo	C	library name or file specification
aname	C	member name or filename
rc	N	return code

When a message is surfaced by a query, the QUEUE_QUERY CALL routine returns a flag that indicates whether attachments were included with the message. If attachments were included with the message, QUEUE_GETATT can be invoked to obtain specific attachment information. QUEUE_GETATT returns information for one specific attachment and it identifies that attachment by the attachment identifier. The attachment identifier returned by QUEUE_GETATT should be specified in the *attachId* parameter to identify which attachment you are accepting. QUEUE_ACCEPT accepts the attachment; that is, the attachment is transferred and written out to the file that is specified by the *ainfo* and *aname* parameters.

If the attachment that is being accepted is a catalog or a data set, the *ainfo* parameter should specify the output library name. The *aname* parameter should specify the output member name.

If the attachment is an external file (either binary or text), the *ainfo* parameter should indicate the file specification. It must have a value of either FILENAME or FILEREF. If *ainfo* is set to FILENAME, the *aname* parameter should specify the output file's physical filename. If *ainfo* is set to FILEREF, then the *aname* parameter should indicate the name of the fileref that defines the output external file.

If an error or a warning condition is encountered, a non-zero return code is returned in the *rc* parameter. Use the SYSMSG() function to print the message that is associated with the non-zero *rc*.

QUEUE_ACCEPT Example

This example accepts the attachment that is identified by *attachId* into the data set SASUSER.CENSUS.

```
lib = "SASUSER";
mem = "CENSUS";
call queue_accept(queueId, attachId,
                  lib, mem, rc);
```

QUEUE_COMPLETE

Indicate attachment receipt completion.

Syntax

CALL QUEUE_COMPLETE(*queueId, rc*);

Where...	Is type...	And represents...
queueId	N	queue identifier
rc	N	return code

When a message is surfaced by a query, the QUEUE_QUERY CALL routine returns a flag that indicates whether attachments were included with the message. If the query returns the attachment flag set to 1, indicating that attachments were included with the message, QUEUE_COMPLETE must be called at some point to indicate that attachment receipt is complete. If attachments are to be accepted, the QUEUE_ACCEPT calls should be made before the call to QUEUE_COMPLETE. However, the CALL routine must be called at some point whether or not any attachments were actually accepted. Subsequent queries will be prohibited until QUEUE_COMPLETE is called.

If an error or a warning condition is encountered, a non-zero return code is returned in the *rc* parameter. Use the SYSMSG() function to print the message that is associated with the non-zero *rc*.

QUEUE_COMPLETE Example

This example uses QUEUE_COMPLETE to indicate that the attachment receipt is complete.

```
call queue_complete(queueId, rc);
```

QUEUE_GETAGENT

Retreives agent header information.

Syntax

CALL QUEUE_GETAGENT(*queueId, agentName, runKey, dateTime, completionCode, rc*);

Where...	Is type...	And represents...
queueId	N	queue identifier
agentName	C	name of defined agent
runKey	N	run instance key
dateTime	N	date and time agent was run
completionCode	N	agent run completion code
rc	N	return code

The QUEUE_GETAGENT routine retrieves the agent header information from the specified queue. When the agent runs, you have the ability to define a notification queue. If a notification queue is defined, a completion message is sent to this queue with a message type of 65539.

The *agentName* parameter is the name of an agent that has already been defined using the SCL _defineAgent.

The header information that is returned includes *runkey*, *dateTime*, and *completionCode*.

The *runKey* parameter is used in the SCL interface to retrieve the agent run information.

The *dateTime* parameter indicates the date and time that the agent was run.

The *completionCode* parameter indicates whether the agent was successfully invoked. A value of zero indicates the agent was invoked successfully.

If an error or a warning condition is encountered, a non-zero return code is returned in the *rc* parameter. Use the SYSMSG() function to print the message that is associated with the non-zero *rc*.

QUEUE_GETAGENT Example

This example queries for a message and displays agent information if agent completed successfully.

```
msgtype = 0;
aflag = 0;
put ' ';
call queue_query(qid, etype, msgtype, aflag, rc);

if rc = 0 and (etype = DELIVERY) then do;
   put 'Query successful';
   put 'Etype is ' etype;
   put 'Msgtype is ' msgtype;

      /* agent completion message received, */
      /* get agent info                     */
   if (msgtype = 65539) then do;
      retrieved = retrieved + 1;
      agentName = '';
      dt = 0;
```

```
      cc = 0;
      runkey = 0;
      put ' ';
      call queue_getagent(qid, agentName, runkey,
                          dt, cc, rc);

      if rc = 0 then do;
         put ' ';
         put ' ';
         put 'GetAgent successful';
         put 'Agent name is ' agentName;
         put 'Runkey is ' runkey;
         put 'Datetime is ' dt;
         put 'Completion code is ' cc;
      end;
   else do;
      msg = sysmsg();
      put msg;
   end;
   corr = 0;
   put ' ';
   put 'Calling queue_gethdr';
   call queue_gethdr(qid, desc, respQ, dt,
      origin, security, corr, rc);

   if rc = 0 then do;
      put 'Gethdr successful';
      put 'Desc is ' desc;
      end;
   else do;
      msg = sysmsg();
      put msg;
   end;
end;  /* if msgtype is 65539 */
```

QUEUE_GETPROP

Get queue properties.

Syntax

CALL QUEUE_GETPROP(*queueId, rc, type, def, msgpsist, dlvrmode, crdt, depth, maxdepth, maxmsgl*);

Where...	Is type...	And represents...
queueId	N	queue identifier
rc	N	return code

Where...	Is type...	And represents...
type	C	indicates what happens to a queue after the queue is closed
def	C	defines how the queue was created
msgpsist	C	message persistence enablement
dlvrmode	C	message delivery mode
crdt	N	queue creation date/time stamp
depth	N	queue current depth
maxdepth	N	queue maximum depth allowed
maxmsgl	N	queue maximum message length allowed

The QUEUE_GETPROP routine is used to retrieve the properties that are associated with a queue.

If an error or a warning condition is encountered, a non-zero return code is returned in the *rc* parameter. Use the SYSMSG() function to print the message that is associated with the non-zero *rc*.

The *type* parameter indicates if the queue is defined to be temporary or permanent.

TEMPORARY
The queue is deleted after it is closed.

PERMANENT
The queue continues to exist after it is closed.

The *def* parameter is used to specify if the queue is defined by using pre-defined attributes or dynamic creation attributes.

PREDEFINED
DYNAMIC

The *msgpsist* parameter indicates whether messages delivered to this queue will persist on the queue indefinitely or until they are explicitly fetched from the queue or until the queue is closed.

YES	Messages will persist.
NO	Messages will not persist.

The *dlvrmode* parameter indicates the queue's message delivery mode:

DEFAULT
A query on the queue retrieves the message header information as well as the actual message.

NOTICE
A query on the queue retrieves the message header information only.

The *crdt* parameter is the date and time when the queue was created.

The *depth* parameter indicates the current number of messages on the queue (depth of the queue).

The *maxdepth* parameter indicates the maximum number of messages that can be held by the queue (-1 is unlimited).

The *maxmsgl* parameter indicates the maximum length of a message for the queue (-1 is unlimited).

QUEUE_GETPROP Example

This example prints the information that is obtained about a queue.

```
length type def msgpsist dlvrmode $20;
length crdt depth maxdepth maxmsgl 8;

call queue_getprop(qid, rc, type, def, msgpsist,
   dlvrmode, crdt, depth, maxdepth, maxmsgl);

if rc = 0 then do;
   put 'Queue properties:';
   put 'type = ' type;
   put 'definition = ' def;
   put 'msgpsist = ' msgpsist;
   put 'dlvrmode = ' dlvrmode;
   put 'creation date/time = ' crdt datetime.;
   put 'depth = ' depth;
   put 'maxdepth = ' maxdepth;
   put 'maxmsgl = ' maxmsgl;
end;
else do;
   msg = sysmsg();
   put msg;
end;
```

QUEUE_SETPROP

Set queue properties.

Syntax

CALL QUEUE_SETPROP(*queueId, rc, dlvrmode, maxdepth, maxmsgl*);

Where...	Is type...	And represents...
queueId	N	queue identifier
rc	N	return code
dlvrmode	C	message delivery mode

Where...	Is type...	And represents...
maxdepth	N	queue maximum depth allowed
maxmsgl	N	queue maximum message length allowed

The QUEUE_SETPROP routine allows you to set certain queue properties. In particular, you may set message delivery mode if there are no active (open) fetch or browse queue instances. You may also set the maximum queue depth as well as the maximum message length.

If an error or a warning condition is encountered, a non-zero return code is returned in the *rc* parameter. Use the SYSMSG() function to print the message that is associated with the non-zero *rc*.

The *dlvrmode* parameter indicates the queue's message delivery mode:

DEFAULT
A query on the queue retrieves the message header information as well as the actual message.

NOTICE
A query on the queue retrieves the message header information only.

The *maxdepth* parameter indicates the maximum number of messages that can be held by the queue (-1 is unlimited).

The *maxmsgl* parameter indicates the maximum length of a message for the queue (-1 is unlimited).

QUEUE_SETPROP Example

This example re-sets the message delivery mode to NOTICE as well as sets the maximum depth and maximum message length. If you do not want to set a particular queue property, set its value to an empty string if its type is character, or set its value to missing if its type is numeric.

```
length dlvrmode $20;
length maxdepth maxmsgl 8;

dlvrmode="notice";
maxdepth=50;
maxmsgl=4096;
call queue_setprop(qid, rc, dlvrmode,
                   missing, missing);

if rc ^= 0 then do;
   msg = sysmsg();
   put msg;
end;
else put 'Setprop was successful';
```

QUEUE_GETSEC

Set queue security.

Syntax

CALL QUEUE_GETSEC(*queueId, rc, security*);

Where...	Is type...	And represents...
queueId	N	queue identifier
rc	N	return code
security	C	security permissions string

The QUEUE_GETSEC routine allows you to obtain information about the permissions or privileges that are associated with a particular queue.

If an error or a warning condition is encountered, a non-zero return code is returned in the *rc* parameter. Use the SYSMSG() function to print the message that is associated with the non-zero *rc*.

The *security* parameter is the access control list for the queue. This parameter is returned in the form

'user1:*permissions*,user2:*permissions*,...'

where *permissions* is one or more of the following separated by a plus sign (+):

DELIVER|D
Deliver privileges.

FETCH|F
Fetch privileges.

BROWSE|B
Browse privileges.

GETPROP|GP
Get properties privileges.

SETPROP|SP
Set properties privileges.

GETSEC|GS
Get security privileges.

SETSEC|SS
Set security privileges.

ALL
Full privileges.

QUEUE_GETSEC Example

This example obtains a list of user privileges for a specific queue.

```
length security $200;
security="";
```

```
call queue_getsec(qid, rc, security);

if rc ne 0 then do;
   msg = sysmsg();
   put msg;
end;
else do;
   put 'User Access Rights:';
   put security;
end;
```

QUEUE_SETSEC

Set queue security.

Syntax

CALL QUEUE_SETSEC(*queueId, rc, security*);

Where...	Is type...	And represents...
queueId	N	queue identifier
rc	N	return code
security	C	security permissions string

The QUEUE_SETSEC routine allows you to set the permissions or privileges that are associated with a specific queue.

If an error or a warning condition is encountered, a non-zero return code is returned in the *rc* parameter. Use the SYSMSG() function to print the message that is associated with the non-zero *rc*.

The *security* parameter is the access control list for the queue. This parameter is sent in the form of

'user1:*permissions*,user2:*permissions*,...'

where *permissions* is one or more of the following separated by a plus sign (+):

DELIVER|D
Deliver privileges.

FETCH|F
Fetch privileges.

BROWSE|B
Browse privileges.

GETPROP|GP
Get properties privileges.

SETPROP|SP
Set properties privileges.

GETSEC|GS
Get security privileges.

SETSEC|SS
Set security privileges.

ALL
Full privileges.

QUEUE_SETSEC Example

This example sets two user privileges for a specific queue. The first user (USER1) is defined to have all or full privileges. Full privileges consist of the following: deliver, fetch, browse, get properties, set properties, get security, and set security. The second user (USER2) is defined to have only browse, get properties, and get security privileges.

```
length security $200;
security="user1:all,user2:b+gp+gs";
call queue_setsec(qid, rc, security);

if rc ne 0 then do;
   msg = sysmsg();
   put msg;
end;
else put 'SetSec was successful';
```

QUEUE_CLOSE

Closes a queue.

Syntax

CALL QUEUE_CLOSE(*queueId*, *rc*<, *attribs*>);

Where...	Is type...	And represents...
queueId	N	queue identifier
rc	N	return code
attribs	C	(optional) attributes

When invoked on a *queueId*, QUEUE_CLOSE closes the queue. The *queueId* is no longer open and no subsequent messaging can occur on this *queueId*.

If an error or a warning condition is encountered during the close, a non-zero return code is returned in the *rc* parameter. Use the SYSMSG() function to print the message that is associated with the non-zero *rc*.

The following optional *attribs* may be specified on the close:

SURVIVE
This attribute indicates that the queue will not be purged from memory. Its purpose is to allow temporary queues a way to survive an initial close, thereby preserving the queue for the life of the DOMAIN server without having to back messages to disk.

DELETE
This attribute causes a permanent dynamic queue to be deleted if no messages reside on the queue. If messages still exist on the queue, the queue is closed, but a warning is returned to designate that the queue was not deleted as intended. If you use this attribute to close an administrator pre-defined queue, a warning is returned because these types of queues can only be deleted by an administrator using PROC ADMIN. This attribute is ignored when closing temporary queues because they are automatically deleted when the creating instance closes it.

DELETE_PURGE
This attribute behaves exactly as the DELETE attribute does but there is one difference. DELETE_PURGE causes a permanent dynamic queue to be deleted even if messages remain on the queue.

The correct bibliographic citation for this manual is as follows: SAS Institute Inc., *SAS/CONNECT User's Guide, Version 8*, Cary, NC: SAS Institute Inc., 1999. pp. 537.

SAS/CONNECT User's Guide, Version 8

ISBN 1–58025–477–2

SAS Institute Inc., SAS Campus Drive, Cary, North Carolina 27513.

1st printing, September 1999