CHAPTER

35

SAS Component Language (SCL) Interface to Indirect Messaging

Introduction

In some instances, you do not want all the programs in your application to run at the same time, nor do you want them to be synchronized (one side sends a message and waits for a reply before it can send another message). These restrictions disappear with SAS indirect messaging (message queuing). Indirect messaging enables programs to communicate indirectly by placing messages on queues in storage. Therefore, the pieces of your application can run independently of each other, at different speeds and times, and without a direct connection between them.

Indirect messaging provides capabilities that enable applications developers to deploy multi-tiered distributed applications based on a message-passing paradigm. This multi-tiered design allows you to separate and centralize business logic and data access from the client environment. Servers, which receive requests and return responses, may be implemented through the utilization of simple yet flexible message construction, transmission, and notification services that span operating system and hardware boundaries across the enterprise.

Messages are free-form. Their structure is defined by the applications developer and may range from a simple collection of variables to complex hierarchies of SCL lists. Additionally, messages may include one or more attachments which can take the form of SAS data sets or filtered subsets, catalogs or catalog entries, external files, MDDB files, DMDB files, FDB files, and SQL Views.

The asynchronous messaging capability is especially beneficial because it does not require the intended message target to be active when a request is sent. For example, long-running transactions can be batched for off-hours execution, and due to its non-blocking semantics, multiple requests can be dispatched concurrently across parallel server processes.

Similar to direct messaging, indirect messaging requires that stations be established by using the Station class. However, because direct connections are not required by indirect messaging, the Station class QUERY method function is not valid when using indirect messaging. The Queue class provides services to message queues (such as opening, closing, and querying a queue) as well as sending and receiving messages.

Station Class

Access to all distributed messaging services is obtained by opening a "collection" using the station class interface instance. After it is created and opened, the station interface instance can be used for indirect (queued) messaging to access a collection at the DOMAIN server. Unlike direct messaging, queries cannot be performed on stations for indirect messaging.

The Station class is used to open and close stations for both direct and indirect messaging. The instance methods defined to the Station class are:

_open
Open a station interface instance for distributed messaging collection services.

_close
Close a station interface instance.

For information about using the Station class, refer to Chapter 29, “Using Direct Messaging,” on page 273 .

Collection Manager

A "collection" is simply a user-defined grouping of queues. You can decide what queues to group together to form a collection. Each queue must be associated with a collection and must have a unique name within the collection.

You may use the same queue name in different collections, and it will represent different queues. For example, you may have a queue named QUEUE_A in COLLECTION1 and in COLLECTION2. The name can be the same because they are in two different collections, but they represent two completely independent queues.

A collection is created using either the SCL Station class _open method or the CALL routine STATION_OPEN. Both interfaces return a station identifier that is used when opening a queue.

The collection provides a level of management for the queues by using a collection manager. The collection manager is responsible for starting the queue manager that processes the individual messages for each queue, and allows you to access the messages using the Queue class.

Queue Class Overview

Indirect-messaging services are provided by the Queue class. These services include opening, closing and querying a queue, as well as sending and receiving messages. Messages can consist of characters, numerics, SCL lists and parameters. Messages can also include attachments.

In addition to the SCL interface, an indirect-message functional interface is also available that uses CALL Routines (see Chapter 37, “CALL Routine Interface to Indirect Messaging,” on page 375).

PARENT:
SASHELP.FSP.OBJECT.CLASS

CLASS:
SASHELP.CONNECT.QUEUE.CLASS

Queue Class Usage

For information about using the Queue class, refer to Chapter 31, “Using Indirect Messaging,” on page 281.

Queue Class Methods

The instance methods defined to the Queue class are:

_open
Open message queue instance.

_send
Send message to a queue.

_sendlist
Send SCL lists to a message queue.

_query
Query on a message queue.

_recv
Receive message into SCL variables.

_recvlist
Receive SCL lists.

_getfield
Receive one or more SCL parameters at a time.

_getlist
Receive one or more SCL lists at a time.

_acceptAttachment
Receive attachments.

_getprop
Get properties privileges.

_setproc
Set properties privileges.

_getsec
Get security privileges.

_setsec
Set security privileges.

_close
Close a message queue.

Notation that is used to explain the parameter types is as follows:

C	Character Type
N	Numeric Type
L	SCL List Type

Dictionary

_open

Open message queue instance.

Syntax

CALL SEND(*queueInst*, '_open', *stationInst*, *queue_name*, *mode*, *rc* <, *attrib1...attribn*>);

Where...	Is type...	And represents...
stationInst	N	successfully opened station instance
queue_name	C	name of message queue to open
mode	C	open mode
rc	N	return code
attrib1...	C	zero or more optional attributes
attribn		

When invoked on a Queue instance, _open opens a message queue. The user is responsible for obtaining an instance of the Queue class, and then sending the _open method to it. After it is opened, the Queue instance can be used to send or receive messages to the message queue (depending on the open mode).

A station interface instance should already have been opened and should be passed in as the *stationInst*.

queue_name is the name of the message queue to open.

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

FETCH
enables messages to be retrieved from the queue by using the _recv or _recvlist methods. The message is removed from the queue after it is retrieved by using the _query method.

FETCHX
is exactly like FETCH mode except that it ensures that this open instance has exclusive fetching privileges. The queue can still be opened for browsing and delivering messages as described later.

BROWSE
enables the message to be retrieved from the queue instance 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
enables messages to be sent to the queue.

If an error or warning condition is encountered during the open, a non-zero return code is returned in the *rc* parameter. The return codes that follow are a defined set of error conditions that can be checked by using the SYSRC macro, which is provided in the autocall library that is supplied by SAS Institute.

If the *rc* is not one of the messages shown here, use SYSMSG() to determine the exact error message.

_SEINVMO
indicates that the mode that is specified on the open is invalid. Valid modes are FETCH, FETCHX, BROWSE or DELIVERY.

_SEINVAT
indicates that one of the (optional) attributes is invalid. Valid attributes are listed below.

_SEQEXST
indicates an error occurred because of a contention failure; the queue already exists.

_SEQDPD
indicates delivery permission was denied; user does not have permission to deliver messages to the queue.

_SEQFPD
indicates fetch permission was denied; user does not have permission to fetch messages from the queue.

_SEQBPD
indicates browse permission was denied; user does not have permission to browse messages from the queue.

Optional Attributes

Here are a number of optional attributes that may be specified in the attribs parameter(s).

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
There are two ways to dynamically create a queue at open time. 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 pre-defining a queue before it is opened. You can create an administrator predefined queue by either defining it during PROC DOMAIN start-up using registration syntax or by defining it through a remote procedural command interface (Chapter 39, "The ADMIN Procedure," on page 419) that communicates directives to an executing DOMAIN server.

DYN_MSGPSIST
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
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, _recv is required to receive the message into SAS variables. The DYN_NOTICE 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, and not the message, is retrieved, a _recv is not required to receive the message. See the _query method for more information about how to query a NOTICE message delivery mode queue.

DYN_MAXDEPTH=*n*
allows you to specify a maximum queue depth restriction to be placed on this queue.

DYN_MAXMSGL=*n*
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.

DYN_REQUIRED
specifies that this open is required to be successful. 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 will get a contention error. With this attribute specified, a same name queue will be opened successfully without any 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 the queue at the beginning. To override the default, specify this attribute so that the queue is positioned at the end. 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. This means that an application can dynamically create a temporary queue when the

DYNTEMP and SURVIVE attributes are set and then exit. This leaves the queue to "survive" so that others can use it. If SURVIVE was not specified, the temporary queue is deleted automatically when the queue is closed.

Example

This example opens two Queue instances. The first open specifies the DYNTEMP attribute, which indicates that this queue should not already exist, but it will be dynamically created. It will be 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 will be opened with the DELIVERY attribute so that messages can be sent to the queue that is named "inventory".

After the queues are opened, FETCHQ and DELIVQ can be used to send and receive messages.

```
stationid = loadclass('sashelp.connect.station');
stationInst = instance(stationid);
collectionName = "DMMAPPL";

call send(stationInst, "_open", collectionName, rc);

queueid = loadclass('sashelp.connect.queue');
fetchq = instance(queueid);
call send(fetchq, '_open', stationInst, "inventory",
          "FETCH", rc, "DYNTEMP");

delivq = instance(queueid);
call send(delivq, '_open', stationInst, "inventory",
          "DELIVERY", rc, "POLL");
```

_send

Send message to a queue.

Syntax

CALL SEND(*queueInst*, '_send', *msgtype*, *header*,
attachlist, *rc* <, *parm1*,...,*parmn*>);

Where...	Is type...	And represents...
msgtype	N	user-specified message type
header	L	delivery header list (or 0 if none)
attachlist	L	attachment list (or 0 if none)
rc	N	return code
parm1...parmn	N or C	message to send, which consists of 0 or more numerics or characters in any order

The _send method allows a message to be sent to a message queue. A message can consist of numerics and/or characters. SCL lists are not supported by _send. Use _sendlist to send SCL lists.

The *msgtype* parameter is set by the user when the message is sent. *msgtype* will be surfaced on the receiving side upon return from 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 by using the _recv method.

The delivery *header* is an SCL list that may be specified on the send. Information that can be supplied in the delivery header includes descriptive user-supplied text, a response queue name and a user-definable correlation value. A 0 may be specified if there is no header information to send. Otherwise, the header information may be specified by creating an SCL list that has one or more of the following named items set accordingly:

DESCRIPTOR
Descriptive, user-supplied text.

RESPONSE_QUEUE_NAME
User-supplied response queue name.

CORRELATOR
User-supplied correlation value.

The *attachlist* parameter is an SCL list that indicates a list of attachments to be sent with the message. If there are no attachments to send, a zero should be specified. Otherwise, a valid attachment list should be passed in; this attachment list will be surfaced by the query on the receiving side. The receiving side then has the flexibility to decide which, if any, attachments to receive. If an error occurs while sending attachments to the queue, the message and its attachments are NOT delivered to the queue; instead an appropriate error message is returned. See “Sending Attachments” on page 358 for more information about *attachlist*.

If an error or warning condition is encountered during the send, a non-zero return code is returned in the *rc* parameter. The return codes that follow are a defined set of warning or error conditions that can be checked by using the SYSRC macro, which is provided in the autocall library that is supplied by SAS Institute.

If the *rc* is not one of the messages shown here, use SYSMSG() to determine the exact error message.

_SEATTXF
indicates that neither the message nor the attachments were sent to the queue because an error was encountered during attachment transfer. See “Attachment Error Handling” on page 370 for more information.

_SEACTRM
indicates that the connection was abnormally terminated. If this return code is surfaced, the queue was abnormally closed. The queue must be re-opened before any subsequent processing can take place on that queue instance.

The *parm1...parmn* parameters are the 0 to *n* numeric and/or character values that are sent. Any number of parameters can be sent in any order.

Example

This example invokes the _send method on the Queue instance to send five parameters with no attachments.

```
name = "John Doe"
age = 35;
company = "SAS";
code = 13484;
type = 472;
attach = 0;
msgtype = 22;

   /* set the delivery header fields */
header = makelist();
rc = setnitemc(header, "This message contains
                names and ages","DESCRIPTOR");
rc = setnitemc(header, "inventory",
               "RESPONSE_QUEUE_NAME");

call send(queueInst, '_send', msgtype, header,
          attach, rc, name, age, company, code,
          type);
```

_sendlist

Send SCL lists to a message queue.

Syntax

CALL SEND(*queueInst*, '_sendlist', *msgtype*, *header*, *attachlist*, *rc* <, *list1...listn*>);

Where...	Is type...	And represents...
msgtype	N	user-specified message type
header	L	delivery header list (optional)
attachlist	L	attachment list (or 0 if none)
rc	N	return code
list1...listn	L	message to send which consists of 0 or more SCL lists

The _sendlist method allows one or more SCL lists to be sent to a message queue. Any type of SCL list is supported (named lists, unnamed lists, embedded lists, and recursive lists).

The *msgtype* parameter is set by the user when the message is sent and will be surfaced on the receiving side upon return from 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 by using the _recvlist method.

The delivery *header* is an SCL list that may be specified on the send. Information that can be supplied in the delivery header includes descriptive user-supplied text, a response queue name, and a user-definable correlation value. A 0 may be specified if there is no header information to send. Otherwise, the header information may be specified by creating an SCL list that has one or more of the following named items set accordingly:

DESCRIPTOR
Descriptive, user-supplied text.

RESPONSE_QUEUE_NAME
User-supplied response queue name.

CORRELATOR
User-supplied correlation value

The *attachlist* parameter is an SCL list that indicates a list of attachments to be sent with the message. If there are no attachments to send, a 0 should be specified. Otherwise, a valid attachment list should be passed. This attachment list will be surfaced by the query on the receiving side. The receiving side then has the flexibility to decide which (if any) attachments to receive. If an error occurs while sending attachments to the queue, the message and its attachments are NOT delivered to the queue; instead an appropriate error message is returned. See “Sending Attachments” on page 358 for more information about *attachlist*.

If an error or warning condition is encountered during the send, a non-zero return code is returned in the *rc* parameter. The return codes that follow are a defined set of warning or error conditions that can be checked by using the SYSRC macro, which is provided in the autocall library that is supplied by SAS Institute.

If the *rc* is not one of the messages shown here, use SYSMSG() to determine the exact error message.

_SEATTXF
indicates that neither the message nor the attachments were sent to the queue because an error was encountered during attachment transfer. See “Attachment Error Handling” on page 370 for more information.

_SEACTRM
indicates that the connection was abnormally terminated. If this return code is surfaced, the queue was abnormally closed. It must be re-opened before any subsequent processing can take place on that queue instance.

The *list1...listn* parameters are the 0 to *n* SCL lists to send.

Example

This example invokes the _sendlist method on the Queue instance to send two SCL lists with no attachments.

```
   /************************************/
   /* first list to send has 4 items    */
   /************************************/
namelist = makelist();
rc=setnitemc(namelist, "Mary Gill",  "STUDENT");
rc=setnitemc(namelist, "Jane Smith", "TEACHER");
rc=setnitemc(namelist, "Julie Jones","PRINCIPAL");
rc=setnitemc(namelist, "Bob Thomas", "COACH");

   /************************************/
   /* second list to send is a list     */
   /* that contains two embedded lists  */
   /************************************/
mainlist = makelist();

data1 = makelist();
rc = setitemc(data1, 'WORK.ABC', 1);
rc = setitemc(data1, 'SASUSER.COMPANY', 2);
rc = setitemc(data1, 'SASUSER.LOCATION',1);

data2 = makelist();
rc = setitemc(data2, 'SASHELP.BASE',1);
rc = setitemc(data2, 'SASHELP.EIS', 2);

   /************************************/
   /* insert the above two lists into   */
   /* mainlist                          */
   /************************************/
mainlist = insertl(mainlist, data1);
mainlist = insertl(mainlist, data2);

msgtype = 3;

   /************************************/
   /* set the delivery header fields,   */
   /* descriptor & response_queue_name; */
   /* they must be set as named items   */
   /* in the delivery header list       */
   /************************************/
header = makelist();
rc = setnitemc(header,
```

```
                "This message contains lists",
                "DESCRIPTOR");
rc = setnitemc(header, "school",
                "RESPONSE_QUEUE_NAME");

attachlist = 0;

call send(queueInst, '_sendlist', msgtype,
          header, attachlist, rc, namelist,
          mainlist);
```

_query

Query on a message queue.

Syntax

CALL SEND(*queueInst*, '_query', *etype*, *msgtype*, *header*, *attachlist*, *rc* <, *delivery_key;*>);

Where...	Is type...	And represents...
etype	C	event type of received message
msgtype	N	message type of received message
header	L	delivery header list
attachlist	L	attachment list associated with message
rc	N	return code
delivery_key	N	(optional) delivery key

The _query method 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. That is, the query will not return until a message is received on the queue.

The *etype* parameter represents the event type and will have one of the following values upon return from the query:

DELIVERY
message received.

NO_MESSAGE
no message on the queue.

ERROR
queue has been closed or deleted.

END_OF_QUEUE
end of queue.

The *msgtype* was set by the user when the message was sent. *msgtype* is surfaced on the query. This value is user-specified. 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 by using the _recv or _recvlist methods.

The *header* parameter is a delivery header that returns delivery information. A value of 0 may be passed in to indicate that the delivery information should not be surfaced by the query. Otherwise, *header* is passed into the _query as an empty SCL list. If a message event is returned, *header* is updated with the delivery information as a list of named items:

DESCRIPTOR
Descriptive, user-supplied text.

RESPONSE_QUEUE_NAME
User-supplied response queue name.

QUEUED_DATETIME
Queued date/time stamp.

ORIGIN_NAME
Originator's name.

SECURITY_NAME
Security name of originator.

CORRELATOR
User-supplied correlator value.

The *attachlist* parameter is a list of attachments that have been included with the message. A value of 0 may be passed in to indicate that the attachment list should not be surfaced by the query. In this scenario, the attachment list is never surfaced and so the attachments do not have to be received and accepted.

Otherwise, *attachlist* must be passed into the _query as an empty SCL list. If a message event is returned, *attachlist* is updated only if any attachments were included with the message. Only the attachment list is surfaced by the query; the attachments have not actually been transferred. If attachments are surfaced, actions must be taken to actually receive the attachments and to indicate that the receipt is complete. See "Accepting Attachments" on page 365 for more information.

If an error or warning condition is encountered during the query, a non-zero return code is returned in the *rc* parameter. The return codes shown here are a defined set of warning or error conditions that can be checked by using the SYSRC macro, which is provided in the autocall library that is supplied by SAS Institute.

If *rc* is not one of the messages shown here, use SYSMSG() to determine the exact error message.

_SEREL
indicates that the query failed because the previous message (surfaced by previous query) has not been received. When a _query returns a message, no subsequent queries are allowed until the previous message is received by using the _recv or _recvlist method.

_SEATTAC
indicates that the query failed because the attachment transfer is not complete. If the previous message surfaced by the query had attachments, no subsequent queries will be allowed until _acceptAttachment and the COMPLETE flag are called to signal that attachment transfer is complete. The COMPLETE flag may be specified without an attachment list to signal that no attachments are to be received and attachment acceptance is complete:

```
attachlist = 0;
call send(obj, "_acceptAttachment",
          attachlist, rc, "COMPLETE");
```

_SEHEADR
indicates that an invalid delivery header list was specified. If the header parameter is non-zero, it MUST be an empty SCL list so that it can be updated by the query.

_SEATTEM
indicates that the query failed because a non-empty attachment list was passed in. If *attachlist* is non-zero, it must be an empty SCL list.

_SEACTRM
indicates that the connection was abnormally terminated. If this return code is surfaced, the queue was abnormally closed. It must be re-opened before any subsequent processing can take place on that queue instance.

If the NOTICE queue attribute is in effect, the *delivery_key* parameter is required on the query. Set the *delivery_key* to 0 and call _query 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 (including *msgtype*, *attachlist*, and *header*) is returned. In addition, *delivery_key* will be updated. This key can be used at a later time to retrieve this message from the queue. To retrieve the actual message, _query should be called again, this time specifying the *delivery_key* that was returned on the initial query.

If the queue is not operating under NOTICE mode, the *delivery_key* parameter should not be specified.

Example 1

This example queries on a Queue instance where the queue was opened in FETCH mode and the attribute POLL is set.

```
header = makelist();
attachlist = makelist();
call send(queueInst, '_query', etype, msgtype,
          header, attachlist, rc);

if (etype = "DELIVERY") then do;
   if (msgtype = 1) then do;
      /* add salary information */
   end;
end;

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

```
end;
```

Example 2

This example queries on a Queue instance where the queue was opened by using the NOTICE attribute. In this case, if the message type is 4, the application calls the query again to retrieve the actual message on the queue.

```
header = makelist();
attachlist = makelist();
key = 0;
call send(queueInst, '_query', etype, msgtype,
          header, attachlist, rc, key);

if (etype eq "DELIVERY") then do;
   if (msgtype eq 4) then do;
      rc = dellist(header,'Y');
      rc = dellist(attachlist,'Y');

      header = makelist()
      attachlist = makelist();

         /*******************************/
         /* specify the key value       */
         /* returned by the initial     */
         /* query                       */
         /*******************************/
      call send(queueInst, '_query', etype, msgtype,
                header, attachlist, rc, key);
   end;
end;
```

_recv

Receive message into SCL variables.

Syntax

CALL SEND(*queueInst*, '_recv', *rc* <, *parm1*,...,*parmn*>);

Where...	Is type...	And represents...
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 needs to be received into SCL parameters. _recv supports the receipt of numerics and characters. SCL lists are not supported by _recv; _recvlist should be called to receive SCL lists. _recv must be called with the correct parameter types. For example, if a character and a numeric variable are sent to the queue, _recv must be called with a numeric and a character variable in the correct order.

If an error or warning condition is encountered during the receive, a non-zero return code is returned in the *rc* parameter. The return codes shown here are a defined set of warning or error conditions that can be checked by using the %SYSRC macro, which is provided in the autocall library that is supplied by SAS Institute.

If the *rc* is not one of the messages shown here, use SYSMSG() to determine the exact error message.

_SWTRUNC
is a WARNING that indicates that the message has been truncated because too few parameters were passed into the _recv method. All parameters that were passed into the method are updated, but the remainder of the message is truncated.

_SENOBUF
indicates that the receive failed because there is no message to receive.

_SEMORE
indicates that the receive failed because more receive parameters (*parm1...parmn*) were passed into _recv than were actually received. Parameters are NOT updated and _recv needs to be called again to receive the message.

If an unexpected message is received, _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.

Example 1

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

```
header = makelist();
attachlist = makelist();
call send(queueInst, '_query', etype, msgtype,
          header, attachlist, rc);

if (etype = "DELIVERY") then do;

   if (msgtype = 1) then do;
      name = '';
```

```
      age = 0;
      race ='';
         /* receive 3 parameters */
      call send(queueInst, '_recv', rc,
                name, age, race);
   end;
   else if (msgtype = 5) then do;
         /* receive 1 parameter */
      task = 0;
      call send(queueInst, '_recv', rc, task);
   end;
   else do;
         /* unexpected message, force */
         /* truncation                */
      call send(queueInst,'_recv', rc);
   end;
end;
```

Example 2

This example throws the unexpected message away by forcing truncation.

```
call send(queueInst, '_recv', rc);
```

_recvlist

Receive SCL lists.

Syntax

CALL SEND(*queueInst*, '_recvlist', *rc* <, *list1*,..,.*listn*>);

Where...	Is type...	And represents...
rc	N	return code
list1...listn	L	parameters in which to receive the SCL lists; consists of 0 or more SCL lists to receive into

When a message is surfaced by a query, it needs to be received into SCL parameters. The _recvlist method supports the receipt of SCL lists only. Use the _recv method to receive the message into numeric and character variables that are not SCL lists.

If an error or warning condition is encountered during the receive, a non-zero return code is returned in the *rc* parameter. The return codes shown here are a defined set of warning or error conditions that can be checked by using the SYSRC macro, which is provided in the autocall library that is supplied by SAS Institute.

If the *rc* is not one of the messages shown here, use SYSMSG() to determine the exact error message.

_SWTRUNC
a WARNING that indicates that the message has been truncated because too few parameters were passed into the _recv method. All parameters that were passed into the method are updated, but the remainder of the message is truncated.

_SENOBUF
indicates that the receive failed because there is no message to receive.

_SEMORE
indicates that the receive failed because more receive parameters (*list1...listn*) were passed into _recvlist than were actually received. Parameters are NOT updated and _recvlist must be called again to receive the lists.

If an unexpected message is received, _recvlist 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 received and truncated.

Example 1

This example queries on a fetch queue, and then based on the message type, receives the message into the appropriate SCL variables.

```
header = makelist();
attachlist = makelist();
call send(queueInst, '_query', etype,
          msgtype, header, attachlist, rc);

if (etype = "DELIVERY") then do;
   if (msgtype = 22) then do;
         /* receive 2 SCL lists */
      namelist = makelist();
      agelist = makelist();
      call send(queueInst, '_recvlist',
                rc, namelist, agelist);
   end;
   else if (msgtype = 5) then do;
         /* receive 1 SCL list */
      userlist = makelist();
      call send(queueInst, '_recvlist', rc, userlist);
   end;
   else do;
         /* unexpected message, force */
         /* truncation                */
      call send(queueInst,'_recvlist', rc);
   end;
end;
```

Example 2

This example throws the message away by forcing truncation.

```
call send(queueInst, '_recvlist', rc);
```

_getfield

Receive one or more SCL parameters at a time.

Syntax

CALL SEND(*queueInst*, '_getfield', *status*, *rc*,
parm1 <, *parm2...parmn*>);

Where...	Is type...	And represents...
status	N	status of parameter receipt
rc	N	return code
parm1...parmn	C or N	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 SCL parameters. The _getfield method behaves like the _recv method in that it receives the message into SCL parameters. The two methods differ in that _recv requires that you receive the entire message at one time, while _getfield allows each parameter to be received separately. The _getfield method supports the receipt of numeric and character parameters, but it does not support the receipt of SCL lists. Use _getlist to receive SCL lists one at a time.

The *status* parameter has a value of 1 if this is the last parameter, indicating there are no additional parameters to retrieve. Otherwise, it has the 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 SYSMSG() to determine the exact error message.

Example

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

```
name1 = '';
name2 = '';
name3 = '';
name4 = '';
call send(queueInst, '_getfield', status, rc, name1);

if (status ne 1) and (rc eq 0) then
   call send(queueInst, '_getfield',
             status, rc, name2, name3);

if (status ne 1) and (rc eq 0) then
   call send(queueInst, '_getfield', status, rc, name4);
```

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

if (status eq 1) and (rc eq 0) then
   /* all parameters have been received    */
   /* and can be used in processing        */
```

_getlist

Receive one or more SCL lists at a time.

Syntax

CALL SEND(*queueInst*, '_getlist', *status*, *rc*,
list1 <, *list2*,...,*listn*>);

Where...	Is type...	And represents...
status	N	status of parameter receipt
rc	N	return code
list1...listn	L	one or more SCL lists to receive

When a message is surfaced by a query, it needs to be received into SCL parameters. The _getlist method behaves like the _recvlist method in that it receives the message into SCL list parameters. The two methods differ in that _recvlist requires that you receive the entire message at one time. _getlist allows each list to be received separately. The _getlist method supports the receipt of SCL lists _getfield should be used to receive non-SCL list parameters (that is numeric and character parameters).

The *status* parameter has a value of 1 if this is the last list, indicating there are no additional lists to retrieve. Otherwise, it has the 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 SYSMSG() to determine the exact error message.

Example

This example receives one SCL list, then two lists, then the last one.

```
list1 = makelist();
list2 = makelist();
list3 = makelist();
list4 = makelist();
call send(queueInst, '_getlist', status, rc, list1);
```

```
if (status ne 1) and (rc eq 0) then
   call send(queueInst, '_getlist',
             status, rc, list2, list3);

if (status ne 1) and (rc eq 0) then
   call send(queueInst, '_getlist',
             status, rc, list4);

   /*************************************/
   /* status should be set to 1 if this is */
   /* the last list to be received         */
   /*************************************/

if (status eq 1) and (rc eq 0) then
   /* all lists have been received and can */
   /* be used in processing                */
```

_acceptAttachment

Receive attachments.

Syntax

CALL SEND(*queueInst*, '_acceptAttachment', *attachlist*, *rc* <, *attribs*>);

Where...	Is type...	And represents...
attachlist	L	list of attachments to receive
rc	N	return code
attribs	C	(optional) attributes

When invoked on a Queue instance, _acceptAttachment indicates which, (if any) attachments are to be received.

When a _query surfaces a message event, it also surfaces an attachment list if an attachment list was included with the message. Only the attachment list is surfaced by the query, no attachments have actually been transferred at this point. Therefore, the user must indicate which, if any, attachments should be transferred or received by using the _acceptAttachment method. See “Accepting Attachments” on page 365 for more information.

If an attachment list is surfaced by the query, _acceptAttachment and the COMPLETE attribute *must be called at some point* before subsequent sends and/or queries will be allowed. This attribute indicates the completion of attachment receipt. Even if no attachments are to be received, _acceptAttachment must be called with the COMPLETE attribute to indicate completion.

If an error or warning condition is encountered during an attachment transfer, a non-zero return code is returned in the *rc* parameter. The return codes shown here are

a defined set of warning or error conditions that can be checked by using the SYSRC macro, which is provided in the autocall library that is supplied by SAS Institute.

If the *rc* is not the message shown here, use SYSMSG() to determine the exact error message.

_SWATTXF
is a WARNING that indicates one or more attachments were not successfully transferred. See “Attachment Error Handling” on page 370 for more information.

Example

This example accepts two attachments. The first attachment is placed in the file WORK.ABC. The second is an external file that will be placed in **/tmp/text.file**. Setting the COMPLETE flag indicates that attachment transfer will be complete after these two attachments are received.

```
alist = makelist();

att1 = makelist();
rc = setnitemn(att1, 1, "ATTACH_ID");
rc = setnitemc(att1, "WORK", "OUTLIB");
rc = setnitemc(att1, "ABC", "OUT");

att2 = makelist();
rc = setnitemn(att2, 2, "ATTACH_ID");
rc = setnitemc(att2, '/tmp/text.file', "OUTFILE");

rc = insertl(alist, att1);
rc = insertl(alist, att2);

call send(queueInst,'_acceptAttachment',
          alist, rc, "COMPLETE");
```

_getprop

Get queue properties.

Syntax

CALL SEND(*queueInst*, '_getprop', *rc, type, def, msgpsist, dlvrmode, crdt, depth, maxdepth, maxmsgl*);

Where...	Is type...	And represents...
rc	N	return code
type	C	indicates what happens to a queue after the queue is closed
def	C	defines how the queue was created

Where...	Is type...	And represents...
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 _getprop method retrieves the properties that are associated with a queue.

If an error or warning condition is encountered when retrieving the queue properties, a non-zero return code is returned in the *rc* parameter. Use SYSMSG() to determine the exact error message.

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 specifies whether the queue is defined by using predefined 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 in addition to the actual message.

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

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 held by the queue (-1 is unlimited).

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

Example

This example prints the information obtained about a queue.

```
rc = 0;
type = '';
def = '';
msgpsist = '';
dlvrmode = '';
crdt = 0;
depth = 0;
maxdepth = 0;
maxmsgl = 0;
msg = '';
datetime '';

call send(queueInst, '_getprop', rc,
          type, def, msgpsist, dlvrmode,
          crdt, depth, maxdepth, maxmsgl);
if (rc NE 0) then do;
   msg = sysmsg();
   put msg;
end;
else do;
   put 'Queue properties:';
   put 'type = ' type;
   put 'definition = ' def;
   put 'msg persistence = ' msgpsist;
   put 'delivery mode = ' dlvrmode;
   datetime = putn(crdt, 'datetime.');
   put 'creation date/time = ' datetime;
   put 'current depth = ' depth;
   put 'maximum depth = ' maxdepth;
   put 'maximum message length = ' maxmsgl;
end;
```

_setprop

Set queue properties.

Syntax

CALL SEND(*queueInst*, '_setprop', *rc, dlvrmode, maxdepth, maxmsgl*);

Where...	Is type...	And represents...
rc	N	return code
dlvrmode	C	message delivery mode
maxdepth	N	queue maximum depth allowed
maxmsgl	N	queue maximum message length allowed

The _setprop method allows you to set certain queue properties. In particular, you may set the message delivery mode if there are no 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 warning condition is encountered while setting the queue properties, a non-zero return code is returned in the *rc* parameter. Use SYSMSG() to determine the exact error message.

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

DEFAULT
A query on the queue retrieves the message header information in addition to the actual message.

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

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

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

Note: 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. △

Example

This example sets the message queue delivery mode to NOTICE, the maximum depth to 50, and maximum message length to 4K bytes.

```
rc = 0;
dlvrmode = 'notice';
maxdepth = 50;
maxmsgl = 4096;
msg = '';

call send(queueInst, '_setprop', rc, dlvrmode,
          maxdepth, maxmsgl);
if (rc NE 0) then do;
   msg = sysmsg();
   put msg;
end;
```

```
else put 'SetProp was successful';
```

_getsec

Get queue security information.

Syntax

CALL SEND(*queueInst*, '_getsec', *rc*, *acl*);

Where...	Is type...	And represents...
rc	N	return code
acl	L	access control list

The _getsec method allows you to obtain information about the permissions or privileges that are associated with a specific queue.

If an error or warning condition is encountered when retrieving security information, a non-zero return code is returned in the *rc* parameter. Use SYSMSG() to determine the exact error message.

The *acl* parameter is an SCL list that is returned from the _getsec method and contains the access control information.

Example

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

```
rc = 0;
msg = '';

acl = makelist();
call send(queueInst, '_getsec', rc, acl);
if (rc NE 0) then do;
   msg = sysmsg();
   put msg;
end;
else do;
   put 'User Access Rights:';
   n = listlen(acl);
   i = 1;
   do while (n > 0);
      userid = nameitem(acl, i);
      permission = getitemc(acl, i);
```

```
      put userid '=' permission;
      n = n - 1;
      i = i + 1;
   end;
end;

rc = dellist(acl);
```

_setsec

Set queue security information.

Syntax

CALL SEND(*queueInst*, '_setsec', *rc*, *acl*);

Where...	Is type...	And represents...
rc	N	return code
acl	L	access control list

The _setsec method allows you to specify the permissions or privileges that are associated with a specific queue.

If an error or warning condition is encountered while setting the security information, a non-zero return code is returned in the *rc* parameter. Use SYSMSG() to determine the exact error message.

The *acl* parameter is an SCL list that contains the access control information.

Example

This example sets two user privileges for a specific queue. The first user (USER1) is defined to have full privileges. Full privileges consist of the following: deliver, fetch, browse, getprop, setprop, getsec, and setsec. These privileges can be set individually or using the 'all' parameter. The second user (USER2) is defined to have only browse, getprop, and getsec privileges.

```
rc = 0;
acl = makelist();

userid = 'user1';
permission = 'd+f+b+gp+sp+gs+ss'; /* or 'all' */
rc = setnitemc(acl, permission, userid);
```

```
userid = 'user2';
permission = 'b+gp+gs';
rc = setnitemc(acl, permission, userid);

call send(queueInst, '_setsec', rc, acl);
if (rc NE 0) then do;
   msg = sysmsg();
   put msg;
end;
else 'SetSec was successful';
```

_close

Close a message queue.

Syntax

CALL SEND(*queueInst*, '_close', *rc* <, *attribs*>);

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

When invoked on a Queue instance, _close closes the queue. The Queue instance still exists, but it is no longer open. Therefore, no subsequent messaging can occur on this instance until it is opened by using the _open method.

If an error or warning condition is encountered during the close, a non-zero return code is returned in the *rc* parameter. The return codes shown here are a defined set of warning conditions that can be checked by using the SYSRC macro, which is provided in the autocall library that is supplied by SAS Institute.

If the *rc* is not one of the messages shown here, use SYSMSG() to determine the exact error message.

_SWQDMSG
is a WARNING that indicates that the queue was successfully closed, but the queue was not deleted because messages still remain on the queue.

_SWQDADM
is a WARNING that indicates that the queue was successfully closed, but the queue cannot be deleted because it is an administrator predefined queue.

The following optional *attribs* may be specified with the close method:

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

DELETE
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. Using this attribute when closing an administrator pre-defined queue returns a warning because these types of queues can only be deleted by an administrator (PROC ADMIN). This attribute is ignored when closing temporary queues because they are automatically deleted when the creating instance closes it.

DELETE_PURGE
behaves exactly like the DELETE attribute with one difference. It 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