CHAPTER

34

SAS Component Language (SCL) Interface to Direct Messaging

Introduction

In its simplest form, messaging requires that both the client and the server portions of the application be active at the same time. The client cannot send a message unless a server is listening for a message. This is referred to as *direct messaging*.

The direct-messaging facility allows basic and flexible message construction, transmission, and notification services that span operating system and hardware boundaries across the enterprise. Messages are free-form. Their structure, which is defined by the application developer, may range from a simple collection of variables to complex hierarchies of SCL lists. Additionally, messages may include one or more attachments that 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.

These applications can run on a single platform or on separate platforms of the same or of a unique type. Also, any number of applications can be simultaneously communicating using direct messaging.

SAS/CONNECT provides tools that enable application 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.

The application developer tools include a Station class and a Cnction class. Direct messaging allows messages to be sent (using SCL methods) between two or more instances of the Station class. After station initialization has occurred, the Cnction class is used to make a connection. A server station can service any number of client stations.

Station Class Overview

Access to all distributed messaging services is obtained by opening a station interface instance. After opening, the station interface instance can be used for either direct or indirect (queued) messaging.

PARENT:
SASHELP.FSP.OBJECT.CLASS

CLASS:
SASHELP.CONNECT.STATION.CLASS

Station Class Usage

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

Station Class Methods

The instance methods that are defined to the Station class are discussed in this section. The syntax for these methods requires that they be enclosed in single quotes.

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

_query
Generic query on any connection.

CAUTION:
_query is only valid for direct messaging. Indirect messaging communicates using message queues, and therefore does not require a connection. △

_close
Close a station interface instance.

Notation that is used to define the parameter type:

C	Character Type
N	Numeric Type
L	SCL List Type

Dictionary

_open

Open a station interface instance for distributed messaging services.

Syntax

CALL SEND(*stationInst*, '_open', *instanceName*, *rc*);

Where...	Is Type...	And represents...
instanceName	C	name of station instance
rc	N	return code

When invoked on a station instance, _open initializes a station interface instance and enables access to distributed messaging services. The station *instanceName* must be unique to the SAS session in which _open is invoked. The _open method must be successfully invoked before any point-to-point or queue messaging can take place.

The *instanceName* parameter represents a collection name in the queue messaging environment. A collection is a user-defined grouping of queues that is managed by the same collection manager.

Example

This example opens the station interface instance DMMAPPL.

```
stationid = loadclass('sashelp.connect.station');
stationInst = instance(stationid);
call send(stationInst, '_open', "DMMAPPL", rc);
```

_query

Generic query on any connection.

Syntax

CALL SEND(*stationInst*, '_query', *etype*, *msgtype*, *header*, *attachlist*, *cnctionInst*, *rc*);

Where...	Is type...	And represents...
etype	C	type of event received
msgtype	N	message type of received message
header	L	delivery header list
attachlist	L	attachment list
cnctionInst	N	Cnction instance on which the event was received
rc	N	return code

Used for direct messaging, the _query listens on a station instance for incoming events on any connection.

The event type *etype* is returned with one of the following values:

CONNECT
connect event received

DISCONNECT
disconnect event received

MESSAGE
message received

ABORT
abort event received.

The *msgtype* parameter can be set by the user when the message is sent and is surfaced on the query. This value is user-specified, and when surfaced by the query, the user can use the message type to determine how many and what type of parameters should be used in receiving the actual message when using the _recv and _recvlist methods.

The *header* parameter is an SCL list 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* must be passed into the _query as an empty SCL list. If a message event is returned, *header* is updated with the delivery information.

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. If the value is not 0, *attachlist* must be an empty SCL list. If a message event is returned, *attachlist* is updated only if attachments were received along with the message. If attachments are surfaced, actions must be taken to receive the attachments and to indicate that the receipt is complete. See “Sending Attachments” on page 358 for more information about the receiving of attachments.

The *cnctionInst* parameter is set upon return. This indicates the Cnction instance on which the event was received.

- If a CONNECT event is received, a successfully opened instance of the Cnction class is returned in *cnctionInst*. This Cnction instance can now be used to send, receive or query on the connection.
- If a MESSAGE event is received, *cnctionInst* will be the Cnction instance on which the message was received.
- If a DISCONNECT or ABORT event is received, *cnctionInst* will be the Cnction instance on which the disconnect or abort was received.

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

If the parameter *rc* is not one of the parameters that are listed below, use SYSMSG() to determine the exact error message.

_SEREL
This return code 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 will be allowed until the previous message is received using the _recv or _recvlist method.

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

```
call send(cnctionInst, "_acceptAttachment",
          0, rc, "COMPLETE");
```

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

_SEHEADR
This return code indicates that an invalid delivery header list was specified. If non-zero, the header must be an empty SCL list so that it can be updated by the query.

Example

This example queries on the station instance by listening for messages on any connections.

```
   /***********************************/
   /* create empty attachment list to   */
   /* pass into query                   */
   /***********************************/
attachlist = makelist();
call send(stationInst, '_query', etype,
          msgtype, attachlist, cnctionInst, rc);

if (etype = "CONNECT") then do;
      /*********************************/
      /* send message back on new      */
      /* connection instance           */
      /*********************************/
   msgtype = 5;
   call send(cnctionInst, '_send',
             msgtype, 0, 0, rc, "Return string");
end;

else if (etype = "DISCONNECT") then
end;

else if (etype = "MESSAGE") then do;
      /*********************************/
      /* will have meaning to some user */
      /*********************************/
   if (msgtype = 1) then
      ...more data lines...
```

```
end;
```

_close

Close a station interface instance.

Syntax

CALL SEND(*stationInst*, '_close', *rc*);

Where...	Is type...	And represents...
rc	N	return code

When the _close method is invoked, the station interface instance is closed. All active Cnction instances on the station (meaning any connections not previously disconnected) will be disconnected at _close time.

After the _close has executed, the station object still exists but is no longer open. Therefore, no subsequent distributed messaging services can be executed until the station object is re-opened by using an _open call.

Example

This example closes a station interface instance.

```
call send(stationInst, '_close', rc);
```

Cnction Class Overview

Direct-messaging services are provided by the Cnction class. The instance methods for the Cnction class, which is defined below, enable end-user applications to implement client/server applications through the provision of a stateless, message-passing facility.

PARENT:
SASHELP.FSP.OBJECT.CLASS

CLASS:
SASHELP.CONNECT.CNCTION.CLASS

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

Cnction Class Methods

The instance methods defined to the Cnction class are discussed in this section

_open
Bind to a same-named interface instance in another SAS session.

_send
Message exchange between two bound session instances.

_sendlist
Message exchange of SCL lists between two bound session instances.

_query
Query on a specific connection.

_recv
Receive message into SCL variables.

_recvlist
Receive SCL lists.

_getfield
Receive one or more parameters at a time.

_getlist
Receive one or more parameters at a time.

_acceptAttachment
Receives attachments.

_getConnectInfo
Get connection information.

_disconnect
Sever a session instance binding.

Here is the notation that is used to define the parameter types:

C	Character Type
N	Numeric Type
L	SCL List Type

Dictionary

_open

Bind to a same-named interface instance in another SAS session.

Syntax

CALL SEND(*cnctionInst*, '_open', *stationInst*,
connection_name, *rc*);

Where...	Is type...	And represents...
stationInst	N	successfully opened station instance
connection_name	C	connection name (destination)
rc	N	return code

When invoked on a Cnction instance, _open creates a duplex message channel between the bound session instances, which enables point-to-point messaging. The user is responsible for obtaining the instance of the Cnction class, and then sending the _open method to it. After opening, the active connection can send and receive stateless, free-form messages.

A station interface instance should have already been opened, and should be passed in as the *stationInst*. In order for the Cnction _open to be successful, the station instance names that are used to create the station instance must be the same in both SAS sessions. If the station names do not match, the Cnction _open fails.

The *connection_name* designates the destination; it may be network qualified or moniker based. The supported syntax is

```
[protocol]//[network node]
   /[network resource identifier]
```

or

```
[domain]:[moniker]
```

Note: Not all qualifiers are required. △

For example, the specification: **tcp//host1/shr1** defines the communications access method as TCP/IP, the host node as HOST1 and the service as SHR1. If the protocol qualifier is omitted, the default COMAMID setting is used. If the network node qualifier is omitted, the binding is LOCAL.

The *rc* return code will be non-zero if an error is encountered during the open. Use SYSMSG() to determine the exact error message.

Example

This example binds to the same-named station interface (DMMAPPL) in another SAS session, designated by the service "/SHR1". No network node is provided on the connection name, so the binding is LOCAL, that is, both SAS sessions are on the same host. In addition, no protocol qualifier was provided on the connection name, so the default COMAMID setting is used.

```
   /* create and open station instance */
stationid = loadclass('sashelp.connect.station');
stationObj = instance(stationid);
call send(stationObj, '_open', "DMMAPPL", rc);

   /* open connection */
cnctionid = loadclass('sashelp.connect.cnction');
cnctionInst = instance(cnctionid);
call send(cnctionInst, '_open', stationObj,
          "/shr1", rc);
```

_send

Message exchange between two bound session instances.

Syntax

CALL SEND(*cnctionInst*, '_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)
*attachlis*t	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 between two bound session instances. 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 and will be 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 using the _recv method.

The delivery *header* parameter is an SCL list that represents delivery information that is to be included. This information is surfaced on the query so it may be viewed by the receiver. If there is no information to include, *header* may be set to 0. Otherwise, *header* should be a valid SCL list, which consists of supported named items that are used to relay delivery information.

The *attachlist* parameter is an SCL list that represents 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 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. See “Sending Attachments” on page 358 for further details about the specific syntax for *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 listed are a defined set of warning and 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 parameters that are defined below, use SYSMSG() to determine the exact error message.

_SEREL
This return code indicates that the send failed because the previous message (surfaced by a query) has not been received. When _query returns a message, no sends are allowed on that connection until the previous message is received using the _recv or _recvlist method.

_SEATTAC
This return code indicates that the send failed because the attachment transfer is not complete. If the previous message surfaced by the query had attachments, no sends will be allowed on the connection until the _acceptAttachment method is called by using the COMPLETE flag. This signals 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 that attachment acceptance is complete:

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

_SEDISCP
This return code indicates that the send failed because a DISCONNECT is pending on this connection. If this return code is surfaced, _query should be called for this connection so that the disconnect can be received and the resources cleaned up.

_SWATTXF
This return code is a WARNING that indicates that the message was sent successfully, but one or more errors were encountered during attachment transfer. See “Attachment Error Handling” on page 370 for more details.

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 Cnction instance to send 5 parameters with no attachments.

```
name = "John Doe"
age = 35;
company = "SAS";
code = 13484;
type = 472;
msgtype = 11;
header = 0;
attachlist = 0;

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

_sendlist

Message exchange between two bound session instances.

Syntax

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

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
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 between two bound session instances. Any type of SCL list is supported (that is, 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 using the _recvlist method.

The delivery *header* parameter is an SCL list that represents delivery information that is to be included. This information is surfaced on the query so it may be viewed by the receiver. If there is no information to include, *header* may be set to 0. Otherwise, *header* should be a valid SCL list, which consists of supported named items used to relay delivery information.

The *attachlist* parameter is an SCL list that represents 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 identified by *attachlist*. 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. See “Sending Attachments” on page 358 for more details about the specific syntax for 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 are listed are a defined set of warning and 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 return codes that are listed below, use SYSMSG() to determine the exact error message.

_SEREL
This return code indicates that the send failed because the previous message (surfaced by a query) has not been received. When _query returns a message, no sends are allowed on that connection until the previous message is received by using the _recv or _recvlist method.

_SEATTAC
This return code indicates that the send failed because the attachment transfer is not complete. If the previous message surfaced by the query had attachments, no sends

will be allowed on the connection until the _acceptAttachment method is called with the COMPLETE flag. This signals 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 that attachment acceptance is complete:

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

_SEDISCP
This return code indicates that the send failed because a DISCONNECT is pending on this connection. If this return code is surfaced, _query should be called for this connection so that the disconnect can be received and the resources cleaned up.

_SWATTXF
This return code is a WARNING that indicates that the message was sent successfully but one or more errors were encountered during attachment transfer. See “Attachment Error Handling” on page 370 for more details.

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

Example

This example invokes the _sendlist method on the Cnction instance to send two SCL lists that have 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);
```

```
   /***********************************/
   /* set message type so that the      */
   /* receiving side knows how many     */
   /* lists are to be received          */
   /***********************************/
msgtype =22;
header = 0;
attachlist = 0;

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

_query

Query on a specific connection.

Syntax

CALL SEND(*cnctionInst*, '_query', *etype*, *msgtype*, *header*, *attachlist*, *rc*);

Where...	Is type...	And represents...
etype	C	type of event received
msgtype	N	message type of received message
header	L	delivery header list
attachlist	L	attachment list
rc	N	return code

When _query is invoked on a Cnction instance, only events received on this specific connection are returned. This is different behavior than the _query method that is invoked on a station instance, which listens for events on all connections.

etype will have one of the following values when returned from the query:

DISCONNECT
Disconnect event received on this connection. The *cnctionInst* must be opened again (_open) before any subsequent sends, receives, or queries will be allowed.

MESSAGE
Message received.

ABORT
Abort event received.

The *msgtype* parameter is set by the user when the message is sent and 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 when using the _recv or _recvlist methods.

The *header* parameter is an SCL list that returns delivery information. A header value of 0 indicates that the delivery information should not be surfaced by the QUERY. Otherwise, *header* must be an empty SCL list. If a message event is returned, *header* is updated with the delivery information.

The *attachlist* parameter is a list of attachments that have been included with the message. An *attachlist* value of 0 indicates 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. If non-zero, *attachlist* must be an empty SCL list. If a message event is returned, *attachlist* will be updated only if any attachments were received along with the message. If attachments are surfaced, actions must be taken to receive the attachments and to indicate that the receipt is complete. See “Sending Attachments” on page 358 for more information about the receiving of attachments.

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 that are listed are a defined set of warning and 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 return codes that are listed here, use SYSMSG() to determine the exact error message.

_SEREL
This return code 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 will be allowed until the previous message is received using the _recv or _recvlist method.

_SEATTAC
This return code 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 the _acceptAttachment method is called by using the COMPLETE flag. This signals that the attachment transfer is complete. The COMPLETE flag may be specified without an attachment list to signal that no attachments are to be received and that attachment acceptance is complete.

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

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

_SEHEADR
This return code indicates that an invalid delivery *header* list was specified. If non-zero, *header* must be an empty SCL list so that it can be updated by the query.

Example

This example queries on the Cnction instance, listening for messages on *this specific connection only*.

```
   /* create empty attachment list to */
attachlist = makelist();

   /* create empty header list */
header = makelist();

   /* query on this specific connection */
call send(cnctionInst, '_query', etype,
         msgtype, header, attachlist, rc);

if (etype = "DISCONNECT") then do;
end;

else if (etype = "MESSAGE") then do;

      /* if message type is one,        */
      /* application realizes that only */
      /* one list needs to be received  */
   if (msgtype eq 1) then do;
      list1 = makelist();
      call send(cnctionInst, '_recvlist',
                rc, list1);
   end;

      /* if message type is two,        */
      /* application realizes that name */
      /* and age must be received       */
   else if (msgtype eq 2) then do;
      name = ''
      age = 0;
      call send(cnctionInst, '_recv',
                rc, name, age);
      end;
end;
```

_recv

Receive message into SCL variables.

Syntax

CALL SEND(*cnctionInst*, '_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. The _recv method supports the receipt of numerics and characters. SCL lists are not supported by _recv; _recvlist should be called to receive SCL lists. The _recv method must be called with the correct parameter types. For example, if a character and a numeric variable were sent, _recv must be called with a character and a numeric 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 that follow are a defined set of warning and 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 will be 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 will NOT be updated and _recv must be called again to receive the message.

If an unexpected message is received, _recv can be called by using 0 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 specific connection, and based on the msgtype that is returned, receives the message into the appropriate SCL variables.

```
attachlist = makelist();
header = makelist();

call send(cnctionInst, '_query', etype,
          msgtype, header, attachlist,
          rc);

if (etype eq "MESSAGE") then do;
   if (msgtype eq 1) then do;
      name = '';
```

```
      age = 0;
      race = '';
         /*****************************/
         /* receive 3 parameters.     */
         /*****************************/
      call send(cnctionInst, '_recv',
                rc, name, age, race);
   end;

   else if (msgtype eq 2) then do;
      company = ''; address='';
         /*****************************/
         /* receive 2 character       */
         /* parameters.               */
         /*****************************/
      call send(cnctionInst, '_recv',
              rc, company, address);
   end;

   else do;
         /*****************************/
         /* unknown message type; throw */
         /* away message by forcing     */
         /* truncation.               */
         /*****************************/
      call send(cnctionInst, '_recv', rc);
   end;
end;
```

Example 2

This example throws the message away by forcing truncation.

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

_recvlist

Receive SCL lists.

Syntax

CALL SEND(*cnctionInst*, '_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 must be received into SCL parameters. _recvlist 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 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.

_SWTRUNC
is a WARNING that indicates the message has been truncated because too few parameters were passed into the _recvlist method. All parameters that were passed into the method will be 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 will NOT be updated and _recvlist must be called again to receive the lists.

If an unexpected message is received, _recvlist can be called by using 0 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 receives two SCL lists.

```
attachlist = makelist();
header = makelist();

call send(cnctionInst, '_query', etype,
          msgtype, header, attachlist,
          rc);

if (etype eq "MESSAGE") then do;
      /*********************************/
      /* if message type is one, must   */
      /* receive two lists              */
      /*********************************/
   if (msgtype eq 1) then do;
      namelist = makelist();
```

```
      agelist = makelist();
      call send(cnctionInst, '_recvlist',
                rc, namelist, agelist);
   end;

      /*********************************/
      /* if message type is two, must   */
      /* receive one list               */
      /*********************************/
   else if (msgtype eq 2) then do;
      reports= makelist();
      call send(cnctionInst, '_recvlist',
                rc, reports);
   end;

   else do;
         /*******************************/
         /* unexpected message;         */
         /* throw away message by       */
         /* forcing truncation          */
         /*******************************/
      call send(cnctionInst, '_recvlist',
               rc);
   end;
end;
```

Example 2

This example throws the unexpected message away by forcing truncation.

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

_getfield

Receive one or more parameters at a time.

Syntax

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

Where...	Is type...	And represents...
status	N	status of parameter receipt
rc	N	return code
parm1...parmn	N or C	parameters in which to receive the message; consists of 0 or more numeric or character variables

When a message is surfaced by a query, it must 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 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 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(cnctionInst, '_getfield',
          status, rc, name1);

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

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

   /*************************************/
   /* If this is the last parameter to be  */
   /* received, status should have a value */
   /* of 1.                               */
   /*************************************/

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 parameters at a time.

Syntax

CALL SEND(*cnctionInst*, '_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 must be received into SCL list 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, while _getlist allows each list to be received separately. The _getlist method supports the receipt of SCL lists. Use _getfield 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. This indicates 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(cnctionInst, '_getlist',
          status, rc, list1);

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

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

   /*************************************/
```

```
   /* status should have a value of 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

Receives attachments.

Syntax

CALL SEND(*cnctionInst*, '_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 Cnction 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 using the _acceptAttachment method. See “Accepting Attachments” on page 365 for more information.

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

If an error or warning condition is encountered during attachment transfer, a non-zero return code is returned in the *rc* parameter. The return codes that follow are a defined set of warning and 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 different from the message shown here, use SYSMSG() to determine the exact error message.

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

Example 1

This example accepts two attachments. Setting the COMPLETE flag indicates that attachment transfer will be complete after these 2 attachments are received.

```
alist = makelist();

   /************************************/
   /* attachment one will be placed    */
   /* in the file WORK.ABC             */
   /************************************/
att1 = makelist();
rc = setnitemn(att1, 1, "ATTACH_ID");
rc = setnitemc(att1, "WORK", "OUTLIB");
rc = setnitemc(att1, "ABC", "OUT");

   /************************************/
   /* attachment two is an external    */
   /* file and will be placed in       */
   /* /tmp/text.file                   */
   /************************************/
att2 = makelist();
rc = setnitemn(att2, 2, "ATTACH_ID");
rc = setnitemc(att2, '/tmp/text.file',
               "OUTFILE");
rc = insert1(alist, att1);
rc = insert1(alist, att2);
call send(cnctionInst,'_acceptAttachment',
          alist, rc, "COMPLETE");
```

Example 2

In this example, no attachments will be received; however, _acceptAttachment using the COMPLETE attribute must be called to indicate completion.

```
call send(cnctionInst,'_acceptAttachment',
          0, rc, "COMPLETE");
```

_getConnectInfo

Get connection information.

Syntax

CALL SEND(*cnctionInst*, '_getConnectInfo',

connect_name, *security_id*, *rc*);

Where...	Is type...	And represents...
connect_name	C	connection name
security_id	C	security id
rc	N	return code

After a valid connection exists, _getConnectInfo may be called to obtain the connection name and security name that is associated with this particular Cnction instance.

connect_name
parameter is the partner's connection name.

security_id
parameter is the partner's security id. This parameter is returned only if this information was given by the partner.

If an error is encountered during the open the *rc* return code will be non-zero. Use SYSMSG() to determine the exact error message.

Example

This example obtains the connection information.

```
connect_name='';
security_id='';
call send(cnctionInst, '_getConnectInfo',
          connect_name, security_id, rc);
```

_disconnect

Sever a session instance binding.

Syntax

CALL SEND(*cnctionInst*, '_disconnect', *rc*);

Where...	Is type...	And represents...
rc	N	return code

When the _disconnect method is invoked, the connection is severed. The Cnction instance still exists, but is no longer bound to a same-named interface. Therefore, no

subsequent distributed messaging services can be executed until the Cnction instance is re-opened by using an _open call.

If an error is encountered during the disconnect, *rc* will be non-zero. Use SYSMSG() to determine the exact error message.

Example

This example disconnects.

```
call send(cnctionInst, '_disconnect', rc);
```

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