The SAS Component Language Debugger (SCL Debugger) is a powerful window-oriented utility that can interactively monitor the execution of SCL programs, enabling you to locate run-time errors. The SCL debugger also enables you to suspend the execution of one program that is part of a series of programs and to execute the other programs in the series. The SCL debugger interface consists of two windows, the debugger SOURCE window and the debugger MESSAGE window. The debugger displays the source program, specifying which line is executing, in the SOURCE window. The MESSAGE window contains the debugger command line, as well as the results of any debugger commands.

The debugger can

- suspend execution at selected statements and programs. The point at which execution is suspended is called a breakpoint. Breakpoints can be set based on the evaluation of an expression.
- monitor the values of selected variables. This is called a watch variable. When a watch variable is set to some specified value, the debugger stops executing the program at the statement where this occurred.
- set or query the value of SCL variables.
- display the attributes of variables.
- bypass a group of statements.
- continue execution of a halted program.
Establishing the Debugging Environment

Before you can use the SCL debugger, you must compile your SCL programs with the DEBUG option. This directs the SCL compiler to collect information from the programs that will be used in the debugging session.

You can activate the debug option in the following ways:

- Issue the DEBUG ON command from the SOURCE or DISPLAY window of an open application. After you issue this command, it remains active in all currently running tasks, and all subsequent compiles will collect debugging information.
- Specify the DEBUG option in the COMPILE statement of the BUILD procedure:

  ```sas
  proc build c=libref.catalog.entry.type;
    compile debug;
  run;
  ```

- Specify the DEBUG option in the FSEDIT procedure:

  ```sas
  proc fsedit data=data-set
    screen=screen-entry debug;
  run;
  ```

- Select Build  Debug  Debug On

- In the Explorer window, select from the pull-down menu Tools  Options  Build  Debugger

  Change Debugger status to On or Off.

Note: If you specify the DEBUG option from a procedure statement, the option is active only for that procedure or task.

In the BUILD procedure, using the DEBUG option in the COMPILE statement compiles all the specified programs with the DEBUG option. For example, the following statements compile all the FRAME entries in the catalog with the DEBUG option turned on:

```sas
proc build c=mylib.mycat;
  compile debug entrytype=frame;
```
run;

By compiling specific SCL entries with the DEBUG option, you can debug pieces of a larger application.

The debugger session terminates and exits the current stream when you issue the QUIT command in the MESSAGE window, or when you end the procedure session. If the current stream is nested, then the last window in the previous stream will be activated. Otherwise, control returns to the point at which the application was started. A stream contains information about the entries and windows that are used in the application.

Because compiling with the DEBUG ON option results in a larger code size, you should subsequently recompile your programs with the DEBUG OFF option before installing your application in production mode.

---

**Invoking the Debugger**

After you have successfully compiled a program with the DEBUG ON option, you can choose to invoke the debugger for that entry. For an entry that can be executed with the TESTAF, AF, or AFA command, you can invoke the debugger in the following ways:

- Enter the TESTAF command in the application window.
- If the debugging environment has not been established through the DEBUG ON command, specify DEBUG=YES in the AF or AFA command:
  ```
  afa c=libref.catalog.entry.type debug=yes
  ```
  Otherwise, submit the AF or AFA command.
- Specify the DEBUG option in conjunction with the TESTAF option in a PROC BUILD statement:
  ```
  proc build c=libref.catalog.entry.type testaf debug;
  ```
- Select [Build](#) → [Test](#)

For an FSEDIT or FSVIEW application, you can invoke the debugger in the following ways:

- Close the SOURCE window, which compiles the program, when you build the application.
- Specify the DEBUG option in the procedure statement:
  ```
  proc fsedit data=SAS-table
      screen=libref.catalog.entry.SCREEN debug;
  run;
  ```

  ```
  proc fsvIEW data=SAS-table
      formula=libref.catalog.entry.FORMULA debug;
  run;
  ```
Using the Debugger Windows

When a debugging session starts, the debugger SOURCE window opens above the debugger MESSAGE window. The debugger SOURCE window displays the text of the current SCL program. The debugger MESSAGE window echoes the commands that you enter from the debugger command prompt, **DEBUG>**.

You can enter debugger commands from the following locations:

- the debugger command prompt **DEBUG>**
- the main SAS command line, if the command menus are active
- the command line at the top of the debugger MESSAGE window, if active.

However, each debugger command that you enter from the command line must be preceded with the word SCL, as in this example, which sets a breakpoint at line 10:

```
scl b 10
```

When you enter a debugger command, the SCL debugger

1. echoes the command in the debugger MESSAGE window
2. checks the syntax of the command and the parameters that you entered. The debugger returns error messages for any syntax errors and reports the positions of the errors.
3. prints the results of the command in the debugger MESSAGE window if there are no errors.

Retrieving Previously Entered Commands

If the debugger detects an error in your command, you can recall the previous command, fix the error, and press the ENTER key to re-execute the command.

There are two ways to retrieve commands that you previously entered:

- Use the ? command, which enables you to retrieve up to the last five commands. This feature recalls a command once after you press ENTER and does not cycle through the commands again.
- Define a function key (using the KEYS command) to issue the AGAIN command. Once the key is defined, position the cursor on a line in the debugger MESSAGE window and press the key that is defined as the AGAIN command. The text on that line is displayed on the debugger command line. You can re-edit the line and then re-execute the command or commands.

Using the SAS Macro Interface

The SCL debugger has a complete interface with the SAS macro facility. You can display macros with the MACEXPAND command. In addition, you can define a macro in the debugger session to replace a debugger command list that you use frequently, as in this example:

```
DEBUG> %macro ckvars; e var1 var2; %mend ckvars;
```

After the macro CKVARS is defined, you can invoke the macro as follows:

```
DEBUG> %ckvars
```
The macro CKVARS expands to
\texttt{e \ var1 \ var2}

### Debugger Commands by Functional Category

The following sections briefly describe the commands that are available in the SCL debugger. For detailed information about a command, see its corresponding dictionary entry later in this chapter.

#### Controlling Program Execution

While you are in a debugger session, you can use the following commands to monitor the flow of the program and even to change the way the program executes:

- **GO**
  - continues program execution until a specified statement or until the next breakpoint (if any) or program is finished.

- **JUMP**
  - restarts program execution at a specified executable statement. This command causes the interpreter to bypass execution of any intermediate statement.

- **STEP**
  - steps through the program statement by statement. By default, the ENTER key is set to STEP.

#### Manipulating Debugging Requests

The following debugger commands enable you to set breakpoints, tracepoints, and watched variables, which you can then use to suspend or trace the execution so that you can further manipulate the program variables. Whenever a debugging request is set, it remains in effect until you use the DELETE command to delete it.

- **BREAK**
  - sets a breakpoint at a particular executable program statement. When a breakpoint is encountered, execution of the program is suspended and the cursor is placed on the DEBUG prompt line in the debugger MESSAGE window.

- **DELETE**
  - removes breakpoints, tracepoints, or watched variables that were previously set by the BREAK, TRACE, and WATCH commands.

- **LIST**
  - displays all the breakpoints, tracepoints, and watched variables that have been set by the BREAK, TRACE, and WATCH commands.

- **TRACE**
  - sets a tracepoint. When a tracepoint is encountered, the debugger prints the information in the debugger MESSAGE window and continues processing.
WATCH

sets a watched variable. If the value of the watched variable is modified by the program, the debugger suspends execution at the statement where the change occurred, and it prints the old and the new values of the variable in the debugger MESSAGE window. This command is especially useful in large programs to detect when the value of a particular variable is being modified.

Manipulating Variables

When program execution is suspended, the debugger allows you to examine the values and attributes of variables. If the value of a variable would result in a logic error, you can then modify it so that you can continue the debugging session. Use these commands to manipulate SCL variables from the debugger:

ARGS

displays the values of arguments that are passed into the current program through the ENTRY statement.

CALCULATE

acts as an online calculator by evaluating expressions and displaying the result. This is useful when you try to set the value of a variable to the result of an expression. You can use SCL functions and dot notation in CALCULATE expressions.

DESCRIBE

displays the name, type, length, and class attributes of a variable.

EXAMINE

displays the values of one or more variables. You can use SCL functions with EXAMINE. You can also use dot notation to display values of object attributes and values that are returned by methods.

PARM

displays the values of parameters that you are passing if the next executable statement contains a function call.

PUTLIST

displays the contents of an SCL list.

SET

changes the value of a variable in the SCL program. This enables you to continue the debugging session instead of having to stop, modify the source, and recompile the program. You can also assign new values to variables in other active entries.

Expanding Macros and Macro Variable References

When program execution is suspended, the debugger can expand macros and macro variable references.

MACEXPAND

displays expanded macro invocations and macro variable references. The text of the macro or the value of the macro variable is displayed in the debugger window.
Controlling the Windows

The following commands manipulate the debugger windows:

ENVIRONMENT
   enables you to set a developer environment by redisplaying the source of any
   program in the execution stack. When a developer environment is set, the
   debugger generates messages that show you what the current program
   environment and the developer environment are. You can then scroll through the
   source program, set debugging requests, and operate on the variables.

HELP
   displays information about debugger commands.

QUIT
   terminates a debugger session.

SWAP
   switches control between the debugger SOURCE and MESSAGE windows.

TRACEBACK
   displays the execution stack, which contains information about which entries are
   running.

Customizing the Debugger Session

The following commands enable you to customize your debugging sessions:

ENTER
   enables you to assign one or more frequently used commands to the ENTER key.
   The default for ENTER is STEP, which steps through the program statement by
   statement.

IF
   enables you to conditionally execute other commands.

Dictionary

ARGS

Displays the values of arguments declared in the current program's ENTRY statement

Syntax

ARGS
Details
The ARGS command displays the values of the arguments received from a calling program and declared in the current program's ENTRY statement. This command is valid only when the current program contains an ENTRY statement. If you use this command when you are debugging an entry that does not contain an ENTRY statement, an error message is displayed. For more information about the ENTRY statement, see “ENTRY” on page 361.

Example
Suppose that the program being examined begins with the following statement:

```
entry def8;
```

In a particular program, the ARGS command might produce the following output for the above ENTRY statement:

```
args
Arguments passed to ENTRY:
1 D = 10
2 E = 4
3 F = 6
```

See Also
“DESCRIBE” on page 196
“EXAMINE” on page 200
“PARM” on page 208
“PUTLIST” on page 209

---

**BREAK**

Suspends program execution at an executable statement

Abbreviation:  B

**Syntax**

```
BREAK <location <AFTER count> <WHEN clause| DO list>>
```

**location**

specifies where to set a breakpoint (the current line, by default):

```
_ALL_
```

sets a breakpoint at every executable statement.
ENTRY

sets a breakpoint at the first executable statement in all entries in the application catalog that contain a program.

entry-name

specifies a catalog entry. A breakpoint is set at the first executable statement in the program in the specified entry. If the entry resides in the current catalog, then entry-name can be a one-level name. If the entry resides in a different catalog, then entry-name must be a four-level name, and the entry must already be loaded into the application’s execution stack. A backslash must follow the entry name.

label

specifies a program label. A breakpoint is set at the first executable statement in the program label.

line-num

specifies a line number in an SCL program where a breakpoint is set. The specified line must contain at least one executable SCL statement.

AFTER count

specifies the number of times for the debugger to execute a statement before executing the BREAK command.

   Note: When multiple statements appear on a single line, the debugger treats them as separate statements. That is, the debugger will break on the same line as each statement on that line is executed. In the following example, the line will break three times in line number 10 because the condition is met three times

   10 x=1 y=2 z=3
   b10 after 3;

WHEN clause

specifies an expression that must be true in order for the command to be executed.

DO list

specifies one or more debugger commands to execute. Use semicolons to separate multiple commands.

Details

The BREAK command sets a breakpoint at a specified statement. A breakpoint is an executable SCL program statement at which the debugger suspends program execution. An exclamation mark replaces the line number in the debugger SOURCE window to designate the line at which the breakpoint is established.

When an SCL program detects a breakpoint, it

- suspends program execution
- checks the count that you specified with the AFTER command and resumes program execution if the statement has not yet executed the specified number of times
- evaluates the condition specified with the WHEN clause and resumes execution if the condition evaluates to FALSE
- displays the entry name and line number at which execution is suspended
- executes any command that is specified in a DO list
returns control to you.

If a breakpoint is set at a program line that contains more than one statement, then
the breakpoint applies to each statement on the source line. If a breakpoint is set at a
line that contains a SAS macro expansion, then the debugger breaks at each statement
that is generated by the macro expansion.

Examples

- Set a breakpoint at line 5 in the current program:
  DEBUG> b 5
  The output to the debugger MESSAGE window is
  stop at line 5 in MYLIB.MYCAT.TEST.SCL
  b 5
  Stop at line 5 in MYLIB.MYCAT.TEST.SCL
  Set breakpoint at line 5 in program
  MYLIB.MYCAT.TEST.SCL

- Set a breakpoint in each executable statement:
  DEBUG> b _all_

- Set a breakpoint in each executable line and print all the values:
  DEBUG> b _all_ do; E _all_; end;

- Set a breakpoint at the first executable statement in each entry that contains a
  program in the catalog:
  DEBUG> b entry

- Set a breakpoint at the first executable statement in the MAIN section:
  DEBUG> b main

- Set a breakpoint at the first executable statement in the entry TEST1.SCL:
  DEBUG> b test1\n
- Set a breakpoint at line 45 in the entry TEST1.SCL:
  DEBUG> b test1\45

- Set a breakpoint at the MAIN label in the entry TEST1.SCL:
  DEBUG> b test1\main

- Set a breakpoint at line 45 before the fourth execution of line 45:
  DEBUG> b 45 after 3

- Set a breakpoint at line 45 in the entry TEST1.SCL only when both the divisor
  and the dividend are 0:
  DEBUG> b test1\45 when (divisor=0 AND dividend=0)

- Set a breakpoint at line 45 only when both the divisor and dividend are 0 before
  the fourth execution of line 45:
  DEBUG> b 45 after 3 when (divisor=0 AND dividend=0)

- Set a breakpoint at line 12 when the value of the maxLength attribute on object1
  is greater than 12:
  DEBUG> b 12 when (object1.maxLength > 12)
Set a breakpoint at line 45 of the program and examine the values of variables NAME and AGE:

```
DEBUG> b 45 do; e name age; end;
```

See Also

“DELETE” on page 194
“EXAMINE” on page 200
“LIST” on page 205
“TRACE” on page 214
“WATCH” on page 217

---

**CALCULATE**

Evaluates a debugger expression and displays the result

**Abbreviation:** CALC

**Syntax**

```
CALCULATE expression
```

**expression**

is any standard debugger expression, which can include SCL functions and dot notation.

**Details**

The CALCULATE command is an online calculator that evaluates expressions for the debugger. Expressions can include standard debugger expressions, SCL functions, and many of the SAS arithmetic functions. You can also use dot notation to perform operations on values that are returned by object attributes and methods.

**Examples**

- Add 1.1, 1.2, 3.4 and multiply the result by 0.5:

  ```
  DEBUG> calc (1.1+1.2+3.4)*0.5
  The output to the debugger MESSAGE window is
  calc (1.1+1.2+3.4)*0.5
  2.85
  ```

- Calculate the values of the variable SALE minus the variable DOWNPAY and then multiply the result by the value of the variable RATE. Divide that value by 12 and add 50:

  ```
  DEBUG> calc (((sale-downpay)*rate)/12)+50
  ```
Calculate the sum of the values of array variables A(1), A(2), and A(3):

\[
\text{DEBUG} > \text{calc sum}(a(1), a(2), a(3))
\]

Concatenate the string, the value of variable X, and the value returned from `getMaxValue` method on object1:

\[
\text{DEBUG} > \text{calc} \ 'Values='||x||\text{object1.getMaxValue()}
\]

---

**DELETE**

Deletes breakpoints, tracepoints, or watched variables

*Abbreviation:* D

**Syntax**

```plaintext
DELETE debug-request <location>
```

- **debug-request**
  - is an SCL debugger command to be deleted:
    - BREAK deletes breakpoints.
    - TRACE deletes tracepoints.
    - WATCH deletes watched variables.

- **location**
  - specifies where a debugging request should be deleted. For `debug-request` BREAK or TRACE, location can be
  - _ALL_ deletes debugging requests from all programs that are in the application’s execution stack.
  - ENTRY deletes debugging requests from the first executable statement in each entry that contains an SCL program. If the entry resides in the current catalog, then entry-name can be a one-level name.
  - entry-name\ specifies a catalog entry. The debugging requests on the first executable statement of the specified catalog entry are deleted. If the entry resides in a different catalog, then entry-name must be a four-level name, and it must already be loaded into the application’s execution stack. A backslash must follow the entry name.
  - label specifies a program label. The debugging requests on the first executable statement of the specified program label are deleted. Label can be any program label.
  - line-num specifies a line number in an SCL program. The debugging requests on the specified line are deleted.
For debug-request WATCH, location can be

_ALL_

deletes the watch status for all watched variables.

<entry-name> variable

deletes the watch status from the first executable statement of the specified catalog entry. If the entry resides in a different catalog, then entry-name must be a four-level name, and it must already be loaded into the application's execution stack. A backslash must follow the entry name. Variable specifies the name of a particular watched variable for which the watch status is deleted.

Details

The DELETE command deletes any breakpoint, tracepoint, or watched variable debugger requests in one or more programs that you specify.

Examples

- Delete all breakpoints from the entry TEST1.SCL:
  
  DEBUG> d b test1

  The output to the debugger MESSAGE window is

  d b test1
  Stop at line 5 in MYLIB.MYCAT.TEST1.SCL
  Delete all the breakpoints in MYLIB.MYCAT.TEST1.SCL

- Delete all breakpoints from the first executable statement of all entries:

  DEBUG> d b entry

- Delete the tracepoint at line 35 in the program currently executing:

  DEBUG> d t 35

- Delete the tracepoint at the first executable statement of the MAIN section of the program that is currently executing:

  DEBUG> d t main

- Delete the watch status from all variables in all programs that are in the application's execution stack:

  DEBUG> d w _all_

- Delete the watch status from the variable ABC in the program that is currently executing:

  DEBUG> d w abc

- Delete the watch status from the variable XYZ in the entry TEST3.SCL:

  DEBUG> d w test3\xyz
DESCRIBE

Displays the attributes of a variable

Abbreviation: DES

Syntax

DESCRIBE arg-list | _ALL_

arg-list

contains one or more arguments specified in the form <entry-name\ variable:

entry-name

is the name of a catalog entry that contains an SCL program. If the entry resides
in the current catalog, then entry-name can be a one-level name. If the entry
resides in a different catalog, then entry-name must be a four-level name, and the
entry must already be loaded into the application’s execution stack. A backslash
must follow the entry name.

variable

identifies an SCL variable to describe. The program that uses the specified
variable must already be loaded in the application’s execution stack.

_ALL_

describes all variables in all programs that are in the application’s execution stack.

Details

The DESCRIBE command displays the attributes of the specified variables. You can
also use dot notation to specify object attributes. The attributes reported are

Name

contains the name of the variable whose attributes are displayed.

Length

contains the variable’s length. All numeric variables have a length of 8 bytes.
Variables of type list return the number of items in the list.

Category

contains the variable’s class or category:

SYSTEM
designates a system variable.
WINDOW
    designates a window variable.

NONWINDOW
    designates a nonwindow variable.

Type
contains the data type of the variable:
 ARRAY ELMT
    array element
 CHAR
    character
 CHAR ARRAY
    character array
 LIST
    list
 LIST ARRAY
    list array
 NUM
    numeric
 NUM ARRAY
    numeric array
 OBJECT
    object
 OBJECT ARRAY
    object array

Examples

□ Display the name, data type, length and class of variable A:

DEBUG> des a
A is described as
 des age
 AGE  NUM 8 NONWINDOW

□ Display the name, data type, length, and class of all elements in the array ARR:

DEBUG> des arr

□ Display the attributes of array element ARR[i+j]:

DEBUG> des arr[i+j]

□ Display the attributes of variable A in the entry TEST1.SCL:

DEBUG> des test1\a

□ Display the attributes of all elements of array BRR in the entry TEST2.SCL:
DEBUG> des test2.scl\brr

- Display the attributes of object1:
  DEBUG> des object1

- Display the attributes of the visible attribute on object1:
  DEBUG> des object1.visible

- Display the attributes of the attribute name on object1 in the entry TEST2.SCL:
  DEBUG> des test2\object1.name

See Also

“ARGS” on page 189
“EXAMINE” on page 200
“PARM” on page 208
“PUTLIST” on page 209

---

## ENTER

Assigns one or more debugger commands to the ENTER key

**Syntax**

`ENTER <command-list>`

**command-list**

contains one or more debugger commands, separated by semicolons.

**Details**

The ENTER command assigns one or more debugger commands to the ENTER key. Each debugger command assignment replaces an existing debugger command assignment. To clear the key setting, enter the command without any options. By default, the ENTER key is set to the STEP command.

**Example**

Assign the commands EXAMINE and DESCRIBE, both for the variable ABC, to the ENTER key:

```
DEBUG> enter e abc; des abc
```

See Also

“STEP” on page 212

---

## ENVIRONMENT

Displays the developer debugging environment
Abbreviation: ENV

Syntax
ENVIRONMENT <<entry-name, >line-num> | RUN>

entry-name
sets the developer environment at the first executable statement in the program in
the specified entry. If the entry resides in the current catalog, then entry-name can
be a one-level name. If the entry resides in a different catalog, then entry-name must
be a four-level name, and the entry must already be loaded into the application's
execution stack. A backslash must follow the entry name.

line-num
is the line to display in reverse-video.

RUN
returns the debugger to executing the program.

Details
The ENVIRONMENT command enables you to display and modify the source program
(that is, it sets a developer debugging environment) for any program in the application's
execution stack while another program is active. When a developer environment is set,
the debugger generates messages showing both the current program environment and
the developer environment. In the developer environment, you can scroll through the
source program, set debugging requests, and operate on the variables. For example,
while TEST2.SCL is active, the ENVIRONMENT command enables you to display the
source code for TEST1.SCL, reset the values of several variables in TEST1.SCL, and
then return to TEST2.SCL.

By default, when you issue the ENVIRONMENT command from the current
executing program without options, it sets the current program environment as the
developer environment. If you issue the ENVIRONMENT command without an
argument from a program other than the current program, the developer environment
is reset to the program line containing the CALL DISPLAY statement.

Setting a developer environment does not change the way a program executes.
To return control to the active program, use the ENV RUN command to reset the
environment to the active program, or use the GO, STEP or JUMP command to leave
the developer environment and resume execution.

Example
Assume that an execution stack looks like this:

TEST3.SCL line 37
TEST2.SCL line 24
TEST1.SCL line 10

The following examples illustrate valid ENVIRONMENT commands and describe their
effect on the preceding execution stack:

- Display the source of TEST1.SCL with line 10 in reverse video:
  DEBUG> env test1.scl\10
  The output to the debugger MESSAGE window is
env test1.scl
Stop at line 37 in MYLIB.MYCAT.TEST3.SCL
Developer environment at line 10 in
MYLIB.MYCAT.TEST1.SCL

☐ Display the source of TEST2.SCL with line 45 in reverse video:
`DEBUG> env test2\45`

☐ Return to the current program environment (TEST3.SCL at line 37):
`DEBUG> env run`

☐ Attempt to return to the program TEST4.SCL:
`DEBUG> env test4\`
Because TEST4.SCL is not in the SCL execution stack, the SOURCE window
still displays TEST3.SCL. The output to the debugger MESSAGE window is
Program TEST4 is not active

See Also
“SWAP” on page 214

---

**EXAMINE**

Displays the value of one or more variables

Abbreviations:  EX, E

**Syntax**

`EXAMINE arg-list | _ALL_`

arg-list
contains one or more arguments specified in the form `<entry-name> >variable`
entry-name
names a catalog entry that contains an SCL program.
variable
identifies a standard SCL variable. The program that uses variable must already
be loaded in the application’s execution stack.

_ALL_
examines all variables defined in all programs in the application’s execution stack.

**Details**

The EXAMINE command displays the value of one or more specified variables or object attributes.

*Note:* You can examine only one object attribute at a time.
Examples

- Display the values of variables N and STR:
  ```
  DEBUG> e n str
  The output to the debugger MESSAGE window is
  e n str
  N = 10
  STR = ’abcdef’
  ```

- Display variable A in the entry TEST1.SCL, variable B in the current program, and variable C in the entry TEST2.SCL:
  ```
  DEBUG> e test1\a b test2\c
  ```

- Display the elements i, j, and k of the array CRR:
  ```
  DEBUG> e crr[i, j, k]
  ```

- Display the elements i+1, j*2, k-3 of the array CRR:
  ```
  DEBUG> e crr[i+1, j*2, k-3]
  ```

- Display the value of the text in the Text Entry control NAME:
  ```
  DEBUG> e name.text
  ```

- Display the value of the enabled attribute on object1:
  ```
  DEBUG> e object1.enabled
  ```

- Display the reference ID of an object or a list:
  ```
  DEBUG> e object1
  ```
  ```
  or
  DEBUG> e object1.attributeList
  ```

  Once the reference ID is known, use PUTLIST to display the list data.

See Also

- “ARGS” on page 189
- “DESCRIBE” on page 196
- “PARM” on page 208
- “PUTLIST” on page 209

GO

Starts or resumes execution of the active program from the current location

Abbreviation: G

Syntax

```
GO <entry-name\ | line-num | label-name | RETURN >
```
entry-name

is the name of a catalog entry that contains the SCL program to resume executing.

line-num

is the number of the program line to start executing or resume executing.

label

is the program label where execution is to start or resume.

RETURN

starts or resumes execution at the next RETURN statement.

Details

The GO command starts or resumes execution of the active program. By default, program statements execute continuously. However, you can specify one of the optional arguments to establish a temporary breakpoint that stops the program at the corresponding statement.

A temporary breakpoint established through the GO command is ignored when the debugger encounters a breakpoint that was previously set before it encounters the temporary breakpoint. That is, program execution suspends at the breakpoint that was previously set by the BREAK command rather than at the temporary breakpoint.

Examples

- Resume executing the program and execute its statements continuously:
  
  DEBUG> g

- Resume program execution and then suspend execution at the next RETURN statement:
  
  DEBUG> g return

- Resume program execution and then suspend execution at the statement in line 104:
  
  DEBUG> g 104

- Resume program execution and then suspend execution at the first statement of the MAIN section:
  
  DEBUG> go main

- Resume program execution and then suspend execution at the statement in line 15 in program TEST2:
  
  DEBUG> go test2\15

See Also

“JUMP” on page 204

“STEP” on page 212

HELP

Displays information about debugger commands
Syntax
HELP <command>

command
is the debugger command for which to display help. You must use full command names rather than abbreviations.

Details
The HELP command displays information describing the syntax and usage of debugger commands. If command is not supplied, HELP displays a list of the debugger commands. You can then select any command to receive information about that command.
You can also issue the HELP command from the command line of the debugger MESSAGE window.

Examples

- Display a list of the debugger commands:
  DEBUG> help
- Display the syntax and usage information for the BREAK command:
  DEBUG> help break

IF
Evaluates an expression and conditionally executes one or more debugger commands

Syntax
IF expression THEN clause <; ELSE clause>

expression
contains a condition to be evaluated before one or more commands are executed.

clause
contains either a single debugger command or a debugger DO list.

Details
The IF command immediately evaluates an expression and conditionally executes one or more debugger commands. The expression will contain dot notation because it resolves to a numeric variable. Character variables are converted.
The IF command must contain a THEN clause with one or more commands to execute if the expression is true. It can also contain an ELSE clause with one or more commands to execute if the expression is false. The ELSE clause must be separated from the THEN clause with a semicolon, and the ELSE clause cannot be entered separately.

**Examples**

- Examine the variable X if X contains a value that is greater than 0:
  ```
  DEBUG> if x > 0 then e x
  ```

- Examine the variable X if X is greater than 0, or examine the value of the variable Y if X is less than or equal to 0:
  ```
  DEBUG> if (x>0) then e x; else e y
  ```

- Execute the following actions if the value of variable X is less than variable Y and if Y is less than variable Z:
  - Delete all breakpoints in all program entries.
  - Set a breakpoint in the entry TEST2.SCL at the first executable statement.
  - Resume program execution.
    ```
    DEBUG> if ((x<y) & (y<z)) then
    do; d b _all_; b test2.scl\;g;end;
    ```

- Execute the following actions if the value of the variable X is 1:
  - Examine the value of variable A.
  - Set a breakpoint at line 5 of the program that is currently executing.
    If the value of X is not 1, then execute these actions:
    - Examine the value of variable B.
    - Set a breakpoint at line 15 of the program that is currently executing.
    ```
    DEBUG> if x=1 then do; e a; b 5; end;
    else do; e b; b 15; end;
    ```

- Set a breakpoint at line 15 of the program. Whenever the execution suspends at line 15, if the value of the variable DIVISOR is greater than 3, execute the STEP command; otherwise, examine the value of the variable DIVIDEND.
  ```
  DEBUG> b 15 do; if divisor>3 then st;
    else e dividend; end;
  ```

- Examine the variable X if the attribute attrValue on object1 is greater than 10:
  ```
  if object1.attrValue > 10 then e x
  ```

---

**JUMP**

Restarts execution of a suspended program

Abbreviation: J
Syntax

```
JUMP line-num
```

**line-num**

is the number of a program line at which to restart the suspended program. The specified line must contain at least one executable SCL statement.

**Details**

The JUMP command restarts program execution at the executable statement in the specified line. It is different from the GO command because none of the statements between the suspended statement and the specified line are executed. With this capability, the JUMP command enables you to skip execution of some code that causes incorrect results or program failure.

The JUMP command can restart only the current source entry. The values of all variables are the same as the values at the original suspending point.

Although the JUMP command can jump to any statement in the current source, if the target statement resides in a different section of code, then the first RETURN statement encountered in the section that contains the target statement is treated as the RETURN statement from the section where the JUMP command was executed.

For example, suppose you were in the TERM section and you issued a JUMP command to jump to a statement in the MAIN section. When the program resumes execution, the first RETURN statement that it encounters in the MAIN section terminates the program (as the RETURN statement does in the TERM section) instead of redisplaying the screen.

**Note:** Using the JUMP command to jump to a statement that is inside a DO loop may produce an illogical result.

**Example**

The following example illustrates the use of the JUMP command:

```
DEBUG> j 5
```

The output to the debugger MESSAGE window is

```
Stop at line 5 in MYLIB.MYCAT.TEST2.SCL
```

**See Also**

“GO” on page 201

“STEP” on page 212
Syntax

LIST <<BREAK | TRACE | WATCH | _ALL_> entry-name\ | _ALL_>

_ALL_
lists all breakpoints, tracepoints and watched variables. LIST _ALL_ performs the same function as LIST.

BREAK
lists all breakpoints.

TRACE
lists all tracepoints.

WATCH
lists all watched variables.

entry-name
is the name of a catalog entry that contains an SCL program. If the entry resides in the current catalog, then entry-name can be a one-level name. If the entry resides in a different catalog, then entry-name must be a four-level name, and the entry must already be loaded into the application’s execution stack. A backslash must follow the entry name.

_ALL_

 displays the debugging requests that are currently set for all entries.

Details

The LIST command displays a list of all debugging requests that have been set for an application. These requests include breakpoints, tracepoints, and watched variables. By default, the list contains all of the debugging requests for the current entry.

Examples

- List all the breakpoints, tracepoints, and watched variables for the current program:
  DEBUG> l _all_
  
  The output to the debugger MESSAGE window is
  
  _all_
  Stop at line 5 in MYLIB.MYCAT.TEST2.SCL
  List all the breakpoints in program
  MYLIB.MYCAT.TEST2.SCL
  Breakpoint has been set at line 4
  Breakpoint has been set at line 8
  Breakpoint has been set at line 10
  List all tracepoints in program
  MYLIB.MYCAT.TEST2.SCL
  No tracepoint has been set in program
MYLIB.MYCAT.TEST2.SCL
No variables have been watched in program MYLIB.MYCAT.TEST2.SCL

List all the breakpoints, tracepoints, and watched variables for all active programs in the execution stack:
DEBUG> l _all_ _all_

List all the breakpoints in the current entry:
DEBUG> l b

List all the breakpoints in all active programs in the execution stack:
DEBUG> l b _all_

List all the breakpoints, tracepoints, and watched variables in the entry TEST1.SCL:
DEBUG> l _all_ test1\n
List all the watched variables in the entry TEST3.SCL:
DEBUG> l w test3\n
See Also
“BREAK” on page 190
“DELETE” on page 194
“TRACE” on page 214
“WATCH” on page 217

MACEXPAND

Expands macro calls
Abbreviation: MACX

Syntax
MACEXPAND line-num

line-num
is the number of the program line that contains either a macro invocation or a macro variable reference to expand.

Details
The MACEXPAND command expands macro invocations and macro variables. The expansion is displayed in the debugger window. If the line does not contain a macro invocation or a macro variable reference, then the MACEXPAND command is ignored.

Example
In this example, the program contains the macro VALAMNT:
%macro valamnt(amount);
   if amount <0 or amount >500 then
      do;
         erroron amount;
         _msg_='Amount must be between $0 and $500.‘;
         stop;
      end;
   else erroroff amount;
%mend valamnt;

Line 33 of the SCL program calls the macro:
%valamnt(amt)

After entering 250 into the Amount control, enter in the debugger window:

DEBUG> macx 33

The debugger window displays the following output:

AMOUNT
$_T0 = AMOUNT < 0$
$_T1 = AMOUNT > 500$
$_T2 = $T0 OR $T1$
IF $T2 == 0 THEN #24 ERRORON(AMOUNT)
   _MSG_ = ‘Amount must be between $0 and $500.’
STOP
JUMP #26
ERROROFF(AMOUNT)

---

**PARM**

Displays the values of variables that are passed as parameters to any SCL function or routine

**Syntax**

**PARM**

**Details**

The PARM command displays the values of variables that are passed as parameters to an SCL function or routine. This command is valid only when the next executable statement contains a function call. Otherwise, the debugger issues a warning.

If a nested function call is encountered — that is, if the parameters passed to a function or routine are themselves function calls — then the PARM command displays the parameter list only for the nested function. You have to keep using the PARM command in order to display the parameter list for other function calls. For example, assume that the next executable statement is

```
str1=substr(upcase(string), min(x,y), max(x,y));
```
A PARM command first displays the parameter STRING, which is passed to the function UPCASE. A second PARM command displays the parameter list X, Y, which is passed to the function MIN. Subsequent PARM commands would display the parameter lists passed to the function MAX and then to SUBSTR.

Note: Once the values of arguments for a function or routine have been displayed, you cannot repeat the PARM command for the same function unless you are re-executing it.

Example

A PARM command issued at the following statement

call display ('test2', x, y);

generates the following output:

parm
Arguments passed to DISPLAY:
  1 (Character Literal)='test2'
Parameters passed to DISPLAY ENTRY:
  1 X=0
  2 Y=4

See Also

"ARGS" on page 189
"DESCRIBE" on page 196
"EXAMINE" on page 200

PUTLIST

Displays the contents of an SCL list

Syntax

PUTLIST <arg-list | n>

arg-list
contains one or more SCL list identifiers that are returned by the MAKELIST or COPYLIST function. Use the form <entry-name>, variable:

entry-name
is a catalog entry that contains the program that uses variable.

variable
is the variable that contains a list identifier.

n
is one or more numeric literals that represent the list to be printed.
Details

The PUTLIST command displays the contents of an SCL list in the debugger MESSAGE window. The list starts with a left parenthesis, (, to mark its beginning, followed by the list of items separated by blanks. Each named item is preceded by its name and an equal sign, =, but nothing is printed before items that do not have names. The PUTLIST function ends the list with a right parenthesis, ), followed by the list's identifier number within square brackets.

If a list appears more than once in the list being printed, the PUTLIST command displays (...) listid for the second and subsequent occurrences of the list. You should scan the output of the PUTLIST command for another occurrence of listid to view the full contents of the list. This prevents infinite loops if a list contains itself as a sublist.

Examples

- Print the contents of List A, which contains the numbers 17 and 328 and the character string 'Any string':

```
DEBUG > putlist a
```

This produces the following output:

```
( 17
  328
  'Any string'
 )[5]
```

- Print the list identified by number 5 (the same list shown in the previous example):

```
DEBUG> putlist 5
```

This produces the following output:

```
( 17
  328
  'Any string'
 )[5]
```

- Print the list identified by the dot notation object.dropoperations, assuming that dropoperations is a valid list attribute on the object identified by object:

```
putlist object.dropoperations
COPY=( POPMENUTEXT=’Copy here’
  ENABLED=’Yes’
  METHOD=’_drop’
 )[11601]
MOVE=( POPMENUTEXT=’Move here’
  ENABLED=’No’
  METHOD=’_drop’
 )[11599]
LINK=( POPMENUTEXT=’Link here’
  ENABLED=’No’
  METHOD=’_drop’
 )[11597]
)[11593]
```
**QUIT**

Terminates a debugger session

Abbreviation: Q

**Syntax**

QUIT

**Details**

The QUIT command terminates a debugger session and returns control to the point at which the debugger was invoked. You can use this command on the debugger command line at any time during program execution.

**SET**

Assigns new values to a specified variable

Abbreviation: S

**Syntax**

SET <entry-name> variable expression

**entry-name**

is the name of a catalog entry containing an SCL program entry that uses variable. If the entry resides in the current catalog, then entry-name can be a one-level name. If the entry resides in a different catalog, then entry-name must be a four-level name, and the entry must already be loaded into the application's execution stack. A backslash must follow the entry name.

**variable**

is an SCL variable.
expression

contains a standard debugger expression.

Details

The SET command assigns either a value or the result of a debugger expression to the specified variable. When you detect an error during program execution, you can use this command to assign new values to variables. This enables you to continue the debugging session instead of having to stop, modify a variable value, and recompile the program. You can also assign new values to variables in other active entries.

Examples

- Set variable A to the value of 3:
  ```debugger
  DEBUG> s a=3
  The output to the debugger MESSAGE window is
  Stop at line 5 in SASUSER.SCL.TEST2.SCL
  A = 3
  ```

- Set X to the value of item 1 in LIST:
  ```debugger
  DEBUG> s x=getitemc(list,1)
  ```

- Set variable A in program PROG1 to the value of the result of the expression a+c*3:
  ```debugger
  DEBUG> s prog1\a=a+c*3
  ```

- Assign to variable B the value 12345 concatenated with the value of B:
  ```debugger
  DEBUG> s b= 12345 || b
  ```

- Set array element ARR[1] to the value of the result of the expression a+3:
  ```debugger
  DEBUG> s arr[1]=a+3
  ```

- Set array element CRR[1,2,3] to the value of the result of the expression crr[1,1,2]+crr[1,1,3]:
  ```debugger
  DEBUG> s crr[1,2,3]=crr[1,1,2]+crr[1,1,3]
  ```

- Set the values of a whole array:
  ```debugger
  DEBUG> s crr=['a', 'b', 'c', 'd']
  ```

STEP

Executes statements one at a time in the active program

Abbreviation: ST

Syntax

STEP <OVER| O>
OVER

specifies that if the next executable statement is a CALL DISPLAY, FSEDIT, or
FSVIEW statement, the whole reference counts as a statement. By default, the
STEP command suspends program execution at the first executable statement of the
called program if that program was compiled with DEBUG ON.

Details

The STEP command executes one statement in the active program, starting with the
suspended statement. When you issue a STEP command, the command

- executes the next statement
- displays the entry name and line number
- returns control to the developer and displays the DEBUG> prompt.

By default, the STEP command suspends the execution at the first executable
statement in the called program if the current statement is a CALL DISPLAY or CALL
FSEDIT statement. The OVER option forces the debugger to count the call of the
DISPLAY, FSEDIT, or FSVIEW routine as a statement, and program execution stops at
the statement after the CALL statement. However, if the called program contains a
display, execution is not suspended until you leave the display window.

When the STEP command is used to execute a SELECT statement, it jumps directly
to the appropriate WHEN or OTHERWISE clause without stepping through any
intervening WHEN statements.

Example

Suppose you are using the STEP command to execute the following program, which is
stopped at line 15. If VAL contains 99, the STEP command goes to line 116 immediately.

```sc lubricator
line #
15  select (val);
16    when (1)
17      call display('a1');
18    when (2)
19      call display('a2');
...more SCL statements...
113   when (98)
114     call display('a98');
115   when (99)
116     call display('a99');
117   when (100)
118     call display('a100');
119   otherwise
120     call display('other');
121  end;
```
SWAP

Switches control between the debugger SOURCE window and MESSAGE window

Syntax

SWAP

Details

The SWAP command enables you to switch control between the MESSAGE window and the SOURCE window when the debugger is running. When a debugging session is initiated, the control defaults to the MESSAGE window until you issue a command. While the program is still being executed, the SWAP command enables you to switch control between the SOURCE and MESSAGE window so that you can scroll and view the text of the program and also continue monitoring program execution.

See Also

“ENVIRONMENT” on page 198

TRACE

Sets a tracepoint for tracing the execution of the corresponding statement

Abbreviation: T

Syntax

TRACE <location <AFTER count <<WHEN clause| DO list>>

location

specifies where to set a tracepoint (at the current line, by default).

_ALL_

sets a tracepoint at every SCL executable program statement.

ENTRY

sets a tracepoint at the first executable statement in every entry in the current catalog that contains an SCL program.
entry-name,
  sets a tracepoint in an SCL program.

label
  sets a tracepoint at the first executable statement in an SCL reserved label or in a
  user-defined label.

line-num
  sets a tracepoint at the specified line.

AFTER count
  is the number of times for the debugger to execute a statement before executing the
  TRACE command.

WHEN clause
  specifies an expression that must be true in order for the command to be executed.

DO list
  contains one or more debugger commands to execute. Use semicolons to separate
  multiple commands.

Details
The TRACE command sets a tracepoint at a specified statement and traces the
execution of that statement.
A tracepoint differs from a breakpoint because a tracepoint resumes program
execution after temporary suspension. Also, a tracepoint has a higher priority than a
breakpoint. If a statement has been specified both as a tracepoint and as a breakpoint,
the debugger first prints the trace message and then suspends program execution. Each
time the tracepoint statement is encountered, the debugger does the following:
  □ suspends program execution
  □ checks the count that is specified with the AFTER command and resumes program
    execution if the specified number of tracepoint activations has not been reached
  □ evaluates any conditions specified in a WHEN clause and resumes execution if the
    condition evaluates as false
  □ displays the entry name and line number at which execution is suspended
  □ executes any command that is specified in a DO list
  □ resumes program execution.

Examples

□ Trace the statement at line 45:
  DEBUG> t 45

□ Trace each executable statement:
  DEBUG> t _all_

□ Trace each executable statement and print all the values:
  DEBUG> t _all_ do; e _all_; end

□ Trace the first executable statement in each program:
  DEBUG> t entry
- Trace the first executable statement in the program's MAIN section:
  `DEBUG> t MAIN`
- Trace the statement at line 45 after each third execution:
  `DEBUG> t 45 after 3`
- Trace the statement at line 45 when the values of the variables DIVISOR and DIVIDEND are both 0:
  `DEBUG> t 45 when (divisor=0 and dividend=0)`
- Trace the statement at line 5 in the entry TEST1:
  `DEBUG> t test1\5`

**See Also**

“BREAK” on page 190
“DELETE” on page 194
“LIST” on page 205
“WATCH” on page 217

---

**TRACEBACK**

Displays the traceback of the entire SCL execution stack

Abbreviation: TB

---

**Syntax**

`TRACEBACK <_ALL_>`

**_ALL_**

displays the link stack information in addition to the SCL execution stack.

**Details**

The TRACEBACK command displays the entire execution stack, which consists of the program that is currently being executed and all programs that were called to display the current program. In addition, the _ALL_ argument displays the link stack, which consists of the labeled sections that are called within the program.

The display of the link stack does not include labeled sections that are called with a GOTO statement.

**Example**

START.SCL contains a link at line 5 and calls ANALYZE.SCL at line 9.
ANALYZE.SCL contains links at lines 5, 11, and 16.

Running the debugger and issuing TRACEBACK at line 21 of ANALYZE.SCL produces the following output:
---- Print The Traceback ----
In routine: TEST.TRACEBAK.ANALYZE.SCL line 21
Called from TEST.TRACEBAK.START.SCL line 9

Running the debugger and issuing TRACEBACK _ALL_ at line 21 of ANALYZE.SCL produces the following output:

---- Print The Traceback ----
In routine: TEST.TRACEBAK.ANALYZE.SCL line 21
  Linked from TEST.TRACEBAK.ANALYZE.SCL line 16
  Linked from TEST.TRACEBAK.ANALYZE.SCL line 11
  Linked from TEST.TRACEBAK.ANALYZE.SCL line 5
Called from TEST.TRACEBAK.START.SCL line 9
  Linked from TEST.TRACEBAK.START.SCL line 5

WATCH

Suspends program execution when the value of a specified variable has been modified

Abbreviation:  W

Syntax

WATCH <entry-name> variable <AFTER count> <WHEN clause | DO list>

entry-name

is the name of the entry that contains the variable to be watched. The debugger starts watching the variable at the first executable statement in the program in the specified entry. If the entry resides in the current catalog, then entry-name can be a one-level name. If the entry resides in a different catalog, then entry-name must be a four-level name, and the entry must already be loaded into the application’s execution stack. A backslash must follow the entry name.

variable

is the name of the variable to watch.

AFTER count

specifies the number of times for the value of the variable to be changed before the debugger suspends program execution. Therefore, for an AFTER specification of 3, the program halts when the value of the watched variable is changed for the third time.

WHEN clause

specifies an expression that must be true in order for the command to be executed. Clause can contain SCL functions.

DO list

contains one or more debugger commands to execute. Use semicolons to separate multiple commands.
Details

The WATCH command monitors a variable and suspends program execution when the value of the variable is modified. A variable is called a watched entry parameter if it is defined as both a watched variable and as an ENTRY statement parameter. A program is not suspended when the value of a watched entry parameter is changed by a called program. However, a program is suspended when a changed value for a watched entry parameter is copied back to the calling program.

Each time the variable is modified, the debugger

- suspends program execution
- checks for any AFTER count and resumes program execution if the specified number of changes has not been reached
- evaluates the WHEN condition and resumes execution if the WHEN condition is false
- displays the entry name and line number at which execution has been suspended
- displays the variable’s old value
- displays the variable’s new value
- executes any commands that are provided in a DO list
- returns control to the developer and displays the DEBUG> prompt.

You can watch only variables that are in the current program.

Examples

- Monitor the variable DIVISOR in TEST2 for value changes:
  
  DEBUG> w divisor

  The output to the MESSAGE window is

  Stop at line 6 in MYLIB.MYCAT.TEST2.SCL
  Watch variable DIVISOR has been modified
  Old value=1
  New value=99

- Monitor all the elements in the array NUM for value changes:
  
  DEBUG> w num

- Monitor the variable DIVISOR in TEST1.SCL for value changes:
  
  DEBUG> w test1\divisor

- Monitor an object attribute for its value changes:
  
  DEBUG> w object.attribute

- Monitor the variable A[1] for value changes and suspend program execution after its value has been altered three times:
  
  DEBUG> w a[1] after 3

- Monitor A[1] for value changes and suspend program execution when neither X nor Y is 0:
  
  DEBUG> w a[1] when (x^=0 and y^=0)

- Monitor FIELD1 for value changes and suspend program execution after the third change in the value of FIELD1 when the variables DIVIDEND and DIVISOR both equal 0:
DEBUG> w field1 after 3 when (dividend=0
  and divisor=0)

Monitor X when it has the same value as item 1 in LIST.

DEBUG> w x when x=getitemc(list,1)

See Also

“BREAK” on page 190
“DELETE” on page 194
“LIST” on page 205
“TRACE” on page 214