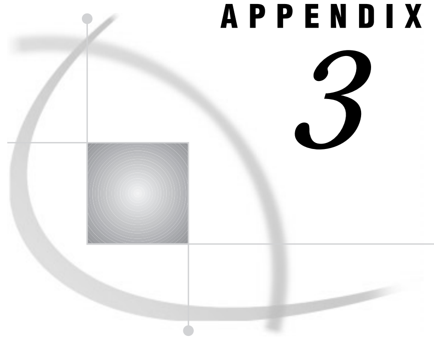


APPENDIX

3

DATA Step Debugger

Introduction **1194**
- *Definition: What is Debugging?* **1194**
- *Definition: The DATA Step Debugger* **1194**

Basic Usage **1195**
- *How a Debugger Session Works* **1195**
- *Using the Windows* **1195**
- *Entering Commands* **1195**
- *Working with Expressions* **1196**
- *Assigning Commands to Function Keys* **1196**

Advanced Usage: Using the Macro Facility with the Debugger **1196**
- *Using Macros as Debugging Tools* **1196**
- *Creating Customized Debugging Commands with Macros* **1196**
- *Debugging a DATA Step Generated by a Macro* **1197**

Examples **1197**
- *Example 1: Debugging a Simple DATA Step* **1197**
  - *Discovering a Problem* **1197**
  - *Using the DEBUG Option* **1198**
  - *Examining Data Values after the First Iteration* **1198**
  - *Examining Data Values after the Second Iteration* **1199**
  - *Ending the Debugger* **1200**
  - *Correcting the DATA Step* **1200**
- *Example 2: Working with Formats* **1201**
- *Example 3: Debugging DO Loops* **1204**
- *Example 4: Examining Formatted Values of Variables* **1204**

Commands **1205**
- *List of Debugger Commands* **1205**
- *Debugger Commands by Category* **1205**

Dictionary **1206**
- *BREAK* **1206**
- *CALCULATE* **1208**
- *DELETE* **1209**
- *DESCRIBE* **1210**
- *ENTER* **1211**
- *EXAMINE* **1211**
- *GO* **1213**
- *HELP* **1214**
- *JUMP* **1214**
- *LIST* **1215**
- *QUIT* **1216**
- *SET* **1217**
- *STEP* **1218**

SWAP 1218
TRACE 1219
WATCH 1220

Introduction

Definition: What is Debugging?

Debugging is the process of removing logic errors from a program. Unlike syntax errors, logic errors do not stop a program from running. Instead, they cause the program to produce unexpected results. For example, if you create a DATA step that keeps track of inventory, and your program shows that you are out of stock but your warehouse is full, you have a logic error in your program.

To debug a DATA step, you could

- copy a few lines of the step into another DATA step, execute it, and print the results of those statements
- insert PUT statements at selected places in the DATA step, submit the step, and examine the values that are displayed in the SAS log.
- use the DATA step debugger.

While the SAS log can help you identify data errors, the DATA step debugger offers you an easier, interactive way to identify logic errors, and sometimes data errors, in DATA steps.

Definition: The DATA Step Debugger

The DATA step debugger is part of base SAS software and consists of windows and a group of commands. By issuing commands, you can execute DATA step statements one by one and pause to display the resulting variable values in a window. By observing the results that are displayed, you can determine where the logic error lies. Because the debugger is interactive, you can repeat the process of issuing commands and observing the results as many times as needed in a single debugging session. To invoke the debugger, add the DEBUG option to the DATA statement and execute the program.

The DATA step debugger enables you to perform the following tasks:

- execute statements one by one or in groups
- bypass execution of one or more statements
- suspend execution at selected statements, either in each iteration of DATA step statements or on a condition you specify, and resume execution on command
- monitor the values of selected variables and suspend execution at the point a value changes
- display the values of variables and assign new values to them
- display the attributes of variables
- receive help for individual debugger commands
- assign debugger commands to function keys
- use the macro facility to generate customized debugger commands.

Basic Usage

How a Debugger Session Works

When you submit a DATA step with the DEBUG option, SAS compiles the step, displays the debugger windows, and pauses until you enter a debugger command to begin execution. If you begin execution with the GO command, for example, SAS executes each statement in the DATA step. To suspend execution at a particular line in the DATA step, use the BREAK command to set breakpoints at statements you select. Then issue the GO command. The GO command starts or resumes execution until the breakpoint is reached.

To execute the DATA step one statement at a time or a few statements at a time, use the STEP command. By default, the STEP command is mapped to the ENTER key.

In a debugging session, statements in a DATA step can iterate as many times as they would outside the debugging session. When the last iteration has finished, a message appears in the DEBUGGER LOG window.

You cannot restart DATA step execution in a debugging session after the DATA step finishes executing. You must resubmit the DATA step in your SAS session. However, you can examine the final values of variables after execution has ended.

You can debug only one DATA step at a time. You can use the debugger only with a DATA step, and not with a PROC step.

Using the Windows

The DATA step debugger contains two primary windows, the DEBUGGER LOG and the DEBUGGER SOURCE windows. The windows appear when you execute a DATA step with the DEBUG option.

The DEBUGGER LOG window records the debugger commands you issue and their results. The last line is the debugger command line, where you issue debugger commands. The debugger command line is marked with a greater than (>) prompt.

The DEBUGGER SOURCE window contains the SAS statements that comprise the DATA step you are debugging. The window enables you to view your position in the DATA step as you debug your program. In the window, the SAS statements have the same line numbers as they do in the SAS log.

You can enter windowing environment commands on the window command lines. You can also execute commands by using function keys.

Entering Commands

Enter DATA step debugger commands on the debugger command line. For a list of commands and their descriptions, refer to “Debugger Commands by Category” on page 1205. Follow these rules when you enter a command:

- A command can occupy only one line (except for a DO group).
- A DO group can extend over more than one line.
- To enter multiple commands, separate the commands with semicolons:

```
examine _all_; set letter='bill'; examine letter
```

Working with Expressions

All SAS operators that are described in Appendix 2, “SAS Operators,” on page 1189, are valid in debugger expressions. Debugger expressions cannot contain functions.

A debugger expression must fit on one line. You cannot continue an expression on another line.

Assigning Commands to Function Keys

To assign debugger commands to function keys, open the Keys window. Position your cursor in the Definitions column of the function key you want to assign, and begin the command with the term DSD. To assign more than one command to a function key, enclose the commands (separated by semicolons) in quotation marks. Be sure to save your changes. These examples show commands assigned to function keys:

- ```
  dsd step3
  ```
- ```
  dsd 'examine cost saleprice; go 120;'
  ```

Advanced Usage: Using the Macro Facility with the Debugger

You can use the SAS macro facility with the debugger to invoke macros from the DEBUGGER LOG command line. You can also define macros and use macro program statements, such as %LET, on the debugger command line.

Using Macros as Debugging Tools

Macros are useful for storing a series of debugger commands. Executing the macro at the DEBUGGER LOG command line then generates the entire series of debugger commands. You can also use macros with parameters to build different series of debugger commands based on various conditions.

Creating Customized Debugging Commands with Macros

You can create a customized debugging command by defining a macro on the DEBUGGER LOG command line. Then invoke the macro from the command line. For example, to examine the variable COST, to execute five statements, and then to examine the variable DURATION, define the following macro (in this case the macro is called EC). Note that the example uses the alias for the EXAMINE command.

```
%macro ec; ex cost; step 5; ex duration; %mend ec;
```

To issue the commands, invoke macro EC from the DEBUGGER LOG command line:

```
%ec
```

The DEBUGGER LOG displays the value of COST, executes the next five statements, and then displays the value of DURATION.

Note: Defining a macro on the DEBUGGER LOG command line allows you to use the macro only during the current debugging session, because the macro is not permanently stored. To create a permanently stored macro, use the Program Editor. △

Debugging a DATA Step Generated by a Macro

You can use a macro to generate a DATA step, but debugging a DATA step that is generated by a macro can be difficult. The SAS log displays a copy of the macro, but not the DATA step that the macro generated. If you use the DEBUG option at this point, the text that the macro generates appears as a continuous stream to the debugger. As a result, there are no line breaks where execution can pause.

To debug a DATA step that is generated by a macro, use the following steps:

1. Use the MPRINT and MFILE system options when you execute your program.
2. Assign the fileref MPRINT to an existing external file. MFILE routes the program output to the external file. Note that if you rerun your program, current output appends to the previous output in your file.
3. Invoke the macro from a SAS session.
4. In the Program Editor window, issue the INCLUDE command or use the File menu to open your external file.
5. Add the DEBUG option to the DATA statement and begin a debugging session.
6. When you locate the logic error, correct the portion of the macro that generated that statement or statements.

Examples

Example 1: Debugging a Simple DATA Step

This example shows how to debug a DATA step when output is missing.

Discovering a Problem

This program creates information about a travel tour group. The data files contain two types of records. One type contains the tour code, and the other type contains customer information. The program creates a report listing tour number, name, age, and sex for each customer.

```
/* first execution */
data tours (drop=type);
   input @1 type $ @;
   if type='H' then do;
      input @3 Tour $20.;
      return;
      end;
   else if type='P' then do;
      input @3 Name $10. Age 2. +1 Sex $1.;
      output;
      end;
   datalines;
H Tour 101
```

```
P Mary E    21 F
P George S  45 M
P Susan K    3 F
H Tour 102
P Adelle S  79 M
P Walter P  55 M
P Fran I    63 F
;

proc print data=tours;
   title 'Tour List';
run;
```

```
                     Tour List                              1
          Obs    Tour       Name       Age     Sex

           1              Mary E        21      F
           2              George S      45      M
           3              Susan K        3      F
           4              Adelle S      79      M
           5              Walter P      55      M
           6              Fran I        63      F
```

The program executes without error, but the output is unexpected. The output does not contain values for the variable Tour. Viewing the SAS log will not help you debug the program because the data are valid and no errors appear in the log. To help identify the logic error, run the DATA step again using the DATA step debugger.

Using the DEBUG Option

To invoke the DATA step debugger, add the DEBUG option to the DATA statement and resubmit the DATA step:

```
data tours (drop=type) / debug;
```

The following display shows the resulting two debugger windows.

The upper window is the DEBUGGER LOG window. Issue debugger commands in this window by typing commands on the debugger command line (the bottom line, marked by a >). The debugger displays the command and results in the upper part of the window.

The lower window is the DEBUGGER SOURCE window. It displays the DATA step submitted with the DEBUG option. Each line in the DATA step is numbered with the same line number used in the SAS log. One line appears in reverse video (or other highlighting, depending on your monitor). DATA step execution pauses *just before* the execution of the highlighted statement.

At the beginning of your debugging session, the first executable line after the DATA statement is highlighted. This means that SAS has compiled the step and will begin to execute the step at the top of the DATA step loop.

Examining Data Values after the First Iteration

To debug a DATA step, create a hypothesis about the logic error and test it by examining the values of variables at various points in the program. For example, issue

the EXAMINE command from the debugger command line to display the values of all variables in the program data vector before execution begins:

```
examine _all_
```

Note: Most debugger commands have abbreviations, and you can assign commands to function keys. The examples in this section, however, show the full command name to help you find the commands in "Debugger Commands by Category" on page 1205. △

When you press ENTER, the following display appears:

The values of all variables appear in the DEBUGGER LOG window. SAS has compiled, but not yet executed, the INPUT statement.

Use the STEP command to execute the DATA step statements one at a time. By default, the STEP command is assigned to the ENTER key. Press ENTER repeatedly to step through the first iteration of the DATA step, and stop when the RETURN statement in the program is highlighted in the DEBUGGER SOURCE window.

Because Tour information was missing in the program output, enter the EXAMINE command to view the value of the variable Tour for the first iteration of the DATA step.

```
examine tour
```

The following display shows the results:

The variable Tour contains the value Tour 101, showing you that Tour was read. The first iteration of the DATA step worked as intended. Press ENTER to reach the top of the DATA step.

Examining Data Values after the Second Iteration

You can use the BREAK command (also known as *setting a breakpoint*) to suspend DATA step execution at a particular line you designate. In this example, suspend execution before executing the ELSE statement by setting a breakpoint at line 7.

```
break 7
```

When you press ENTER, an exclamation point appears at line 7 in the DEBUGGER SOURCE window to mark the breakpoint:

Execute the GO command to continue DATA step execution until it reaches the breakpoint (in this case, line 7):

```
go
```

The following display shows the result:

SAS suspended execution *just before* the ELSE statement in line 7. Examine the values of all the variables to see their status at this point.

```
examine _all_
```

The following display shows the values:

You expect to see a value for Tour, but it does not appear. The program data vector gets reset to missing values at the beginning of each iteration and therefore does not

retain the value of Tour. To solve the logic problem, you need to include a RETAIN statement in the SAS program.

Ending the Debugger

To end the debugging session, issue the QUIT command on the debugger command line:

```
quit
```

The debugging windows disappear, and the original SAS session resumes.

Correcting the DATA Step

Correct the original program by adding the RETAIN statement. Delete the DEBUG option from the DATA step, and resubmit the program:

```
   /* corrected version */
data tours (drop=type);
   retain Tour;
   input @1 type $ @;
   if type='H' then do;
      input @3 Tour $20.;
      return;
      end;
   else if type='P' then do;
      input @3 Name $10. Age 2. +1 Sex $1.;
      output;
      end;
datalines;
H Tour 101
P Mary E    21 F
P George S  45 M
P Susan K    3 F
H Tour 102
P Adelle S  79 M
P Walter P  55 M
P Fran I    63 F
;

run;

proc print;
   title 'Tour List';
run;
```

The values for Tour now appear in the output:

```
                         Tour List                              1
           Obs      Tour         Name       Age    Sex

            1     Tour 101    Mary E       21      F
            2     Tour 101    George S     45      M
            3     Tour 101    Susan K       3      F
            4     Tour 102    Adelle S     79      M
            5     Tour 102    Walter P     55      M
            6     Tour 102    Fran I       63      F
```

Example 2: Working with Formats

This example shows how to debug a program when you use format statements to format dates. The following program creates a report that lists travel tour dates for specific countries.

```
options yearcutoff=1920;

data tours;
  length Country $ 10;
  input Country $10. Start : mmddyy. End : mmddyy.;
  Duration=end-start;
datalines;
Italy       033000 041300
Brazil      021900 022800
Japan       052200 061500
Venezuela   110300 11800
Australia   122100 011501
;

proc print data=tours;
   format start end date9.;
   title 'Tour Duration';
run;
```

```
                         Tour Duration                              1

     Obs    Country          Start          End    Duration

      1     Italy        30MAR2000    13APR2000        14
      2     Brazil       19FEB2000    28FEB2000         9
      3     Japan        22MAY2000    15JUN2000        24
      4     Venezuela    03NOV2000    18JAN2000      -290
      5     Australia    21DEC2000    15JAN2001        25
```

The value of Duration for the tour to Venezuela shows a negative number, -290 days. To help identify the error, run the DATA step again using the DATA step debugger. SAS displays the following debugger windows:

At the DEBUGGER LOG command line, issue the EXAMINE command to display the values of all variables in the program data vector before execution begins:

```
examine _all_
```

Initial values of all variables appear in the DEBUGGER LOG window. SAS has not yet executed the INPUT statement.

Press ENTER to issue the STEP command. SAS executes the INPUT statement, and the assignment statement is now highlighted.

Issue the EXAMINE command to display the current value of all variables:

```
examine _all_
```

The following display shows the results:

Because a problem exists with the Venezuela tour, suspend execution before the assignment statement when the value of Country equals Venezuela. Set a breakpoint to do this:

```
break 4 when country='Venezuela'
```

Execute the GO command to resume program execution:

```
go
```

SAS stops execution when the country name is Venezuela. You can examine Start and End tour dates for the Venezuela trip. Because the assignment statement is highlighted (indicating that SAS has not yet executed that statement), there will be no value for Duration.

Execute the EXAMINE command to view the value of the variables after execution:

```
examine _all_
```

The following display shows the results:

To view formatted SAS dates, issue the EXAMINE command using the DATE*w.* format:

```
examine start date7. end date7.
```

The following display shows the results:

Because the tour ends on November 18, 2000, and not on January 18, 2000, there is an error in the variable End. Examine the source data in the program and notice that the value for End has a typographical error. By using the SET command, you can temporarily set the value of End to November 18 to see if you get the anticipated result. Issue the SET command using the DDMMMYY*w.* format:

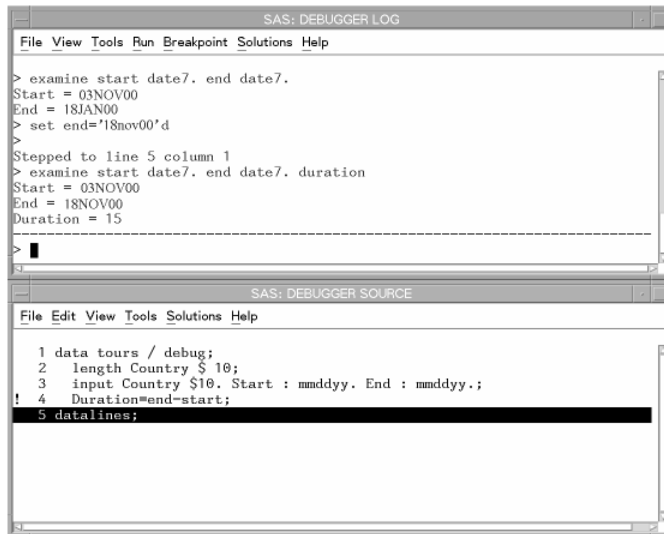
```
set end='18nov00'd
```

Press ENTER to issue the STEP command and execute the assignment statement.

Issue the EXAMINE command to view the tour date and Duration fields:

```
examine start date7. end date7. duration
```

The following display shows the results:



The Start, End, and Duration fields contain correct data.

End the debugging session by issuing the QUIT command on the DEBUGGER LOG command line. Correct the original data in the SAS program, delete the DEBUG option, and resubmit the program.

```
   /* corrected version */
options yearcutoff=1920;

data tours;
  length Country $ 10;
  input Country $10. Start : mmddyy. End : mmddyy.;
  duration=end-start;
datalines;
Italy       033000 041300
Brazil      021900 022800
Japan       052200 061500
Venezuela   110300 111800
Australia   122100 011501
;

proc print data=tours;
   format start end date9.;
   title 'Tour Duration';
run;
```

```
                          Tour Duration                              1

    Obs    Country          Start          End      duration

     1     Italy        30MAR2000    13APR2000         14
     2     Brazil       19FEB2000    28FEB2000          9
     3     Japan        22MAY2000    15JUN2000         24
     4     Venezuela    03NOV2000    18NOV2000         15
     5     Australia    21DEC2000    15JAN2001         25
```

Example 3: Debugging DO Loops

An iterative DO, DO WHILE, or DO UNTIL statement can iterate many times during a single iteration of the DATA step. When you debug DO loops, you can examine several iterations of the loop by using the AFTER option in the BREAK command. The AFTER option requires a number that indicates how many times the loop will iterate before it reaches the breakpoint. The BREAK command then suspends program execution. For example, consider this data set:

```
data new / debug;
   set old;
   do i=1 to 20;
      newtest=oldtest+i;
      output;
   end;
run;
```

To set a breakpoint at the assignment statement (line 4 in this example) after every 5 iterations of the DO loop, issue this command:

```
break 4 after 5
```

When you issue the GO commands, the debugger suspends execution when I has the values of 5, 10, 15, and 20 in the DO loop iteration.

In an iterative DO loop, select a value for the AFTER option that can be divided evenly into the number of iterations of the loop. For example, in this DATA step, 5 can be evenly divided into 20. When the DO loop iterates the second time, I again has the values of 5, 10, 15, and 20.

If you do not select a value that can be evenly divided (such as 3 in this example), the AFTER option causes the debugger to suspend execution when I has the values of 3, 6, 9, 12, 15, and 18. When the DO loop iterates the second time, I has the values of 1, 4, 7, 10, 13, and 16.

Example 4: Examining Formatted Values of Variables

You can use a SAS format or a user-created format when you display a value with the EXAMINE command. For example, assume the variable BEGIN contains a SAS date value. To display the day of the week and date, use the SAS WEEKDATE*w*. format with EXAMINE:

```
examine begin weekdate17.
```

When the value of BEGIN is 033001, the debugger displays

```
Sun, Mar 30, 2001
```

As another example, you can create a format named SIZE:

```
proc format;
   value size 1-5='small'
              6-10='medium'
             11-high='large';
run;
```

To debug a DATA step that applies the format SIZE. to the variable STOCKNUM, use the format with EXAMINE:

```
examine stocknum size.
```

When the value of STOCKNUM is 7, for example, the debugger displays

```
STOCKNUM = medium
```

Commands

List of Debugger Commands

BREAK	JUMP
CALCULATE	LIST
DELETE	QUIT
DESCRIBE	SET
ENTER	STEP
EXAMINE	SWAP
GO	TRACE
HELP	WATCH

Debugger Commands by Category

Table A3.1 Categories and Descriptions of Debugger Commands

Category	DATA Step Debugger	Description
Controlling Program Execution	"GO" on page 1213	Starts or resumes execution of the DATA step
	"JUMP" on page 1214	Restarts execution of a suspended program
	"STEP" on page 1218	Executes statements one at a time in the active program
Controlling the Windows	"HELP" on page 1214	Displays information about debugger commands
	"SWAP" on page 1218	Switches control between the SOURCE window and the LOG window
Manipulating DATA Step Variables	"CALCULATE" on page 1208	Evaluates a debugger expression and displays the result
	"DESCRIBE" on page 1210	Displays the attributes of one or more variables
	"EXAMINE" on page 1211	Displays the value of one or more variables
	"SET" on page 1217	Assigns a new value to a specified variable
Manipulating Debugging Requests	"BREAK" on page 1206	Suspends program execution at an executable statement

	“DELETE” on page 1209	Deletes breakpoints or the watch status of variables in the DATA step
	“LIST” on page 1215	Displays all occurrences of the item that is listed in the argument
	“TRACE” on page 1219	Controls whether the debugger displays a continuous record of the DATA step execution
	“WATCH” on page 1220	Suspends execution when the value of a specified variable changes
Tailoring the Debugger	“ENTER” on page 1211	Assigns one or more debugger commands to the ENTER key
Terminating the Debugger	“QUIT” on page 1216	Terminates a debugger session

Dictionary

BREAK

Suspends program execution at an executable statement

Category: Manipulating Debugging Requests

Alias: B

Syntax

BREAK *location* <AFTER *count*> <WHEN *expression*> <DO *group* >

Arguments

location

specifies where to set a breakpoint. *Location* must be one of these:

label	a statement label. The breakpoint is set at the statement that follows the label.
line-number	the number of a program line at which to set a breakpoint.
*	the current line.

AFTER *count*

honors the breakpoint each time the statement has been executed *count* times. The counting is continuous. That is, when the AFTER option applies to a statement inside a DO loop, the count continues from one iteration of the loop to the next. The debugger does not reset the *count* value to 1 at the beginning of each iteration.

If a BREAK command contains both AFTER and WHEN, AFTER is evaluated first. If the AFTER count is satisfied, the WHEN expression is evaluated.

Tip: The AFTER option is useful in debugging DO loops.

WHEN *expression*
honors a breakpoint when the expression is true.

DO *group*
is one or more debugger commands enclosed by a DO and an END statement. The syntax of the DO *group* is

DO; *command-1* < ... ; *command-n*; >END;

command
specifies a debugger command. Separate multiple commands by semicolons.

A DO group can span more than one line and can contain IF-THEN/ELSE statements, as shown:

IF *expression* THEN *command*; <ELSE *command*;>
IF *expression* THEN DO *group*; <ELSE DO *group*;>

IF evaluates an expression. When the condition is true, the debugger command or DO group in the THEN clause executes. An optional ELSE command gives an alternative action if the condition is not true. You can use these arguments with IF:

expression
specifies a debugger expression. A nonzero, nonmissing result causes the expression to be true. A result of zero or missing causes the expression to be false.

command
specifies a single debugger command.

DO *group*
specifies a DO group.

Details

The BREAK command suspends execution of the DATA step at a specified statement. Executing the BREAK command is called *setting a breakpoint*.

When the debugger detects a breakpoint, it

- checks the AFTER *count* value, if present, and suspends execution if *count* breakpoint activations have been reached
- evaluates the WHEN expression, if present, and suspends execution if the condition that is evaluated is true
- suspends execution if neither an AFTER nor a WHEN clause is present
- displays the line number at which execution is suspended
- executes any commands that are present in a DO group
- returns control to the user with a > prompt.

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

Examples

- Set a breakpoint at line 5 in the current program:

```
b 5
```

- Set a breakpoint at the statement after the statement label **eoflabel**:

```
b eoflabel
```

- Set a breakpoint at line 45 that will be honored after every third execution of line 45:

```
b 45 after 3
```

- Set a breakpoint at line 45 that will be honored after every third execution of that line only when the values of both DIVISOR and DIVIDEND are 0:

```
b 45 after 3
       when (divisor=0 and dividend=0)
```

- Set a breakpoint at line 45 of the program and examine the values of variables NAME and AGE:

```
b 45 do; ex name age; end;
```

- Set a breakpoint at line 15 of the program. If the value of DIVISOR is greater than 3, execute STEP; otherwise, display the value of DIVIDEND.

```
b 15 do; if divisor>3 then st;
           else ex dividend; end;
```

See Also

Commands:

"DELETE" on page 1209
"WATCH" on page 1220

CALCULATE

Evaluates a debugger expression and displays the result

Category: Manipulating DATA Step Variables

Syntax

CALC *expression*

Arguments

expression
specifies any debugger expression.
Restriction: Debugger expressions cannot contain functions.

Details

The CALCULATE command evaluates debugger expressions and displays the result. The result must be numeric.

Examples

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

  ```
  calc (1.1+1.2+3.4)*0.5
  ```

- Calculate the sum of STARTAGE and DURATION:

  ```
  calc startage+duration
  ```

- 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:

  ```
  calc (((sale-downpay)*rate)/12)+50
  ```

See Also

"Working with Expressions" on page 1196 for information on debugger expressions

---

DELETE

Deletes breakpoints or the watch status of variables in the DATA step

Category: Manipulating Debugging Requests

Alias: D

Syntax

DELETE BREAK *location*

DELETE WATCH *variable(s)* | _ALL_

Arguments

BREAK
deletes breakpoints.

Alias: B

location
specifies a breakpoint location to be deleted. *Location* can have one of these values:

ALL	all current breakpoints in the DATA step.
label	the statement after a statement label.
line-number	the number of a program line.
*	the breakpoint from the current line.

WATCH
deletes watched status of variables.

Alias: W

variable
names one or more watched variables for which the watch status is deleted.

ALL
specifies that the watch status is deleted for all watched variables.

Examples

- Delete the breakpoint at the statement label **eoflabel**:

  ```
  d b eoflabel
  ```

- Delete the watch status from the variable ABC in the current DATA step:

  ```
  d w abc
  ```

See Also

Commands:

"BREAK" on page 1206
"WATCH" on page 1220

DESCRIBE

Displays the attributes of one or more variables

Category: Manipulating DATA Step Variables

Alias: DESC

Syntax

DESCRIBE *variable(s)* | _ALL_

Arguments

variable
identifies a DATA step variable.

ALL
indicates all variables that are defined in the DATA step.

Details

The DESCRIBE command displays the attributes of one or more specified variables.
DESCRIBE reports the name, type, and length of the variable, and, if present, the informat, format, or variable label.

Examples

- □ Display the attributes of variable ADDRESS:

```
desc address
```

- □ Display the attributes of array element ARR{*i* + *j*}:

```
desc arr{i+j}
```

ENTER

Assigns one or more debugger commands to the ENTER key

Category: Tailoring the Debugger

Syntax

ENTER <*command-1* <. . . ; *command-n*>>

Arguments

command
specifies a debugger command.
Default: STEP 1

Details

The ENTER command assigns one or more debugger commands to the ENTER key. Assigning a new command to the ENTER key replaces the existing command assignment. If you assign more than one command, separate the commands with semicolons.

Examples

- □ Assign the command STEP 5 to the ENTER key:

```
enter st 5
```

- □ Assign the commands EXAMINE and DESCRIBE, both for the variable CITY, to the ENTER key:

```
enter ex city; desc city
```

EXAMINE

Displays the value of one or more variables

Category: Manipulating DATA Step Variables

Alias: E

Syntax

EXAMINE *variable-1* <*format-1*> <. . . *variable-n* <*format-n*>>

EXAMINE _ALL_ <*format*>

Arguments

variable
identifies a DATA step variable.

format
identifies a SAS format or a user-created format.

ALL
identifies all variables that are defined in the current DATA step.

Details

The EXAMINE command displays the value of one or more specified variables. The debugger displays the value using the format currently associated with the variable, unless you specify a different format.

Examples

- Display the values of variables N and STR:

```
ex n str
```

- Display the element *i* of the array TESTARR:

```
ex testarr{i}
```

- Display the elements *i*+1, *j**2, and *k*-3 of the array CRR:

```
ex crr{i+1}; ex crr{j*2}; ex crr{k-3}
```

- Display the SAS date variable T_DATE with the DATE7. format:

```
ex t_date date7.
```

- Display the values of all elements in array NEWARR:

```
ex newarr{*}
```

See Also

Command:

"DESCRIBE" on page 1210

GO

Starts or resumes execution of the DATA step

Category: Controlling Program Execution

Alias: G

Syntax

GO <*line-number* | *label*>

Without Arguments

If you omit arguments, GO resumes execution of the DATA step and executes its statements continuously until a breakpoint is encountered, until the value of a watched variable changes, or until the DATA step completes execution.

Arguments

line-number
: gives the number of a program line at which execution is to be suspended next.

label
: is a statement label. Execution is suspended at the statement following the statement label.

Details

The GO command starts or resumes execution of the DATA step. Execution continues until all observations have been read, a breakpoint specified in the GO command is reached, or a breakpoint set earlier with a BREAK command is reached.

Examples

- Resume executing the program and execute its statements continuously:

```
g
```

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

```
g 104
```

See Also

Commands:

"JUMP" on page 1214
"STEP" on page 1218

HELP

Displays information about debugger commands

Category: Controlling the Windows

Syntax

HELP

Without Arguments

The HELP command displays a directory of the debugger commands. Select a command name to view information about the syntax and usage of that command. You must enter the HELP command from a window command line, from a menu, or with a function key.

JUMP

Restarts execution of a suspended program

Category: Controlling Program Execution

Alias: J

Syntax

JUMP *line-number* | *label*

Arguments

line-number
indicates the number of a program line at which to restart the suspended program.

label
is a statement label. Execution resumes at the statement following the label.

Details

The JUMP command moves program execution to the specified location without executing intervening statements. After executing JUMP, you must restart execution with GO or STEP. You can jump to any executable statement in the DATA step.

CAUTION:
Do not use the JUMP command to jump to a statement inside a DO loop or to a label that is the target of a LINK-RETURN group. In such cases you bypass the controls set up at the beginning of the loop or in the LINK statement, and unexpected results can appear. △

JUMP is useful in two situations:

- when you want to bypass a section of code that is causing problems in order to concentrate on another section. In this case, use the JUMP command to move to a point in the DATA step after the problematic section.
- when you want to re-execute a series of statements that have caused problems. In this case, use JUMP to move to a point in the DATA step before the problematic statements and use the SET command to reset values of the relevant variables to the values they had at that point. Then re-execute those statements with STEP or GO.

Examples

- Jump to line 5: `j 5`

See Also

Commands:

"GO" on page 1213
"STEP" on page 1218

LIST

Displays all occurrences of the item that is listed in the argument

Category: Manipulating Debugging Requests

Alias: L

Syntax

LIST _ALL_ | BREAK | DATASETS | FILES | INFILES | WATCH

Arguments

ALL
displays the values of all items.

BREAK
displays breakpoints.

Alias: B

DATASETS
displays all SAS data sets used by the current DATA step.

FILES
displays all external files to which the current DATA step writes.

INFILES
displays all external files from which the current DATA step reads.

WATCH
displays watched variables.

Alias: W

Examples

- List all breakpoints, SAS data sets, external files, and watched variables for the current DATA step:

```
l _all_
```

- List all breakpoints in the current DATA step:

```
l b
```

See Also

Commands:

"BREAK" on page 1206
"DELETE" on page 1209
"WATCH" on page 1220

QUIT

Terminates a debugger session

Category: Terminating the Debugger

Alias: Q

Syntax

QUIT

Without Arguments

The QUIT command terminates a debugger session and returns control to the SAS session.

Details

SAS creates data sets built by the DATA step that you are debugging. However, when you use QUIT to exit the debugger, SAS does not add the current observation to the data set.

You can use the QUIT command at any time during a debugger session. After you end the debugger session, you must resubmit the DATA step with the DEBUG option to begin a new debugging session; you cannot resume a session after you have ended it.

SET

Assigns a new value to a specified variable

Category: Manipulating DATA Step Variables

Alias: None

Syntax

SET *variable=expression*

Arguments

variable
specifies the name of a DATA step variable or an array reference.

expression
is any debugger expression.

Tip: *Expression* can contain the variable name that is used on the left side of the equal sign. When a variable appears on both sides of the equal sign, the debugger uses the original value on the right side to evaluate the expression and stores the result in the variable on the left.

Details

The SET command assigns a value to a 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.

Examples

- Set the variable A to the value of 3:

```
set a=3
```

- Assign to the variable B the value `12345` concatenated with the previous value of B:

```
set b='12345' || b
```

- Set array element ARR{1} to the result of the expression a+3:

```
set arr{1}=a+3
```

- Set array element CRR{1,2,3} to the result of the expression crr{1,1,2} + crr{1,1,3}:

```
set crr{1,2,3} = crr{1,1,2} + crr{1,1,3}
```

- Set variable A to the result of the expression a+c*3:

```
set a=a+c*3
```

STEP

Executes statements one at a time in the active program

Category: Controlling Program Execution
Alias: ST

Syntax

STEP <*n*>

Without Arguments

STEP executes one statement.

Arguments

n
specifies the number of statements to execute.

Details

The STEP command executes statements in the DATA step, starting with the statement at which execution was suspended.

When you issue a STEP command, the debugger:

- executes the number of statements that you specify
- displays the line number
- returns control to the user and displays the > prompt.

Note: By default, you can execute the STEP command by pressing the ENTER key. △

See Also

Commands:
"GO" on page 1213
"JUMP" on page 1214

SWAP

Switches control between the SOURCE window and the LOG window

Category: Controlling the Windows

Alias: None

Syntax

SWAP

Without Arguments

The SWAP command switches control between the LOG window and the SOURCE window when the debugger is running. When you begin a debugging session, the LOG window becomes active by default. While the DATA step is still being executed, the SWAP command enables you to switch control between the SOURCE and LOG window so that you can scroll and view the text of the program and also continue monitoring the program execution. You must enter the SWAP command from a window command line, from a menu, or with a function key.

TRACE

Controls whether the debugger displays a continuous record of the DATA step execution

Category: Manipulating Debugging Requests

Alias: T

Default: OFF

Syntax

TRACE <ON | OFF>

Without Arguments

TRACE displays the current status of the TRACE command.

Arguments

ON
: prepares for the debugger to display a continuous record of DATA step execution. The next statement that resumes DATA step execution (such as GO) records all actions taken during DATA step execution in the DEBUGGER LOG window.

OFF
: stops the display.

Examples

- Determine whether TRACE is ON or OFF:

```
trace
```

- Prepare to display a record of debugger execution:

```
trace on
```

WATCH

Suspends execution when the value of a specified variable changes

Category: Manipulating Debugging Requests

Alias: W

Syntax

WATCH *variable(s)*

Arguments

variable
specifies a DATA step variable.

Details

The WATCH command specifies a variable to monitor and suspends program execution when its value changes.

Each time the value of a watched variable changes, the debugger:

- suspends execution
- displays the line number where execution has been suspended
- displays the variable's old value
- displays the variable's new value
- returns control to the user and displays the > prompt.

Examples

- Monitor the variable DIVISOR for value changes:

```
w divisor
```

The correct bibliographic citation for this manual is as follows: SAS Institute Inc., *SAS® Language Reference, Version 8*, Cary, NC: SAS Institute Inc., 1999.

SAS® Language Reference, Version 8

Copyright © 1999 by SAS Institute Inc., Cary, NC, USA.

ISBN 1–58025–369–5

All rights reserved. Printed in the United States of America. No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form or by any means, electronic, mechanical, photocopying, or otherwise, without the prior written permission of the publisher, SAS Institute Inc.

U.S. Government Restricted Rights Notice. Use, duplication, or disclosure of the software and related documentation by the U.S. government is subject to the Agreement with SAS Institute and the restrictions set forth in FAR 52.227–19 Commercial Computer Software-Restricted Rights (June 1987).

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

1st printing, October 1999

SAS® and all other SAS Institute Inc. product or service names are registered trademarks or trademarks of SAS Institute Inc. in the USA and other countries.® indicates USA registration.

Other brand and product names are registered trademarks or trademarks of their respective companies.

The Institute is a private company devoted to the support and further development of its software and related services.