SAS® Code Debugger 8.1 and SAS® DATA Step Debugger: User's Guide

SAS® Documentation
November 15, 2019

The correct bibliographic citation for this manual is as follows: SAS Institute Inc. 2019. *SAS® Code Debugger 8.1 and SAS® DATA Step Debugger: User's Guide*. Cary, NC: SAS Institute Inc.

SAS® Code Debugger 8.1 and SAS® DATA Step Debugger: User's Guide

SAS Institute Inc., SAS Campus Drive, Cary, NC 27513-2414

November 2019

8.1-P1:cmpdebugug

Contents

PART 1

SAS Code Debugger

1

Getting Started with SAS Code Debugger

Introduction to SAS Code Debugger

Debugging is the process of identifying and removing logic errors from a program. Unlike syntax errors, logic errors do not stop a program from running. Instead, logic errors cause the program to produce unexpected results.

To debug SAS code, typically, you insert PUT statements at selected places in the code, submit the code, and examine the values that are displayed in the SAS log. This debugging process could cause you to spend considerable time and effort locating the problem.

SAS Code Debugger provides a user-friendly, graphical user interface that streamlines the debugging process by enabling you to perform the following tasks:

- interactively monitor the execution of your PROC FCMP, PROC COMPILE, and PROC HPRISK code.
- start, pause, and resume execution of your program
- execute statements one by one or in groups
- step into, step over, or step out of functions, subroutines, and methods
- set breakpoints that interrupt an executing program
- monitor the values of selected variables
- assign new values to variables

SAS Code Debugger supports debugging on only one node and one thread. The debugger also supports only SAS numeric and fixed character types and arrays of those types.

You cannot use SAS Code Debugger to debug objects such as dictionaries, analytic stores, or binary files. You also cannot debug a line of code that initializes an array.

Connect Your SAS Session to SAS Code Debugger

Debugging SAS code requires two user sessions: a SAS session and a debugger session. A SAS session is created when you sign in to the application, such as SAS Studio, that you plan to use to execute the code that you want to debug. A debugger session is created when you sign in to SAS Code Debugger.

To create a debugger session and connect it to your SAS session:

1. Enter the URL for SAS Code Debugger in a web browser (for example, `http://host.example.com/SASCodeDebugger/`). Contact your administrator to obtain this URL.
2. Sign in to SAS Code Debugger to create your debugger session.
3. In the **Code** panel in SAS Code Debugger, click **Copy to Clipboard** to copy the DEBUGOPT OPTIONS statement that connects your debugger session to your SAS session. If the button is not visible, click in the debugger toolbar, and then click **Copy** in the Debug OPTIONS String window.
4. Paste the OPTIONS statement into the code in your SAS session (for example, a SAS session in SAS Studio). The OPTIONS statement must precede the code that you want to debug.

   You can use the same OPTIONS statement in your SAS session until the debugger session times out or until you terminate the debugger session.
5. Execute your code. When SAS encounters code that can be debugged, SAS attempts to compile the program. If the program compiles successfully, SAS does the following:
   1. enters debug mode
   2. suspends execution at the first executable line in your program
   3. loads your program into the **Code** panel in SAS Code Debugger

4 waits for you to invoke an action or submit a command

CAUTION! If your code contains syntax errors, SAS Code Debugger might not behave as expected. For example, if your code is missing a semicolon, the debugger does not load the code.

Customizing the User Interface

User Interface Layout

SAS Code Debugger provides an interactive debugging environment that consists of a toolbar and the following panels:

- **Breakpoints**: Lists all the breakpoints that exist in the current debugger session. For more information about breakpoints, see Chapter 2, "Managing Breakpoints," on page 13.
- **Call Stack**: Shows the layers of invocations that brought the code to the current point. The layers are listed in descending order according to the order in which the layers were called.
- **Code**: Displays either the OPTIONS statement or the code to be debugged. If your code is displayed, the following sections are also provided:
  - **Functions and Methods**: Lists the functions and methods that are included in your code, along with any functions or methods that are included in the same library as your functions and methods.
  - Tab list: Contains a separate tab for the main code (default) and for each function or method that you open.
- **Console**: Provides a **Command line** and a running log for your debugger session. For a list of the supported commands, see "Debugger Commands by Category".
- **Variables**: Enables you to examine your variables, to create watch variables, and to change the values of variables as you step through the code. For more information, see Chapter 3, "Working with Variables," on page 19.
- **Watch**: Provides a quick and easy way for you to monitor the values of particular variables.

You can customize the user interface by resizing panels, maximizing and minimizing panels, rearranging panels, and removing and adding panels.

Resize Panels

Complete one of the following steps:

- Click and drag the separator that is between the panels.

- Tab to the separator that is between the panels, and use the arrow keys to resize the panel.

Maximize and Minimize Panels

Complete one of the following steps:

- To maximize a panel, click ⋮ in the toolbar for that panel and select **Maximize**. The debugger maximizes the selected panel, and displays the minimized panels as buttons at the bottom of the window.

  To unmaximize a panel, click ⋮ in the toolbar for the maximized panel and select **Restore down**.

- To minimize a panel, click ⋮ in the toolbar for that panel and select **Minimize**. The debugger displays the minimized panel as a button at the bottom of the window. Click the button to re-display the panel.

Rearrange Panels

To rearrange the panels, click and drag the panel title area to a new drop zone, which is identified by a blue horizontal or vertical bar.

Remove Panels

Complete one of the following steps:

- Click ⋮ in the toolbar for the panel that you want to remove, and select **Remove**. If a confirmation message appears, click **Remove** to confirm that you want to remove the panel.
- Click the **Panels** menu in the debugger toolbar, and select a panel whose name is preceded by a ✓. If a confirmation message appears, click **Remove** to confirm that you want to remove the panel.

Add Panels

To re-display a panel that you hid from view, click the **Panels** menu in the debugger toolbar, and select a panel whose name is not preceded by a ✓.

Restore Default Panel Layout

To undo all of your changes to the layout and return to the system default layout, click the **Panels** menu in the debugger toolbar, and select **Restore defaults**.

Controlling Program Execution

With SAS Code Debugger, you can start, resume, and pause program execution. You can also step over, into, or out of functions and methods, and you can terminate a debugger session.

Start or Resume Program Execution

Complete one of the following steps:

- Click ▷ in the debugger toolbar.
- Press the F8 key.
- Issue the GO command.

The debugger executes the code until the debugger encounters a breakpoint, until you pause program execution, or until the debugger reaches the end of the code.

See Also

"GO Command" on page 35

Step over a Function or Method

Complete one of the following steps:

- Click in the debugger toolbar.
- Press the F9 key.
- Issue the STEP command.

The debugger executes the current line of code. If this line contains a function call or a method call, the debugger executes the entire function or method, and then suspends program execution.

See Also

"STEP Command" on page 39

Step into a Function or Method

Complete one of the following steps:

- Click in the debugger toolbar.
- Press the F10 key.

- Issue the following command:

```
step -t
```

The debugger executes the current line of code. If this line contains a function call or a method call, the debugger steps into the function or method and suspends execution at the first executable line of code in that function or method.

See Also

"STEP Command" on page 39

Step out of a Function or Method

Complete one of the following steps:

- Click in the debugger toolbar.
- Press the Ctrl + F10 keys.
- Issue the following command:

```
go -o
```

The debugger executes the current line of code. If this line is inside a function or method, the debugger executes the current function or method, and then suspends program execution after the return of that function or method.

See Also

"GO Command" on page 35

Pause Program Execution

Complete one of the following steps:

- Click in the debugger toolbar.
- Press the F8 key.
- Issue the PAUSE command.

The debugger pauses program execution as soon as possible. The icon indicates the line of code at which program execution was suspended.

See Also

"PAUSE Command" on page 36

Terminate a Debugger Session

Complete one of the following steps:

- Issue the QUIT command.
- Sign out of SAS Code Debugger.
- Close the browser window or tab.

SAS Code Debugger terminates the debugger session and returns control to the SAS session. The SAS session finishes executing the program.

Note: The debugger has a time-out interval that is permanently set to two hours. If the debugger has completed the current instruction and is waiting for more than two hours for you to provide the next instruction, the debugger times out and automatically terminates the debugger session.

See Also

"QUIT Command" on page 37

Debugging a Sample Program

This section walks you step-by-step through the debugging process for a sample PROC FCMP program.

Debug the Sample Program

1. Sign in to the application that you want to use to execute the sample PROC FCMP program. For example, you can use SAS Studio to execute the code.
2. Execute the sample code. The results are as follows:
   - myM = 1269.05
   - myS = 385.28237588

   When you manually calculate the mean and the standard deviation, the values are as follows:
   - The mean is 1269.05.
   - The standard deviation is 422.0556966.

   The mean is correct, but there is an error with the standard deviation.
3. Sign in to SAS Code Debugger so that you can debug the program.
4. In the **Code** panel, copy the OPTIONS statement to the clipboard.
5. In the sample program, replace the following code with the OPTIONS statement that you copied:

   ```
   /*<insert_options_statement>*/
   ```
6. Execute the sample program again. The program loads in SAS Code Debugger, and the debugger suspends execution at the first executable line in the program.

7 In the **Functions and Methods** section in the **Code** panel, expand the **Functions** node. The sample program contains the following functions:

- **myMean**: Calculates the mean.
- **mySTD**: Calculates the standard deviation.

8 The program is generating an incorrect standard deviation, so it might be helpful to set breakpoints in the **mySTD** function. Complete the following steps:

a Double-click **main.mySTD** to display the function in a separate tab.

b Lines 5, 7, and 8 in the function calculate a portion of the standard deviation, so set a breakpoint for each of those lines of code. Click the margin that precedes lines 5, 7, and 8.

A breakpoint (●) icon appears. This icon indicates that a breakpoint is enabled for the corresponding line of code.

The breakpoints are also added to the list in the **Breakpoints** panel with the syntax `<package_name>.<function_name>:<line_number>`. Each breakpoint is preceded by a check mark, which indicates that the breakpoints are enabled.

9 Make the variable SD a watch variable so that you can monitor its value as you step through the code. Click **+** in the **Watch** panel. Then, type `sd` in the **Variable name** field and click **Save**. The variable name is grayed out because the variable is not in scope at the current point in the program.

10 Click ▷ in the debugger toolbar or press the F8 key to start executing the program. The debugger stops executing at your first breakpoint. At this point, the value of the variable SD is 0.

11 Click in the debugger toolbar or press the F10 key six times to step through the DO loop. The following table lists the values for the variable SD as you step through the loop.

These values match the results that were calculated manually, so this line of code is yielding the correct results.

Table 1.1 Values of Variable "sd" for the DO Loop

Step	Sample Program Results	Manually Calculated Results
1	300139.6225	300139.6225
2	368025.925	368025.925
3	398319.3275	398319.3275
4	408979.89	408979.89
5	458019.9925	458019.9925
6	890655.055	890655.055

12 Debug the line of code that is at the next breakpoint. Click . The value of the variable SD is 148442.5092. This value does not match the manually calculated

value of 178131.011. This mismatch indicates that there is an error on the following line of code:

```
sd = sd / theDim;
```

To calculate the standard deviation, this line of code should divide the output of Step 11 by `(theDim - 1)`.

13 Now that you have identified the problem, stop debugging the program. Enter the following command in the **Command line** in the **Console** panel:

```
quit
```

14 Complete the following steps:

a In the sample program, find the following line of code:

```
sd = sd / theDim;
```

b Change that line of code to this code:

```
sd = sd / (theDim - 1);
```

c Comment out the OPTIONS statement at the top of the program.

15 Execute the updated sample program. Both the mean and the standard deviation have the correct values.

- myM = 1269.05
- myS = 422.05569656

You have successfully debugged the sample program.

Sample PROC FCMP Program

This sample program calculates the mean and the standard deviation for a set of metabolic rates for birds. An error has been intentionally added to the program. Use SAS Code Debugger to identify the error. For instructions, see "Debug the Sample Program" on page 9.

```
/*<insert_options_statement>*/

ods listing file=_dataout;

proc fcmp;

function myMean(v[*]);
   theDim = DIM(v);
   if (theDim < 1) then
      return(.);

   s = 0;

   do i=1 to theDim;
      s = s + v[i];
   end;

   theMean = s/theDim;
```

```
   return(theMean);
endfunc;

function mySTD(v[*], m);
   theDim = DIM(v);
   sd = 0;

   do i = 1 to theDim;
      sd = sd + (v[i] - m) ** 2;
   end;

   sd = sd / theDim;
   sd = sqrt(sd);
   return(sd);
endfunc;

array metabolicRate[6]  (721.2,  1008.5, 1095.0,
                         1372.3, 1490.5, 1926.8);

myM = .;
myS = .;

myM = myMean(metabolicRate);
myS = mySTD(metabolicRate, myM);

put myM=;
put myS=;
quit;

%let _DATAOUT_NAME=results.txt;
```

2

Managing Breakpoints

About Breakpoints

A breakpoint identifies a point in the code at which the debugger stops executing the program so that you can perform the following actions:

- examine the current variables
- change the value of a variable
- add watch variables
- issue commands
- review the call stack

The debugger suspends program execution only on executable lines of code. Format statements, array definitions, empty lines, and comments, for example, are not valid locations for breakpoints.

In SAS Code Debugger, you can add, disable, enable, and remove breakpoints. You can also export and import breakpoints.

Add Breakpoints

Do one of the following:

- In the **Functions and Methods** section in the **Code** panel, complete one of the following steps:
  - Select one or more functions and methods. Then, right-click on the selection, and select **Add breakpoint** from the context menu.
  - Select one or more functions and methods. Then, click ⋮, and select **Add breakpoint to selected functions and methods**.
  - Click ⋮, and select **Add breakpoint to all functions and methods**, **Add breakpoint to all functions**, or **Add breakpoint to all methods**.
  - Double click the name of a function or method to display the code in a separate tab. Then, click the left margin for the line number where you want to add a breakpoint. You can also right-click the left margin, and select **Add breakpoint** from the context menu.
- In the **Command line** field in the **Console** panel, issue the BREAK command or the BREAKROUTINES command.

When you add a breakpoint, the following changes appear in the UI:

- A breakpoint (●) icon precedes the line of code for which you added a breakpoint. This icon indicates that the breakpoint is enabled.
- The breakpoint is added to the list in the **Breakpoints** panel, and the name of the breakpoint is preceded by a check mark. The check mark indicates that the breakpoint is enabled.

  The name of the breakpoint has the following syntax:

  - functions: `<package_name>.<function_name>:<line_number>`
  - methods: `main.<method_name>:<line_number>`

  Note: The line number might not appear until you open the function or method in a tab in the **Code** panel.
- A message in the **Console** panel indicates that the breakpoint was added.

See Also

- “BREAK Command” on page 27
- “BREAKROUTINES Command” on page 32

Disable Breakpoints

Do one of the following:

- In the tab list section in the **Code** panel, right-click ●, and select **Disable breakpoint** from the context menu. The icon changes to ○.
- In the **Breakpoints** panel, do one of the following:
  - Deselect the check box that precedes the name of an enabled breakpoint.
  - Right-click an enabled breakpoint, and select **Disable** from the context menu.
  - Select one or more enabled breakpoints. Then, click ⋮ in the toolbar for the **Breakpoints** panel, and select **Disable selected breakpoints**.
  - Click ⋮ in the toolbar for the **Breakpoints** panel, and select **Disable all breakpoints**.
- In the **Command line** field in the **Console** panel, issue the BREAKDISABLE command.

See Also

“BREAKDISABLE Command” on page 30

Enable Breakpoints

Do one of the following:

- In the tab list section in the **Code** panel, right-click ○, and select **Enable breakpoint** from the context menu. The icon changes to ●.
- In the **Breakpoints** panel, do one of the following:
  - Select the check box that precedes a disabled breakpoint.
  - Right-click a disabled breakpoint, and select **Enable** from the context menu.
  - Select one or more disabled breakpoints. Then, click ⋮ in the toolbar for the **Breakpoints** panel, and select **Enable selected breakpoints**.
  - Click ⋮ in the toolbar for the **Breakpoints** panel, and select **Enable all breakpoints**.
- In the **Command line** field in the **Console** panel, issue the BREAKENABLE command.

See Also

"BREAKENABLE Command" on page 31

Remove Breakpoints

Do one of the following:

- In the tab list section in the **Code** panel, do one of the following:
  - Click ● or ○.
  - Right-click ● or ○, and select **Remove breakpoint** from the context menu.
- In the **Breakpoints** panel, do one of the following:
  - Right-click a breakpoint, and select **Remove** or **Remove all** from the context menu.
  - Select one or more breakpoints. Then, click ⋮ in the toolbar for the **Breakpoints** panel, and select **Remove selected breakpoints**.
  - Click ⋮ in the toolbar for the **Breakpoints** panel, and select **Remove all breakpoints**.
- In the **Command line** field in the **Console** panel, issue the BREAKCLEAR command.

See Also

"BREAKCLEAR Command" on page 29

Export Breakpoints

1. Click ⋮ in the debugger toolbar, and select **Export**. The Export window appears.
2. In the **File name** field, enter the name to use for the exported file. Omit the file extension.
3. In the **Export** field, ensure that the **Include breakpoints** option is selected. If you want to include your breakpoints and watch variables in the same export file, ensure that the **Include watch variables** option is also selected.

   Note: The debugger displays both options even if you have not defined any breakpoints or watch variables. The debugger does not validate your selections.
4. Click **Export** to export the selected items to a file named <*filename*>.dbgcfg.

Import Breakpoints

1. Click ⋮ in the debugger toolbar, and select **Import**. The Import window appears.
2. In the **File to import** field, click **Browse** and select the file that contains the breakpoints that you want to import. The debugger supports only the .dbgcfg file extension.
3. In the **Import** field, ensure that the **Include breakpoints** option is selected.

   If the selected file also contains watch variables, you can import your watch variables along with your breakpoints. Ensure that the **Include watch variables** option is also selected.

   Note: The debugger displays both options even if the selected file does not contain any breakpoints or watch variables. The debugger does not validate your selections.
4. Click **Import**. The debugger imports the selected items if those items exist in the specified file.

3

Working with Variables

About Variables

Variables are an essential part of the debugging process. You can use variables to identify data and logic errors and to pinpoint the lines of code that might be producing unexpected results.

To examine your variables as you step through the code, use the **Variables** panel in SAS Code Debugger. The **Variables** panel contains the following tabs:

- The **Local** tab lists all the variables that are in scope at the current point in the program, including the variables that are referenced in your methods.
- The **Global** tab lists all the global variables that have been initialized so far in the program.

With SAS Code Debugger, you can change the values of variables; configure and inspect array variables; and, add, remove, export, and import watch variables.

Change the Value of a Variable

SAS Code Debugger enables you to change the values of variables while you are debugging a program. You might change the values of variables to see how the new values impact the results or to resume program execution at a specific point in a loop.

To change the value of a variable:

1. Suspend program execution, or wait until the debugger reaches a breakpoint.
2. In the **Variables** panel, click the cell in the **Value** column that corresponds with the variable whose value you want to change. For editable variables, an editable field appears in the cell. For immutable variables, no field appears.
3. If the variable is editable, enter the new value.

   If the variable has a character data type, the length of the new value must be less than or equal to the length that is specified in the Type column [Char (`length`)].
4. To save the new value, press the Enter key or click elsewhere in the user interface. If the new value is invalid, the debugger displays a message and reverts to the previous value.

   TIP If you have not saved the new value and you want to cancel the update, press the Esc key. The debugger reverts to the previous value.

See Also

"SETV Command" on page 38

Specify the Number of Array Values to Display

To specify the number of values that the debugger can display for your array variables, do one of the following:

- If you are using a stand-alone instance of SAS Code Debugger, complete the following steps:
  1. Click the application options button, which is the last button in the banner.
  2. Click **Settings**. The Settings window appears.
  3. Expand **SAS Code Debugger**, and click **Preferences**.

4. In the **Maximum number of array values to display** field, enter an integer that is greater than zero.

   Note: This is a global setting. It is preserved across debugger sessions.

5. Click **Close**. The debugger displays no more than the maximum number of array values for each array variable. The debugger hides any remaining array values.

- If you are using an instance of SAS Code Debugger that is embedded in another application, complete the following steps:

  1. Click ⚙. The **SAS Code Debugger Settings** window appears.
  2. In the **Maximum number of array values to display** field, enter an integer that is greater than zero.

     Note: This is a global setting. It is preserved across debugger sessions.

  3. Click **Close**. The debugger displays no more than the maximum number of array values for each array variable. The debugger hides any remaining array values.

See Also

- "Inspect an Array Variable" on page 21
- "SET Command" on page 37

Inspect an Array Variable

If an array variable has numerous values, you can use the **Inspect array** action to explore the array in a separate window and to filter the array by index.

To inspect an array:

1. In the **Variables** panel, right-click an array variable, and select **Inspect array** from the context menu. The Inspect Array window appears.
2. To filter the array values by index, place your cursor over the array variable, and click ▽. The Filter Array window appears.
3. In the **Start index** and **End index** fields, enter the index range that you want to inspect. The values must be positive integers. The end index must be greater than the start index, and the end index must be less than or equal to the total number of array values.

   Note:

   - The total number of array values is provided in the **Type** column for the array variable (`<data_type> [<total_values>]`).
   - If the array contains more values than is specified for the **Maximum number of array values to display** setting, specify an index range that enables you to inspect the hidden array values.

4 Click **Filter**. The array displays only the specified index range.

5 Click **Close** when you are finished inspecting the array.

See Also

- "Specify the Number of Array Values to Display" on page 20
- "DISPLAY Command" on page 33

Add Watch Variables

SAS Code Debugger enables you to add variables to the **Watch** panel so that you can easily monitor the values of those variables as you step through the code.

The **Watch** panel always lists all the variables that you are watching regardless of scope. If a watch variable is out of scope, the debugger grays out that watch variable.

To add a watch variable, do one of the following:

- In the **Variables** panel, do one of the following:
  - Right-click a variable, and select **Add to watch list** from the context menu.
  - Select one or more variables. Then, click ⋮ in the toolbar for the **Variables** panel, and select **Add selected variables to watch list**.
- In the **Watch** panel, complete the following steps:

  1 Click +. The New Watch Variable window appears.

  2 In the **Variable name** field, enter the name of the variable to be watched.

  3 Click **Save** to add the watch variable.

Remove Watch Variables

In the **Watch** panel, do one of the following:

- Select one or more watch variables, and click 🗑.
- Right-click a watch variable, and select **Remove** or **Remove all** from the context menu.
- Select one or more watch variables. Then, click ⋮ in the toolbar for the **Watch** panel, and select **Remove selected watch variables**.
- Click ⋮ in the toolbar for the **Watch** panel, and select **Remove all watch variables**.

Export Watch Variables

1. Click ⋮ in the debugger toolbar, and select **Export**. The Export window appears.
2. In the **File name** field, enter the name to use for the exported file. Omit the file extension.
3. In the **Export** field, ensure that the **Include watch variables** option is selected. If you want to include your breakpoints and watch variables in the same export file, ensure that the **Include breakpoints** option is also selected.

   Note: The debugger displays both options even if you have not defined any breakpoints or watch variables. The debugger does not validate your selections.
4. Click **Export** to export the selected items to a file named <*filename*>.dbgcfg.

Import Watch Variables

1. Click ⋮ in the debugger toolbar, and select **Import**. The Import window appears.
2. In the **File to import** field, click **Browse** and select the file that contains the watch variables that you want to import. The debugger supports only the .dbgcfg file extension.
3. In the **Import** field, ensure that the **Include watch variables** option is selected.

   If the selected file also contains breakpoints, you can import your breakpoints along with your watch variables. Ensure that the **Include breakpoints** option is also selected.

   Note: The debugger displays both options even if the selected file does not contain any breakpoints or watch variables. The debugger does not validate your selections.
4. Click **Import**. The debugger imports the selected items if those items exist in the specified file.

4

Issuing Commands

Debugger Commands Overview

Debugger Commands Overview

SAS Code Debugger provides commands that correspond with many of the point-and-click actions that are provided in the user interface.

The supported commands are grouped into the following categories:

- **Controlling Program Execution**: These commands enable you to start, resume, pause, or stop program execution. They also enable you to step through your program.

- **Manipulating Debugging Requests**: These commands enable you to control when the debugger suspends program execution.
- **Manipulating Variables**: These commands enable you to examine the values and attributes of your variables and to change the values of your variables.
- **Terminating the Debugger**: This command enables you to terminate the debugger session.

To issue commands, use the **Command line** field, which is located at the bottom of the **Console** panel. In the **Command line** field, you can do the following:

- specify a single command that occupies a single line.
- press the Enter key to step to the next executable statement. This action is equivalent to issuing the STEP command.
- use the up and down arrow keys to scroll through the commands that you previously entered.
- type `clear` to remove the contents from the running log that is provided in the **Console** panel.

Debugger Commands by Category

Category	Language Elements	Description
Controlling Program Execution	GO Command (p. 35)	Starts or resumes execution of a program.
	PAUSE Command (p. 36)	Pauses the execution of a program.
	STEP Command (p. 39)	Executes the current line of code.
Manipulating Debugging Requests	BREAK Command (p. 27)	Sets a breakpoint at an executable statement.
	BREAKCLEAR Command (p. 29)	Removes an existing breakpoint.
	BREAKDISABLE Command (p. 30)	Disables a breakpoint that is currently enabled.
	BREAKENABLE Command (p. 31)	Enables a breakpoint that is currently disabled.
	BREAKLIST Command (p. 32)	Lists all the breakpoints that exist in a debugger session.
	BREAKROUTINES Command (p. 32)	Adds breakpoints to your functions, subroutines, or methods.
	ENV Command (p. 34)	Displays the information for a stack level.
	GETROUTINES Command (p. 35)	Returns a list of your functions, subroutines, and methods.

Category	Language Elements	Description
	WHERE Command (p. 41)	Lists the stack levels that are in the current call stack.
Manipulating Variables	ARGS Command (p. 27)	Lists all the variables that are passed as parameters into the current function, subroutine, or method.
	DISPLAY Command (p. 33)	Retrieves the value of a variable.
	SET Command (p. 37)	Sets the maximum number of values to display for an array variable.
	SETV Command (p. 38)	Changes the value of a variable.
	VARS Command (p. 39)	Retrieves all the variables and their values for a given scope.
	WHATIS Command (p. 40)	Displays the Type attribute and the Editable attribute for a variable.
Terminating the Debugger	QUIT Command (p. 37)	Terminates a debugger session.

Dictionary

ARGS Command

Lists all the variables that are passed as parameters into the current function, subroutine, or method.

Category: Manipulating Variables

Syntax

args

Details

The ARGS command lists all the variables that are passed as parameters into the current function, subroutine, or method.

BREAK Command

Sets a breakpoint at an executable statement.

Category: Manipulating Debugging Requests

Syntax

break -n *statement_number* | -f *function_method* <-p *package_name*> <-b>

Without Arguments

You cannot specify a BREAK command without any arguments. A valid BREAK command must contain at least the `-n statement_number` argument, the `-f function_method` argument, or both arguments.

Required Arguments

-n *statement_number*
specifies the statement number at which to set a breakpoint. If you specify a statement number that does not correspond with an executable line of code, the debugger sets the breakpoint at the next executable line of code that follows the specified statement number.

Note: The statement number might differ from the line number that is displayed in the **Code** panel. Scroll through the log in the **Console** panel to locate the statement number that corresponds with the executable statement in which you are interested.

-f *function_method*
specifies the name of the function or method at which to set a breakpoint.

Note: The ▶ icon indicates the line of code at which program execution was suspended. If this icon is not visible in the function or method where you want to add the breakpoint, you must specify the `-f function_method` argument in order for the debugger to set the breakpoint in the expected location.

Optional Arguments

-p *package_name*
provides the name of the package that contains the specified function or method. This argument is valid only if the `-f function_method` argument is also specified.

-b
separates multiple breakpoints in the BREAK command.

Details

The BREAK command sets a breakpoint at the specified statement number. Only one breakpoint is allowed at a given location.

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

Example

- Set a breakpoint at statement 5 in the current function:

```
break -n 5
```

- Set a breakpoint at statement 23 in the TEST function that is included in the Main package:

```
break -n 23 -f test -p main
```

- Set a breakpoint at statements 4 and 5 in the current function and at the first executable statement in the FOO function:

```
break -n 4 -b -n 5 -b -f foo
```

See Also

- "Add Breakpoints" on page 14
- "BREAKROUTINES Command" on page 32
- "BREAKCLEAR Command" on page 29
- "BREAKDISABLE Command" on page 30
- "BREAKENABLE Command" on page 31
- "BREAKLIST Command" on page 32

BREAKCLEAR Command

Removes an existing breakpoint.

Category: Manipulating Debugging Requests

Syntax

breakclear -a | -l *break_ID*

Required Arguments

-a
removes all the breakpoints that exist in the current debugger session.

-l *break_ID*
specifies the ID of the breakpoint that you want to remove.

Details

The BREAKCLEAR command removes a previously created breakpoint. When you remove a breakpoint, subsequent STEP or GO commands do not stop at that program location.

TIP Use the BREAKLIST command to retrieve the IDs for the breakpoints that exist in the current debugger session.

Example

- Remove the breakpoint with ID 3:

```
breakclear -l 3
```

- Remove all breakpoints:

```
breakclear -a
```

See Also

- "Remove Breakpoints" on page 16
- "BREAK Command" on page 27
- "BREAKLIST Command" on page 32

BREAKDISABLE Command

Disables a breakpoint that is currently enabled.

Category: Manipulating Debugging Requests

Syntax

breakdisable -a | -l *break_ID*

Required Arguments

-a
disables all the breakpoints that exist in the current debugger session.

-l *break_ID*
specifies the ID of the breakpoint that you want to disable.

Details

The BREAKDISABLE command disables a breakpoint that is currently enabled. When you disable a breakpoint, subsequent STEP or GO commands do not stop at that program location.

TIP Use the BREAKLIST command to retrieve the IDs for the breakpoints that exist in the current debugger session.

Example

- Disable the breakpoint with ID 3:

```
breakdisable -l 3
```

- Disable all breakpoints:

```
breakdisable -a
```

See Also

- "Disable Breakpoints" on page 15
- "BREAK Command" on page 27
- "BREAKENABLE Command" on page 31
- "BREAKLIST Command" on page 32

BREAKENABLE Command

Enables a breakpoint that is currently disabled.

Category: Manipulating Debugging Requests

Syntax

breakenable -a | -l *break_ID*

Required Arguments

-a
enables all the breakpoints that exist in the current debugger session.

-l *break_ID*
specifies the ID of the breakpoint that you want to enable.

Details

The BREAKENABLE command enables a breakpoint that is currently disabled. When you enable a breakpoint, subsequent STEP or GO commands suspend program execution at that program location.

TIP Use the BREAKLIST command to retrieve the IDs for the breakpoints that exist in the current debugger session.

Example

- Enable the breakpoint with ID 3:

```
breakenable -l 3
```

- Enable all breakpoints:

```
breakenable -a
```

See Also

- "Enable Breakpoints" on page 15
- "BREAK Command" on page 27
- "BREAKLIST Command" on page 32

BREAKLIST Command

Lists all the breakpoints that exist in a debugger session.

Category: Manipulating Debugging Requests

Syntax

breaklist

Details

The BREAKLIST command displays information about the breakpoints that are currently defined in your debugger session.

BREAKROUTINES Command

Adds breakpoints to your functions, subroutines, or methods.

Category: Manipulating Debugging Requests

Syntax

breakroutines < | >

Without Arguments

If you do not specify an argument, the BREAKROUTINES command adds breakpoints to all the functions, subroutines, and methods that are included in your code.

Optional Arguments

-f
adds breakpoints to all the functions and subroutines that are included in your code.

-m
adds breakpoints to all the methods that are included in your code.

Example

- Add breakpoints to all the methods

```
breakroutines -m
```

- Add breakpoints to all the functions and subroutines:

```
breakroutines -f
```

See Also

- "Add Breakpoints" on page 14
- "BREAK Command" on page 27

DISPLAY Command

Retrieves the value of a variable.

Category: Manipulating Variables

Syntax

display -v *variable_name* <-s *start_index* > < -e *end_index*>

Required Argument

-v *variable_name*
specifies the name of the variable whose information is to be retrieved.

Optional Arguments

-s *start_index*
specifies the first index to display for an array variable. The value must be a whole number that is less than the end index. The start of the index range is not inclusive.

If you specify a start index, you must also specify an end index.

-e *end_index*
specifies the last index to display for an array variable. The value must be a whole number that is greater than the start index. The end of the index range is inclusive.

If you specify an end index, you must also specify a start index.

Details

The DISPLAY command retrieves information about a variable, including the variable's type and value.

If the variable is an array variable, you can specify a start index and an end index to display the index range that you want to inspect. If you do not specify an index range, the default is to display the array from index 0 to the value that is specified for the **Maximum number of array values to display** setting.

If the array contains more values than is specified for the **Maximum number of array values to display** setting, you can specify an index range that enables you to inspect the hidden array values.

Note: If you specify an index range for a variable that is not an array variable, the index range is ignored.

Example

- Display the value of variable ABC:

```
display -v abc
```

- Display the index range 1–5 for array variable XYZ:

```
display -v xyz -s 0 -e 5
```

See Also

"Inspect an Array Variable" on page 21

ENV Command

Displays the information for a stack level.

Category: Manipulating Debugging Requests

Syntax

env <-l *stack_level*>

Without Arguments

If you do not specify an argument, the ENV command displays the information for the current stack level.

Optional Argument

-l *stack_level*
specifies the stack level for which to display information.

Example

- Display the information for stack level 5:

```
env -l 5
```

- Display the information for the current stack level:

```
env
```

GETROUTINES Command

Returns a list of your functions, subroutines, and methods.

Category: Manipulating Debugging Requests

Syntax

getroutines < | >

Without Arguments

If you do not specify an argument, the GETROUTINES command returns a list of all the functions, subroutines, and methods that are included in your code along with any functions, subroutines, and methods that are included in the same library as your functions, subroutines, and methods.

Optional Arguments

-f
returns a list of all the functions and subroutines that are included in your code along with any functions and subroutines that are included in the same library as your functions.

-m
returns a list of all the methods that are included in your code along with any methods that are included in the same library as your methods.

Example

- Retrieve a list of all the methods

```
getroutines -m
```

- Retrieve a list of all the functions and subroutines:

```
getroutines -f
```

GO Command

Starts or resumes execution of a program.

Category: Controlling Program Execution

Syntax

go <-1 *statement_number* | | *number_of_statements*>

Without Arguments

If you do not specify an argument on the GO command, the debugger executes the code until the debugger encounters a breakpoint, until you pause program execution, or until the debugger reaches the end of the code.

Optional Arguments

-l *statement_number*
specifies the statement number at which to suspend execution.

Note: The statement number might differ from the line number that is displayed in the **Code** panel. Scroll through the log in the **Console** panel to locate the statement number that corresponds with the executable statement in which you are interested.

-o
executes the current line of code. If this line is inside a function or method, the debugger executes the current function or method, and then suspends program execution after the return of that function or method.

-n *number_of_statements*
specifies the number of statements to execute before program execution is suspended.

Example

- Step out of the current function or method:

```
go -o
```

- Suspend program execution at statement 640:

```
go -l 640
```

- Suspend program execution after five statements have been executed:

```
go -n 5
```

See Also

- “Start or Resume Program Execution” on page 7
- “Step out of a Function or Method” on page 8

PAUSE Command

Pauses the execution of a program.

Category: Controlling Program Execution

Syntax

pause

Without Arguments

The PAUSE command pauses program execution as soon as possible. The ▶ icon indicates the line of code at which program execution was suspended.

See Also

"Pause Program Execution" on page 8

QUIT Command

Terminates a debugger session.

Category: Terminating the Debugger

Syntax

quit

Without Arguments

The QUIT command terminates the debugger session and returns control to the SAS session. The SAS session finishes executing the program.

See Also

"Terminate a Debugger Session" on page 8

SET Command

Sets the maximum number of values to display for an array variable.

Category: Manipulating Variables

Syntax

set -n max_depth -v *number_of_values*

Required Arguments

-n max_depth
specifies to update the maximum number of array values to display setting (max_depth).

-v *number_of_values*
specifies the maximum number of values to display for an array variable.

Example

Display a maximum of five values for an array variable:

```
set -n max_depth -v 5
```

See Also

"Specify the Number of Array Values to Display" on page 20

SETV Command

Changes the value of a variable.

Category: Manipulating Variables

Syntax

setv -v *variable_name* -n *new_value*

Required Arguments

-v *variable_name*
specifies the name of the variable whose value is to be changed.

-n *new_value*
provides the new value to assign to the specified variable.

If the variable has a character data type, the length of the new value must be less than or equal to the length of the current value. Otherwise, the new value is truncated.

If the new value contains a space, the new value must be enclosed in single quotation marks.

Details

The SETV commands assigns a new value to the specified variable.

When you detect an error during program execution, use the SETV command to assign new values to your variables so that you can continue the debugger session without having to recompile the program.

Example

- Change the value of variable ABC to **`123`**:

  ```
  setv -v abc -n 123
  ```

- Change the value of variable FRUIT to **`Honeydew Melon`**:

  ```
  setv -v fruit -n 'Honeydew Melon'
  ```

See Also

"Change the Value of a Variable" on page 20

STEP Command

Executes the current line of code.

Category: Controlling Program Execution

Syntax

step <-t>

Without Arguments

If you do not specify an argument on the STEP command, the debugger executes the current line of code. If this line contains a function call or a method call, the debugger executes the entire function or method, and then suspends program execution.

Optional Argument

-t

executes the current line of code. If this line contains a function call or a method call, the debugger steps into the function or method and suspends execution at the first executable line of code in that function or method.

Details

The STEP command executes one statement at a time. Use the GO command to execute multiple statements.

See Also

- "Step over a Function or Method" on page 7
- "Step into a Function or Method" on page 7
- "GO Command" on page 35

VARS Command

Retrieves all the variables and their values for a given scope.

Category: Manipulating Variables

Syntax

vars < | > <-c> <-e *stack_level*>

Without Arguments

The VARS command returns a list of all the local and global variables for the current scope.

Optional Arguments

-l
retrieves all the local variables that are in scope.

-g
retrieves all the global variables.

-c
retrieves only the variables whose values have changed.

-e *stack_level*
specifies the stack level for which to display variables.

Details

The VARS command displays information about all the variables that are applicable at the point of execution, including automatic variables like _N_ and _ERROR_.

Example

- Retrieve information about the global variables:

```
vars -g
```

- Retrieve information about the local variables whose values have changed:

```
vars -l -c
```

- Retrieve information about the local variables that were in scope for stack level two:

```
vars -l -e 2
```

WHATIS Command

Displays the **Type** attribute and the **Editable** attribute for a variable.

Category: Manipulating Variables

Syntax

whatis -v *variable_name*

Required Argument

-v *variable_name*
specifies the name of the variable whose information is to be retrieved.

Details

The WHATIS command provides the data type for the specified variable. This command also indicates whether the variable is editable.

Example

- Display the information for variable ABC:

```
whatis -v abc
```

WHERE Command

Lists the stack levels that are in the current call stack.

Category: Manipulating Debugging Requests

Syntax

where <-d *number_of_levels*>

Without Arguments

If you do not specify an argument, the WHERE command returns a list of all the stack levels that are in the current call stack.

Optional Argument

-d *number_of_levels*
specifies the number of stack levels to return for the current call stack, starting from the top of the call stack.

Example

- List the last five stack levels that were added to the call stack:

```
where -d 5
```

- List all the stack levels that are in the current call stack:

```
where
```

PART 2

DATA Step Debugger

5

Using the DATA Step Debugger

Introduction

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 do any of the following tasks:

- 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.

Although 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.

What Is the DATA Step Debugger ?

This documentation is for the DATA step debugger that is included with the SAS windowing environment. If you are using the DATA step debugger that is included with SAS Studio 5.2 or later, see SAS Studio 5.2: User's Guide.

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.

Note: The DATA step debugger is not supported for the DATA step that runs on the CAS server.

The DATA step debugger enables you to perform these 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 that 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. For example, if you begin execution with the GO command, 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 that 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.

Restriction: The DATA step debugger debugs only a single DATA step. If the code you are debugging contains code other than a single DATA step, the debugger stops with errors. For more information, see “Troubleshooting the Debugger” on page 61.

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 that 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 that 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

For a list of commands and their descriptions, see "DATA Step Debugger Commands by Category" on page 63.

Enter DATA step debugger commands on the debugger command line. 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 "SAS Operators in Expressions" in *SAS Language Reference: Concepts* 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 that 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;'`

Using the Macro Facility with the Debugger

Using Macros as Debugging Tools

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.

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 enables 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:

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 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 When Output Is Missing

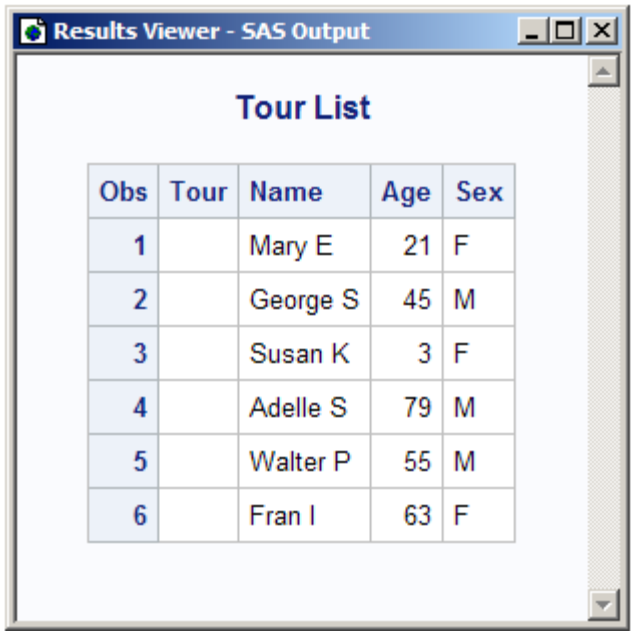
Discovering a Problem

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 gender 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;
```



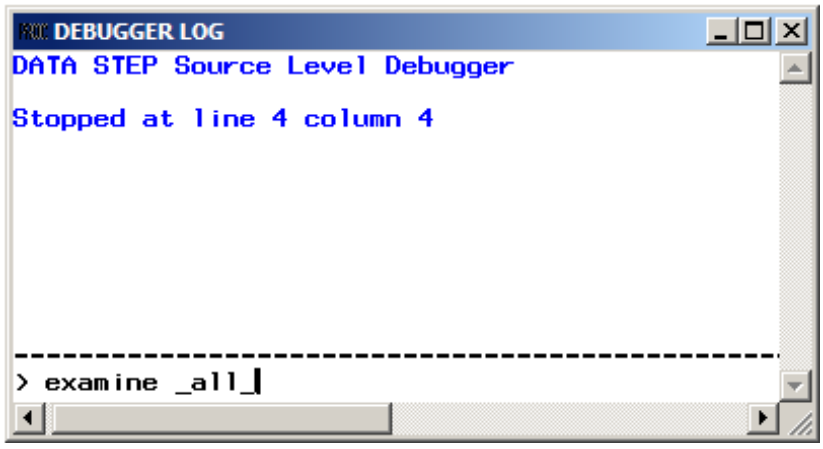
Obs	Tour	Name	Age	Sex
1		Mary E	21	F
2		George S	45	M
3		Susan K	3	F
4		Adelle S	79	M
5		Walter P	55	M
6		Fran I	63	F

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.

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.For a list of all commands, see "DATA Step Debugger Commands by Category".

When you press ENTER, the following display appears:

```
DEBUGGER LOG
DATA STEP Source Level Debugger

Stopped at line 4 column 4
> examine _all_
type =
Tour =
Name =
Age = .
Sex =
_ERROR_ = 0
_N_ = 1
-----------------------------------------------
>
```

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:

```
DEBUGGER LOG
Sex =
_ERROR_ = 0
_N_ = 1
>
Stepped to line 5 column 4
>
Stepped to line 6 column 7
>
Stepped to line 7 column 7
> examine tour
Tour = Tour 101
-----------------------------------------------
>
```

```
DEBUGGER SOURCE
  3 data tours (drop=type) /debug;
  4     input @1 type $ @;
  5     if type='H' then do;
  6        input @3 Tour $20.;
  7        return;
  8        end;
  9     else if type='P' then do;
  10        input @3 Name $10. Age 2. +1 Sex $1.;
  11        output;
  12        end;
  13     datalines;
```

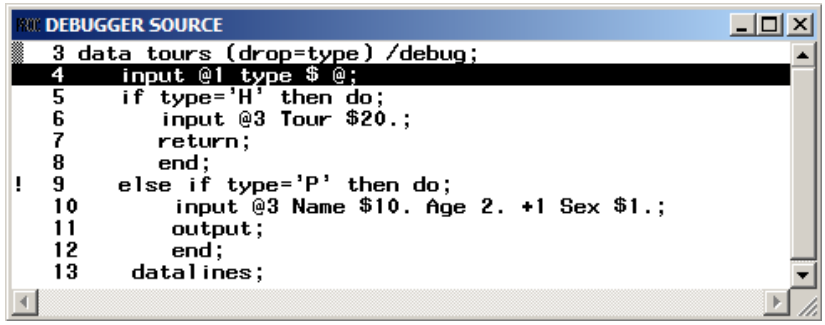
The variable Tour contains the value Tour 101, showing you that Tour was read. The first iteration of the DATA step worked as intended. Press ENTER to reach the top of the DATA step.

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 that you designate. In this example, suspend execution before executing the ELSE statement by setting a breakpoint at line 9.

```
break 9
```

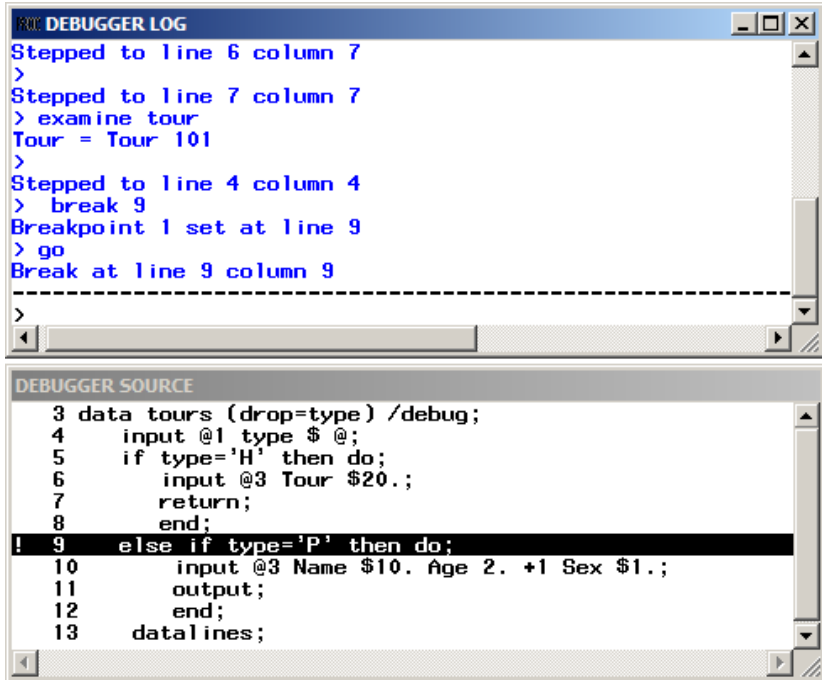
When you press ENTER, an exclamation point appears at line 9 in the DEBUGGER SOURCE window to mark the breakpoint:



Execute the GO command to continue DATA step execution until it reaches the breakpoint (in this case, line 9):

```
go
```

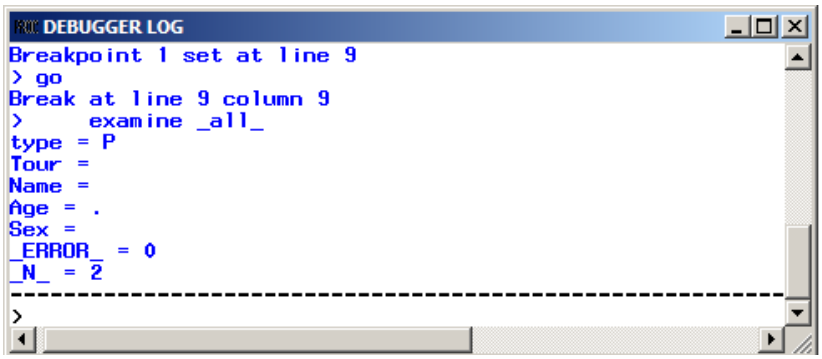
The following display shows the result:



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

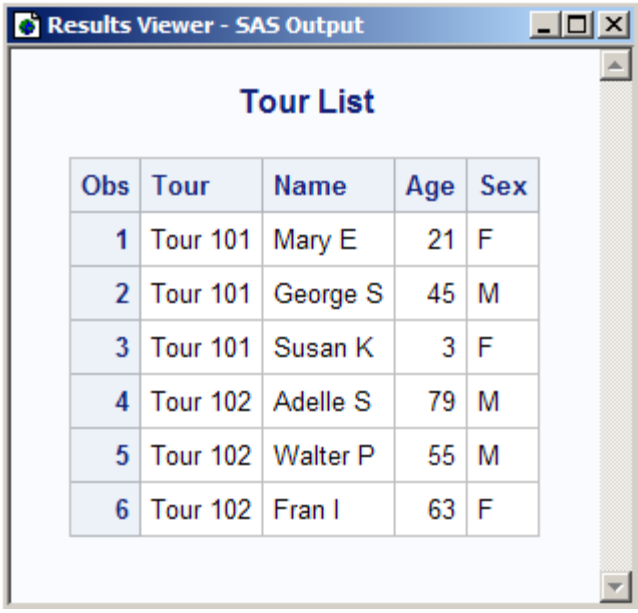
Correct the original program by adding the RETAIN statement. Delete the DEBUG option from the DATA step, and resubmit the program:

```
   /* 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:



Results Viewer - SAS Output

Tour List

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.

```
data tours;
  length Country $ 10;
  input Country $10. Start : mmddyy. End : mmddyy.;
  Duration=end-start;
datalines;
Italy       033012 041312
Brazil      021912 022812
Japan       052212 061512
Venezuela   110312 11801
Australia   122112 011513
;

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

Results Viewer - SAS Output

Tour Duration

Obs	Country	Start	End	Duration
1	Italy	30MAR2012	13APR2012	14
2	Brazil	19FEB2012	28FEB2012	9
3	Japan	22MAY2012	15JUN2012	24
4	Venezuela	03NOV2012	18JAN2012	-290
5	Australia	21DEC2012	15JAN2013	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:

DEBUGGER LOG

```
DATA STEP Source Level Debugger

Stopped at line 3 column 3
```

DEBUGGER SOURCE

```
1 data tours /debug;
2   length Country $ 10;
3   input Country $10. Start : mmddyy. End :
4   Duration=end-start;
5 datalines;
```

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_
```

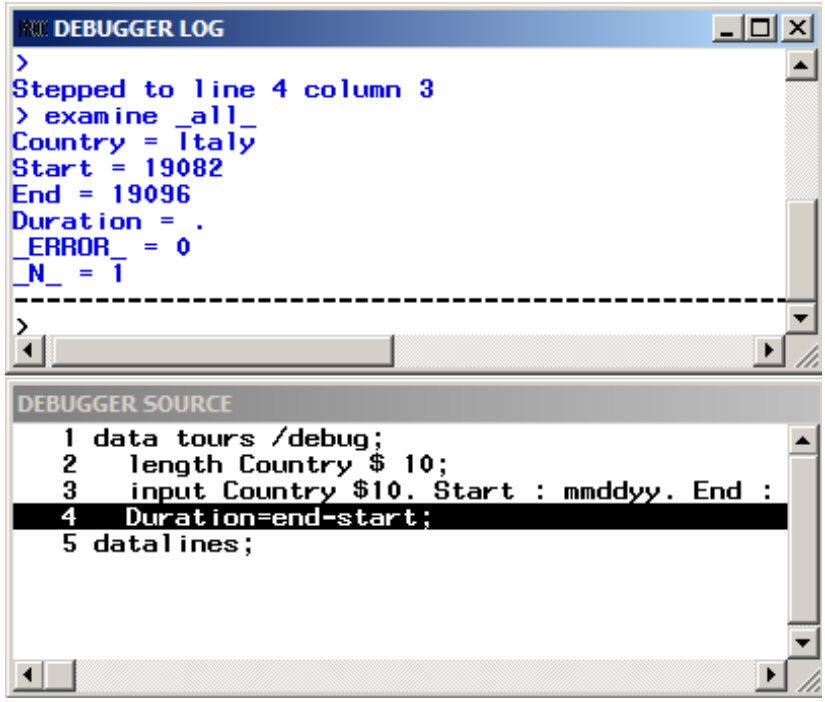
Initial values of all variables appear in the DEBUGGER LOG window. SAS has not yet executed the INPUT statement.

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:

```
DEBUGGER LOG
> go
Break at line 4 column 3
> examine _all_
Country = Venezuela
Start = 19300
End = 19010
Duration = .
_ERROR_ = 0
_N_ = 4
---------------------------------------------
>

DEBUGGER SOURCE
  1 data tours /debug;
  2   length Country $ 10;
  3   input Country $10. Start : mmddyy. End :
! 4   Duration=end-start;
  5 datalines;
```

To view formatted SAS dates, issue the EXAMINE command using the DATEw. format:

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

The following display shows the results:

```
DEBUGGER LOG
Country = Venezuela
Start = 19300
End = 19010
Duration = .
_ERROR_ = 0
_N_ = 4
> examine start date7. end date7.
Start = 03NOV12
End = 18JAN12
---------------------------------------------
>

DEBUGGER SOURCE
  1 data tours /debug;
  2   length Country $ 10;
  3   input Country $10. Start : mmddyy. End :
! 4   Duration=end-start;
  5 datalines;
```

Because the tour ends on November 18, 2012, and not on January 18, 2012, 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 whether you get the anticipated result. Issue the SET command using the DDMMMYY*w*. format:

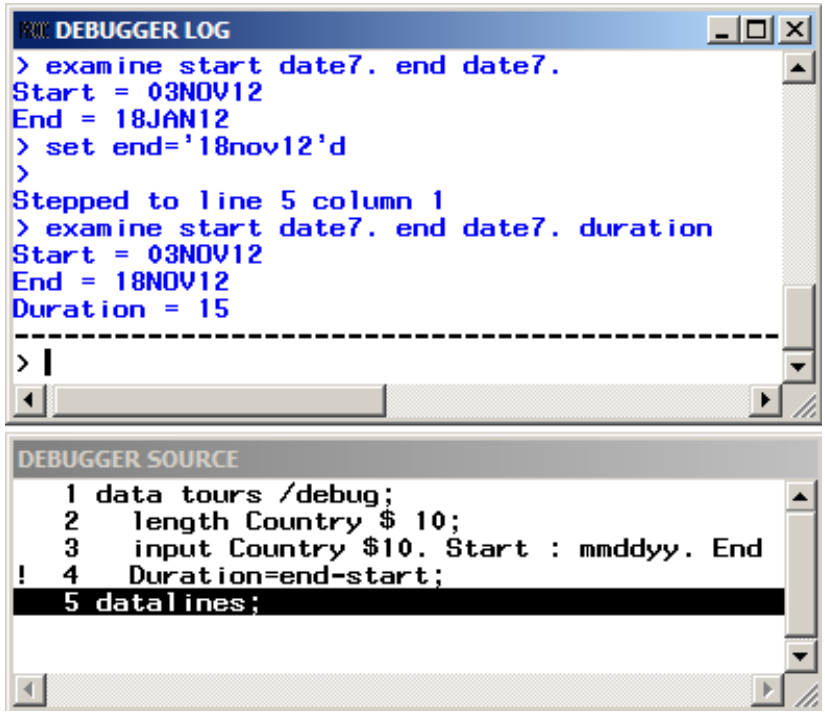
```
set end='18nov12'd
```

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:



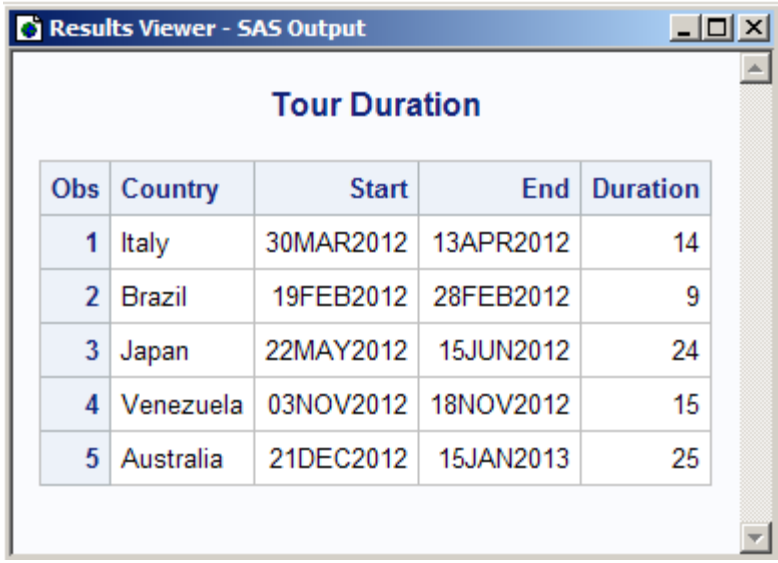
The Start, End, and Duration fields contain correct data.

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 */

data tours;
  length Country $ 10;
  input Country $10. Start : mmddyy. End : mmddyy.;
  duration=end-start;
datalines;
Italy       033012 041312
Brazil      021912 022812
Japan       052212 061512
Venezuela   110312 111812
Australia   122112 011513
;

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



Results Viewer - SAS Output

Tour Duration

Obs	Country	Start	End	Duration
1	Italy	30MAR2012	13APR2012	14
2	Brazil	19FEB2012	28FEB2012	9
3	Japan	22MAY2012	15JUN2012	24
4	Venezuela	03NOV2012	18NOV2012	15
5	Australia	21DEC2012	15JAN2013	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 five 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 that the variable BEGIN contains a SAS date value. To display the day of the week and date, use the WEEKDATE*w*. format with EXAMINE:

```
examine begin weekdate17.
```

When the value of BEGIN is 033012, the debugger displays the following:

```
Sun, Mar 30, 2012
```

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.
```

For example, when the value of STOCKNUM is 7, the debugger displays the following:

```
STOCKNUM = medium
```

Troubleshooting the Debugger

Issues and Resolutions

The DEBUGGER SOURCE window is empty, and when you attempt to set a breakpoint you receive the message "Line number `xx` out of range for compiled source".

The DATA step debugger is unable to retrieve source lines in some scenarios. For example, this occurs when a macro variable is referenced in the DATA statement and a DATA step or procedure has already been run in the current SAS session. To avoid the issue, remove the macro variable reference from the DATA statement. Or, restart SAS and run the debugger on the DATA step prior to running any other DATA steps or procedures.

6

Dictionary of DATA Step Debugger Commands

DATA Step Debugger Commands by Category

Category	Language Elements	Description
Controlling Program Execution	GO Command (p. 71)	Starts or resumes execution of the DATA step.
	JUMP Command (p. 73)	Restarts execution of a suspended program.
	STEP Command (p. 76)	Executes statements one at a time in the active program.
Controlling the Windows	HELP Command (p. 72)	Displays information about debugger commands.
	SWAP Command (p. 77)	Switches control between the SOURCE window and the LOG window.
Customizing the Debugger	ENTER Command (p. 69)	Assigns one or more debugger commands to the ENTER key.

Category	Language Elements	Description
Manipulating DATA Step Variables	CALCULATE Command (p. 67)	Evaluates a debugger expression and displays the result.
	DESCRIBE Command (p. 69)	Displays the attributes of one or more variables.
	EXAMINE Command (p. 70)	Displays the value of one or more variables.
	SET Command (p. 75)	Assigns a new value to a specified variable.
Manipulating Debugging Requests	BREAK Command (p. 64)	Suspends program execution at an executable statement.
	DELETE Command (p. 67)	Deletes breakpoints or the watch status of variables in the DATA step.
	LIST Command (p. 74)	Displays all occurrences of the item that is listed in the argument.
	TRACE Command (p. 78)	Controls whether the debugger displays a continuous record of the DATA step execution.
	WATCH Command (p. 79)	Suspends execution when the value of a specified variable changes.
Terminating the Debugger	QUIT Command (p. 75)	Terminates a debugger session.

Dictionary

BREAK Command

Suspends program execution at an executable statement.

Category: Manipulating Debugging Requests

Alias: B

Syntax

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

Required Argument

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.

Optional Arguments

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 the following:

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 does the following:

- 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.

Example

- 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 Command”
- “WATCH Command”

CALCULATE Command

Evaluates a debugger expression and displays the result.

Category: Manipulating DATA Step Variables

Syntax

CALC *expression*

Required Argument

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.

Example

- 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 48

DELETE Command

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_

Required 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(s)
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.

Example

- 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 Command" on page 64
- "WATCH Command" on page 79

DESCRIBE Command

Displays the attributes of one or more variables.

Category: Manipulating DATA Step Variables

Alias: DESC

Syntax

DESCRIBE *variable(s)* | _ALL_

Required Arguments

variable(s)
identifies one or more DATA step variables

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.

Example

- Display the attributes of variable ADDRESS:

```
desc address
```

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

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

ENTER Command

Assigns one or more debugger commands to the ENTER key.

Category: Customizing the Debugger

Syntax

ENTER *command-1* <; *command-2*; …>

Required Argument

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.

Example

- 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 Command

Displays the value of one or more variables.

Category:	Manipulating DATA Step Variables
Alias:	E

Syntax

EXAMINE *variable-1* <*format-1*> <*variable-2* <*format-2* …> >

EXAMINE _ALL_ <*format*>

Required Arguments

variable
identifies a DATA step variable.

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

Optional Argument

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

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.

Example

- 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

Commands:

- "DESCRIBE Command" on page 69

GO Command

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.

Optional 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.

Example

- 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 Command” on page 73
- “STEP Command” on page 76

HELP Command

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 Command

Restarts execution of a suspended program.

Category: Controlling Program Execution

Alias: J

Syntax

JUMP *line-number* | *label*

Required 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 that they had at that point. Then re-execute those statements with STEP or GO.

Example

- Jump to line 5:

```
j 5
```

See Also

Commands:

- "GO Command" on page 71
- "STEP Command" on page 76

LIST Command

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>

Required 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

Example

- 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 Command" on page 64
- "DELETE Command" on page 67
- "WATCH Command" on page 79

QUIT Command

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 Command

Assigns a new value to a specified variable.

Category:	Manipulating DATA Step Variables
Alias:	None

Syntax

SET *variable*=*expression*

Required 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.

Example

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

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

STEP Command

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.

Optional Argument

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 Command" on page 71
- "JUMP Command" on page 73

SWAP Command

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 Command

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

Use the TRACE command without arguments to determine whether tracing is on or off.

Optional 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.

Comparisons

TRACE displays the current status of the TRACE command.

Example

- Determine whether TRACE is ON or OFF:

```
trace
```

- Prepare to display a record of debugger execution:

```
trace on
```

WATCH Command

Suspends execution when the value of a specified variable changes.

Category: Manipulating Debugging Requests

Alias: W

Syntax

WATCH *variable(s)*

Required Argument

variable(s)
specifies one or more DATA step variables.

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 does the following:

- 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.

Example

- Monitor the variable DIVISOR for value changes:

```
w divisor
```