## Contents

**Chapter 1 • Getting Started with SAS Code Debugger** ........................................ 1
  - Introduction to SAS Code Debugger ........................................ 1
  - Connect Your SAS Session to SAS Code Debugger ....................... 2
  - Customizing the User Interface ........................................... 3
  - Controlling Program Execution ............................................ 4
  - Debugging a Sample Program ............................................... 6

**Chapter 2 • Managing Breakpoints** ............................................................. 11
  - About Breakpoints .................................................................... 11
  - Add Breakpoints ...................................................................... 11
  - Disable Breakpoints .............................................................. 12
  - Enable Breakpoints .............................................................. 13
  - Remove Breakpoints ............................................................. 13
  - Export Breakpoints .............................................................. 14
  - Import Breakpoints .............................................................. 14

**Chapter 3 • Working with Variables** ........................................................... 15
  - About Variables ...................................................................... 15
  - Change the Value of a Variable ............................................. 15
  - Specify the Number of Array Values to Display ....................... 16
  - Inspect an Array Variable ..................................................... 17
  - Add Watch Variables ........................................................... 17
  - Remove Watch Variables ....................................................... 18
  - Export Watch Variables ......................................................... 18
  - Import Watch Variables ......................................................... 18

**Chapter 4 • Issuing Commands (Preproduction)** .......................................... 21
  - Debugger Commands Overview ............................................. 21
  - Supported Debugger Commands by Category ............................ 22
  - Dictionary .............................................................................. 23
Chapter 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 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 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 11.

• **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 Chapter 4, “Issuing Commands (Preproduction),” on page 21.

• **Variables**: Enables you to change the values of variables, to create watch variables, and to examine your variables as you step through the code. For more information, see Chapter 3, “Working with Variables,” on page 15.

• **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 \textbullet\ in the toolbar for that panel and select \textbf{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 \textbullet\ in the toolbar for the maximized panel and select \textbf{Restore down}.

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

\textbf{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.

\textbf{Remove Panels}

Complete one of the following steps:

- Click \textbullet\ in the toolbar for the panel that you want to remove, and select \textbf{Remove}. If a confirmation message appears, click \textbf{Remove} to confirm that you want to remove the panel.

- Click the \textbf{Panels} menu in the debugger toolbar, and select a panel whose name is preceded by a \checkmark. If a confirmation message appears, click \textbf{Remove} to confirm that you want to remove the panel.

\textbf{Add Panels}

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

\textbf{Restore Default Panel Layout}

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

\textbf{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.

\textbf{Start or Resume Program Execution}

Complete one of the following steps:

- Click \textbullet\ 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” on page 30

### 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” on page 33

### Step into a Function or Method

Complete one of the following steps:
- Click \( \text{F}3 \) 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” on page 33

### Step out of a Function or Method

Complete one of the following steps:
- Click \( \text{F}7 \) 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.
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” on page 31

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” on page 32

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 program.

   When you sign in, the application creates a SAS session on your behalf.

2. Execute the sample PROC FCMP program. The following table lists the values for the mean and the standard deviation. The sample program calculates the mean correctly, but the standard deviation does not match the manually calculated value.
Table 1.1  Results for the Mean and the Standard Deviation

<table>
<thead>
<tr>
<th>Statistic</th>
<th>Sample Program Results</th>
<th>Manually Calculated Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mean (myM)</td>
<td>1269.05</td>
<td>1269.05</td>
</tr>
<tr>
<td>Standard deviation (myS)</td>
<td>385.2823759</td>
<td>422.0556966</td>
</tr>
</tbody>
</table>

3. Sign in to SAS Code Debugger so that you can debug the program. When you sign in, SAS Code Debugger creates a debugger session on your behalf.

4. In the Code panel in SAS Code Debugger, copy the OPTIONS statement to the clipboard.

5. In the sample PROC FCMP program, replace the following code with the OPTIONS statement that you copied:

   /*<insert_options_statement>*/

   The OPTIONS statement connects your SAS session to your debugger session.

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. Complete the following steps:
   a. Click ‹ in the Watch panel.
   b. Type `sd` in the Variable name field.
   c. 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 F10 in the debugger toolbar six times 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 1.2 Values of Variable “sd” for the DO Loop**

<table>
<thead>
<tr>
<th>Step</th>
<th>Sample Program Results</th>
<th>Manually Calculated Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>300139.6225</td>
<td>300139.6225</td>
</tr>
<tr>
<td>2</td>
<td>368025.925</td>
<td>368025.925</td>
</tr>
<tr>
<td>3</td>
<td>398319.3275</td>
<td>398319.3275</td>
</tr>
<tr>
<td>4</td>
<td>408979.89</td>
<td>408979.89</td>
</tr>
<tr>
<td>5</td>
<td>458019.9925</td>
<td>458019.9925</td>
</tr>
<tr>
<td>6</td>
<td>890655.055</td>
<td>890655.055</td>
</tr>
</tbody>
</table>

12. Click F10 in the debugger toolbar to debug the line of code that is at the next breakpoint.

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:

```
quitting
```

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. Replace the current OPTIONS statement with the following OPTIONS statement:
      ```
options CMPOPT = NONE;
```
      The new OPTIONS statement disconnects your SAS session from your debugger session.

15. Execute the updated sample program. Both the mean and the standard deviation have the correct values.
   - `myM = 1269.05`
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 6.

```sas
/*<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(s);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;
Chapter 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 \( \text{[•]} \), and select **Add breakpoint to selected functions and methods**.

Click \( \text{[•]} \), 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**” on page 23
- “**BREAKROUTINES**” on page 27

---

**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 \( \text{[•]} \) in the toolbar for the **Breakpoints** panel, and select **Disable selected breakpoints**.
  - Click \( \text{[•]} \) 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” on page 25

---

### 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” on page 26

---

### 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.
Export Breakpoints

1. Click ![image] 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 ![image] 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.
Chapter 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” on page 32

---

**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 Help button. 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 17
- “SET” on page 32

---

**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 \( \checkmark \). 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 (\(<\text{data\_type}> [\langle\text{total\_values}\rangle]\)).
   - 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 16
- “DISPLAY” on page 28

---

**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.
Chapter 4

Issuing Commands (Preproduction)

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.

---

### Supported Debugger Commands by Category

<table>
<thead>
<tr>
<th>Category</th>
<th>Language Elements</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Controlling Program Execution</strong></td>
<td>GO (p. 30)</td>
<td>Starts or resumes execution of a program.</td>
</tr>
<tr>
<td></td>
<td>PAUSE (p. 31)</td>
<td>Pauses the execution of a program.</td>
</tr>
<tr>
<td></td>
<td>STEP (p. 33)</td>
<td>Executes the current line of code.</td>
</tr>
<tr>
<td><strong>Manipulating Debugging Requests</strong></td>
<td>BREAK (p. 23)</td>
<td>Sets a breakpoint at an executable statement.</td>
</tr>
<tr>
<td></td>
<td>BREAKCLEAR (p. 25)</td>
<td>Removes an existing breakpoint.</td>
</tr>
<tr>
<td></td>
<td>BREAKDISABLE (p. 25)</td>
<td>Disables a breakpoint that is currently enabled.</td>
</tr>
<tr>
<td></td>
<td>BREAKENABLE (p. 26)</td>
<td>Enables a breakpoint that is currently disabled.</td>
</tr>
<tr>
<td></td>
<td>BREAKLIST (p. 27)</td>
<td>Lists all the breakpoints that exist in a debugger session.</td>
</tr>
<tr>
<td></td>
<td>BREAKROUTINES (p. 27)</td>
<td>Adds breakpoints to your functions and methods.</td>
</tr>
<tr>
<td></td>
<td>ENV (p. 29)</td>
<td>Displays the information for a stack level.</td>
</tr>
<tr>
<td></td>
<td>GETROUTINES (p. 30)</td>
<td>Returns a list of your functions and methods.</td>
</tr>
<tr>
<td></td>
<td>WHERE (p. 35)</td>
<td>Lists the stack levels that are in the current call stack.</td>
</tr>
<tr>
<td><strong>Manipulating Variables</strong></td>
<td>ARGS (p. 23)</td>
<td>Lists all the variables that are passed as parameters into the current function or method.</td>
</tr>
<tr>
<td></td>
<td>DISPLAY (p. 28)</td>
<td>Retrieves the value of a variable.</td>
</tr>
<tr>
<td></td>
<td>SET (p. 32)</td>
<td>Sets the maximum number of values to display for an array variable.</td>
</tr>
<tr>
<td></td>
<td>SETV (p. 32)</td>
<td>Changes the value of a variable.</td>
</tr>
</tbody>
</table>
### Dictionary

**ARGS**

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

**Category:** Manipulating Variables

**Syntax**

```
args
```

**Details**

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

---

**BREAK**

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
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 11
- “BREAKROUTINES” on page 27
- “BREAKCLEAR” on page 25
- “BREAKDISABLE” on page 25
- “BREAKENABLE” on page 26
BREAKCLEAR
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 13
• “BREAK” on page 23
• “BREAKLIST” on page 27

BREAKDISABLE
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 12
- “BREAK” on page 23
- “BREAKENABLE” on page 26
- “BREAKLIST” on page 27

---

**BREAKENABLE**

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 13
- “BREAK” on page 23
- “BREAKLIST” on page 27

BREAKLIST
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
Adds breakpoints to your functions and methods.

Category: Manipulating Debugging Requests

Syntax
breakroutines <-f | -m>
Without Arguments
If you do not specify an argument, the BREAKROUTINES command adds breakpoints to all the functions and methods that are included in your code.

Optional Arguments
-f
   adds breakpoints to all the functions 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:
  breakroutines -f

See Also
- “Add Breakpoints” on page 11
- “BREAK” on page 23

DISPLAY
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 17

ENV

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
-1 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:
GETROUTINES

Returns a list of your functions and methods.

Syntax

gentroutines <-f | -m>

Without Arguments
If you do not specify an argument, the GETROUTINES command returns a list of all 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.

Optional Arguments

-f
returns a list of all the functions that are included in your code along with any functions 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
  gentroutines -m

• Retrieve a list of all the functions:
  gentroutines -f

GO

Starts or resumes execution of a program.

Syntax

go <-1 statement_number | -o | -n 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 4
- “Step out of a Function or Method” on page 5

---

**PAUSE**

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 6
QUIT
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 6

SET
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 16

SETV
Changes the value of a variable.
**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 15

---

**STEP**

Executes the current line of code.

**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 5
- “Step into a Function or Method” on page 5
- “GO” on page 30

VARS

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

Category: Manipulating Variables

Syntax

vars <l | -g > <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**

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**

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
Gain Greater Insight into Your SAS® Software with SAS Books.

Discover all that you need on your journey to knowledge and empowerment.

support.sas.com/bookstore for additional books and resources.