

Base SAS[®] 9.3 Utilities: Reference



The correct bibliographic citation for this manual is as follows: SAS Institute Inc. 2011. *Base SAS® 9.3 Utilities: Reference*. Cary, NC: SAS Institute Inc.

Base SAS® 9.3 Utilities: Reference

Copyright © 2011, SAS Institute Inc., Cary, NC, USA

ISBN 978–1–60764–905–2 (electronic book)

ISBN 978–1–60764–905–2

All rights reserved. Produced in the United States of America.

For a hardcopy book: No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form or by any means, electronic, mechanical, photocopying, or otherwise, without the prior written permission of the publisher, SAS Institute Inc.

For a Web download or e-book: Your use of this publication shall be governed by the terms established by the vendor at the time you acquire this publication.

The scanning, uploading, and distribution of this book via the Internet or any other means without the permission of the publisher is illegal and punishable by law. Please purchase only authorized electronic editions and do not participate in or encourage electronic piracy of copyrighted materials. Your support of others' rights is appreciated.

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

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

ISBN 978–1–60764–905–2

1st printing, July 2011

ISBN 978–1–60764–905–2

1st printing, July 2011

SAS® Publishing provides a complete selection of books and electronic products to help customers use SAS software to its fullest potential. For more information about our e-books, e-learning products, CDs, and hard-copy books, visit the SAS Publishing Web site at

support.sas.com/publishing or call 1-800-727-3228.

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

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

Contents

<i>About This Book</i>	<i>v</i>
<i>What's New in Base SAS 9.3 Utilities</i>	<i>ix</i>
<i>Recommended Reading</i>	<i>xi</i>

PART 1 Macro Utilities 1

Chapter 1 • Dictionary of Utilities	3
Dictionary	3

PART 2 DATA Step Debugger 7

Chapter 2 • Using the DATA Step Debugger	9
Introduction	9
Basic Usage	10
Using the Macro Facility with the Debugger	12
Examples	13
Chapter 3 • Dictionary of DATA Step Debugger Commands	25
DATA Step Debugger Commands by Category	25
Dictionary	26
Index	41

About This Book

Syntax Conventions for the SAS Language

Overview of Syntax Conventions for the SAS Language

SAS uses standard conventions in the documentation of syntax for SAS language elements. These conventions enable you to easily identify the components of SAS syntax. The conventions can be divided into these parts:

- syntax components
- style conventions
- special characters
- references to SAS libraries and external files

Syntax Components

The components of the syntax for most language elements include a keyword and arguments. For some language elements, only a keyword is necessary. For other language elements, the keyword is followed by an equal sign (=).

keyword

specifies the name of the SAS language element that you use when you write your program. Keyword is a literal that is usually the first word in the syntax. In a CALL routine, the first two words are keywords.

In the following examples of SAS syntax, the keywords are the first words in the syntax:

CHAR (*string, position*)

CALL RANBIN (*seed, n, p, x*);

ALTER (*alter-password*)

BEST *w.*

REMOVE *<data-set-name>*

In the following example, the first two words of the CALL routine are the keywords:

CALL RANBIN(*seed, n, p, x*)

The syntax of some SAS statements consists of a single keyword without arguments:

DO;

... *SAS code* ...

END;

Some system options require that one of two keyword values be specified:

DUPLEX | NODUPLEX**argument**

specifies a numeric or character constant, variable, or expression. Arguments follow the keyword or an equal sign after the keyword. The arguments are used by SAS to process the language element. Arguments can be required or optional. In the syntax, optional arguments are enclosed between angle brackets.

In the following example, *string* and *position* follow the keyword CHAR. These arguments are required arguments for the CHAR function:

CHAR (*string*, *position*)

Each argument has a value. In the following example of SAS code, the argument *string* has a value of 'summer', and the argument *position* has a value of

```
4: x=char('summer', 4);
```

In the following example, *string* and *substring* are required arguments, while *modifiers* and *startpos* are optional.

FIND(*string*, *substring* <,*modifiers*> <,*startpos*>)

Note: In most cases, example code in SAS documentation is written in lowercase with a monospace font. You can use uppercase, lowercase, or mixed case in the code that you write.

Style Conventions

The style conventions that are used in documenting SAS syntax include uppercase bold, uppercase, and italic:

UPPERCASE BOLD

identifies SAS keywords such as the names of functions or statements. In the following example, the keyword ERROR is written in uppercase bold:

ERROR<*message*>;**UPPERCASE**

identifies arguments that are literals.

In the following example of the CMPMODEL= system option, the literals include BOTH, CATALOG, and XML:

CMPMODEL = BOTH | CATALOG | XML*italics*

identifies arguments or values that you supply. Items in italics represent user-supplied values that are either one of the following:

- nonliteral arguments In the following example of the LINK statement, the argument *label* is a user-supplied value and is therefore written in italics:

LINK *label*;

- nonliteral values that are assigned to an argument

In the following example of the FORMAT statement, the argument DEFAULT is assigned the variable *default-format*:

FORMAT = *variable-1* <, ..., *variable-nformat*> <DEFAULT = *default-format*>;

Items in italics can also be the generic name for a list of arguments from which you can choose (for example, *attribute-list*). If more than one of an item in italics can be used, the items are expressed as *item-1*, ..., *item-n*.

Special Characters

The syntax of SAS language elements can contain the following special characters:

=

an equal sign identifies a value for a literal in some language elements such as system options.

In the following example of the MAPS system option, the equal sign sets the value of MAPS:

MAPS = *location-of-maps*

< >

angle brackets identify optional arguments. Any argument that is not enclosed in angle brackets is required.

In the following example of the CAT function, at least one item is required:

CAT (*item-1* <, ..., *item-n*>)

|

a vertical bar indicates that you can choose one value from a group of values. Values that are separated by the vertical bar are mutually exclusive.

In the following example of the CMPMODEL= system option, you can choose only one of the arguments:

CMPMODEL = BOTH | CATALOG | XML

...

an ellipsis indicates that the argument or group of arguments following the ellipsis can be repeated. If the ellipsis and the following argument are enclosed in angle brackets, then the argument is optional.

In the following example of the CAT function, the ellipsis indicates that you can have multiple optional items:

CAT (*item-1* <, ..., *item-n*>)

'value' or "value"

indicates that an argument enclosed in single or double quotation marks must have a value that is also enclosed in single or double quotation marks.

In the following example of the FOOTNOTE statement, the argument *text* is enclosed in quotation marks:

FOOTNOTE <*n*> <*ods-format-options* 'text' | "text">;

;

a semicolon indicates the end of a statement or CALL routine.

In the following example each statement ends with a semicolon: **data** **namegame**;
length **color** **name** **\$8**; **color** = 'black'; **name** = 'jack'; **game** =
trim(**color**) || **name**; **run**;

References to SAS Libraries and External Files

Many SAS statements and other language elements refer to SAS libraries and external files. You can choose whether to make the reference through a logical name (a libref or fileref) or use the physical filename enclosed in quotation marks. If you use a logical name, you usually have a choice of using a SAS statement (LIBNAME or FILENAME) or the operating environment's control language to make the association. Several methods of referring to SAS libraries and external files are available, and some of these methods depend on your operating environment.

In the examples that use external files, SAS documentation uses the italicized phrase *file-specification*. In the examples that use SAS libraries, SAS documentation uses the italicized phrase *SAS-library*. Note that *SAS-library* is enclosed in quotation marks:

```
infile file-specification obs = 100;  
libname libref 'SAS-library';
```


What's New in Base SAS 9.3 Utilities

Changes to *SAS Language Reference: Dictionary*

Prior to SAS 9.3, this document was part of *SAS Language Reference: Dictionary*. Starting with SAS 9.3, *SAS Language Reference: Dictionary* has been divided into seven documents:

- *SAS Data Set Options: Reference*
- *SAS Formats and Informats: Reference*
- *SAS Functions and CALL Routines: Reference*
- *SAS Statements: Reference*
- *SAS System Options: Reference*
- *SAS Component Objects: Reference* (contains the documentation for the Hash Object and the Java Object)
- *Base SAS Utilities: Reference* (contains the documentation for the SAS DATA step debugger and the SAS Utility macro %DS2CSV)

Recommended Reading

Here is the recommended reading list for this title:

- *SAS Component Objects: Reference*
- *SAS Formats and Informats: Reference*
- *SAS Functions and CALL Routines: Reference*
- *SAS Language Reference: Concepts*
- *SAS Macro Language: Reference*
- *SAS Statements: Reference*
- *SAS System Options: Reference*

For a complete list of SAS publications, go to support.sas.com/bookstore. If you have questions about which titles you need, please contact a SAS Publishing Sales Representative:

SAS Publishing Sales
SAS Campus Drive
Cary, NC 27513-2414
Phone: 1-800-727-3228
Fax: 1-919-677-8166
E-mail: sasbook@sas.com
Web address: support.sas.com/bookstore

Part 1

Macro Utilities

Chapter 1

Dictionary of Utilities 3

Chapter 1

Dictionary of Utilities

Dictionary	3
%DS2CSV Macro	3

Dictionary

%DS2CSV Macro

Converts SAS data sets to comma-separated value (CSV) files.

Restriction: This macro cannot be used in a DATA step. Run the macro only in open code.

Syntax

`%DS2CSV(argument-1=value-1, argument-2=value-2 <...argument-n=value-n>)`

Arguments That Affect Input and Output

csvfile=external-filename

specifies the name of the CSV file where the formatted output is to be written. If the file that you specify does not exist, then it is created for you.

Note: Do not use the CSVFILE argument if you use the CSVFREF argument.

csvfref=fileref

specifies the SAS fileref that points to the location of the CSV file where the formatted output is to be written. If the file that you specify does not exist, then it is created for you.

Note: Do not use the CSVFREF argument if you use the CSVFILE argument.

openmode=REPLACE|APPEND

indicates whether the new CSV output overwrites the information that is currently in the specified file or if the new output is appended to the end of the existing file. The default value is REPLACE. If you do not want to replace the current contents, then specify OPENMODE=APPEND to add your new CSV-formatted output to the end of an existing file.

Note: OPENMODE=APPEND is not valid if you are writing your resulting output to a partitioned data set (PDS) on z/OS.

Arguments That Affect MIME and HTTP Headers

For more information about MIME and HTTP headers, refer to the Internet Request for Comments (RFC) documents RFC 1521 (<http://asg.web.cmu.edu/rfc/rfc1521.html>) and RFC 1945 (<http://asg.web.cmu.edu/rfc/rfc1945.html>), respectively.

conttype=Y | N

indicates whether to write a content type header. This header is written by default.

Restriction: This argument is valid only when RUNMODE=S.

contdisp=Y | N

indicates whether to write a content disposition header. This header is written by default.

Restriction: This argument is valid only when RUNMODE=S.

Note: If you specify CONTDISP=N, then the SAVEFILE argument is ignored.

mimehdr1=MIME/HTTP-header

specifies the text that is to be used for the first MIME or HTTP header that is written. This header is written after the content type and disposition headers. By default, nothing is written for this header.

Restriction: This argument is valid only when RUNMODE=S.

mimehdr2=MIME/HTTP-header

specifies the text that is to be used for the second MIME or HTTP header that is written. This header is written after the content type and disposition headers. By default, nothing is written for this header.

Restriction: This argument is valid only when RUNMODE=S.

mimehdr3=MIME/HTTP-header

specifies the text that is to be used for the third MIME or HTTP header that is written when RUNMODE=S is specified. This header is written after the content type and disposition headers. By default, nothing is written for this header.

Restriction: This argument is valid only when RUNMODE=S.

mimehdr4=MIME/HTTP-header

specifies the text that is to be used for the fourth MIME or HTTP header that is written. This header is written after the content type and disposition headers. By default, nothing is written for this header.

Restriction: This argument is valid only when RUNMODE=S.

mimehdr5=MIME/HTTP-header

specifies the text that is to be used for the fifth MIME or HTTP header that is written. This header is written after the content type and disposition headers. By default, nothing is written for this header.

runmode=S | B

specifies whether you are running the %DS2CSV macro in batch or server mode. The default setting for this argument is RUNMODE=S.

- *Server mode* (RUNMODE=S) is used with Application Dispatcher programs and streaming output stored processes. Server mode causes DS2CSV to generate appropriate MIME or HTTP headers. For more information about Application Dispatcher, refer to the Application Dispatcher documentation at <http://support.sas.com/rnd/web/internet/dispatch.html>.
- *Batch mode* (RUNMODE=B) means that you are submitting the DS2CSV macro in the SAS Program Editor or that you included it in a SAS program.

Note: No HTTP headers are written when you specify batch mode.

Restriction: RUNMODE=S is valid only when used within the SAS/IntrNet and Stored Process servers.

savefile=filename

specifies the filename to display in the Web browser's **Save As** dialog box. The default value is the name of the data set plus “.csv”.

Restriction: This argument is valid only when RUNMODE=S.

Note: This argument is ignored if CONTDISP=N is specified.

Arguments That Affect CSV Creation

colhead=Y | N

indicates whether to include column headings in the CSV file. The column headings that are used depend on the setting of the LABELS argument. By default, column headings are included as the first record of the CSV file.

data=SAS-data-set-name

specifies the SAS data set that contains the data that you want to convert into a CSV file. This argument is required. However, if you omit the data set name, DS2CSV attempts to use the most recently created SAS data set.

formats=Y | N

indicates whether to apply the data set's defined variable formats to the values in the CSV file. By default, all formats are applied to values before they are added to the CSV file. The formats must be stored in the data set in order for them to be applied.

labels=Y | N

indicates whether to use the SAS variable labels that are defined in the data set as your column headings. The DS2CSV macro uses the variable labels by default. If a variable does not have a SAS label, then use the name of the variable. Specify labels=N to use variable names instead of the SAS labels as your column headings.

See: The [colhead](#) on page 5 argument for more information about column headings.

pw=password

specifies the password that is needed to access a password-protected data set. This argument is required if the data set has a READ or PW password. (You do not need to specify this argument if the data set has only WRITE or ALTER passwords.)

sepchar=separator-character

specifies the character that is used for the separator character. Specify the two-character hexadecimal code for the character or omit this argument to get the default setting. The default settings are 2C for ASCII systems and 6B for EBCDIC systems. (These settings represent commas (,) on their respective systems.)

var=var1 var2 ...

specifies the variables that are to be included in the CSV file and the order in which they should be included. To include all of the variables in the data set, do not specify this argument. If you want to include only a subset of the variables, then list each variable name and use single blank spaces to separate the variables. Do not use a comma in the list of variable names.

Restriction: A range of values is not valid. For example, **var1-var4**.

where=where-expression

specifies a valid WHERE clause that selects observations from the SAS data set. Using this argument subsets your data based on the criteria that you supply for *where-expression*.

Details

The DS2CSV macro converts SAS data sets to comma-separated value (CSV) files. You can specify the hexadecimal code for the separator character if you want to create some other type of output file (for example, a tab-separated value file).

Example

The following example uses the %DS2CSV macro to convert the SASHELP.RETAIL data set to a comma-separated value file:

```
%ds2csv (data=sashelp.retail, runmode=b, csvfile=c:\temp\retail.csv);
```

Part 2

DATA Step Debugger

<i>Chapter 2</i>	
<i>Using the DATA Step Debugger</i>	<i>9</i>
<i>Chapter 3</i>	
<i>Dictionary of DATA Step Debugger Commands</i>	<i>25</i>

Chapter 2

Using the DATA Step Debugger

Introduction	9
What Is Debugging?	9
What Is the DATA Step Debugger ?	10
Basic Usage	10
How a Debugger Session Works	10
Using the Windows	11
Entering Commands	11
Working with Expressions	11
Assigning Commands to Function Keys	11
Using the Macro Facility with the Debugger	12
Using Macros as Debugging Tools	12
Creating Customized Debugging Commands with Macros	12
Debugging a DATA Step Generated by a Macro	12
Examples	13
Example 1: Debugging a Simple DATA Step When Output Is Missing	13
Example 2: Working with Formats	18
Example 3: Debugging DO Loops	23
Example 4: Examining Formatted Values of Variables	24

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.

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

What Is the DATA Step Debugger ?

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

The DATA step debugger enables you to perform 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.

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 command and their descriptions, see [“DATA Step Debugger Commands by Category” on page 25](#).

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 Chapter 6 of *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;

```



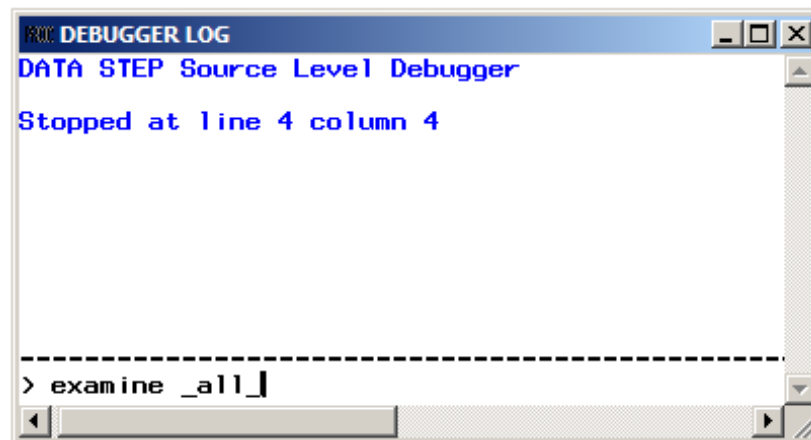
Obs	Tour	Name	Age	Sex
1		Mary E	21	F
2		George S	45	M
3		Susan K	3	F
4		Adelle S	79	M
5		Walter P	55	M
6		Fran I	63	F

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” on page 25](#).

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;

```

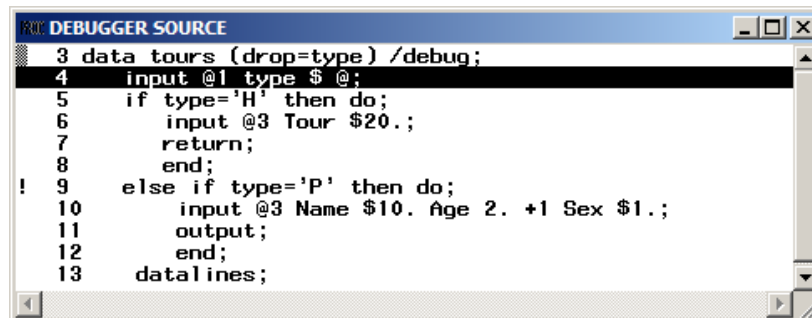
The variable Tour contains the value Tour 101, showing you that Tour was read. The first iteration of the DATA step worked as intended. Press ENTER to reach the top of the DATA step.

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:



```

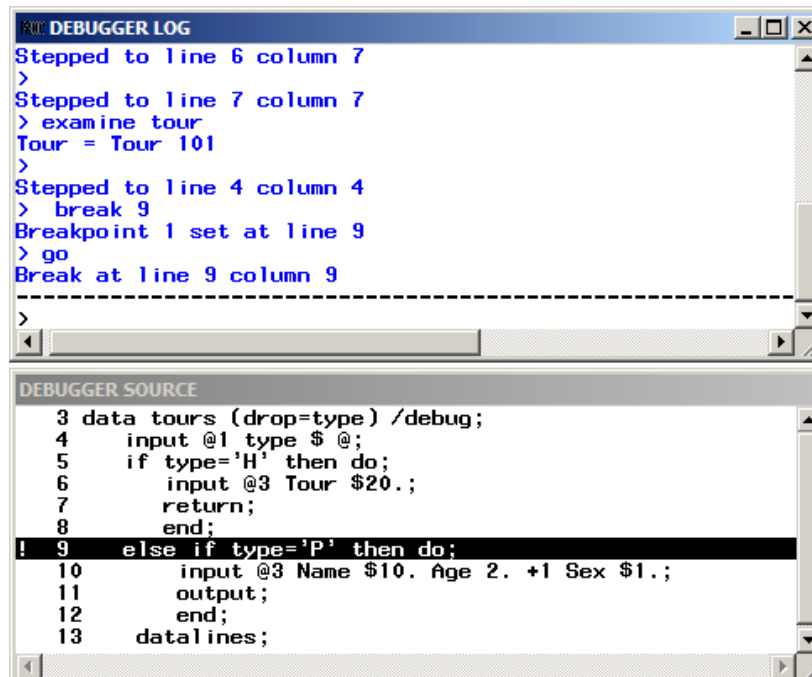
DEBUGGER SOURCE
3 data tours (drop=type) /debug;
4 input @1 type $ @;
5 if type='H' then do;
6   input @3 Tour $20.;
7   return;
8   end;
! 9 else if type='P' then do;
10   input @3 Name $10. Age 2. +1 Sex $1.;
11   output;
12   end;
13 datalines;

```

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:



```

DEBUGGER LOG
Stepped to line 6 column 7
>
Stepped to line 7 column 7
> examine tour
Tour = Tour 101
>
Stepped to line 4 column 4
> break 9
Breakpoint 1 set at line 9
> go
Break at line 9 column 9
-----
>

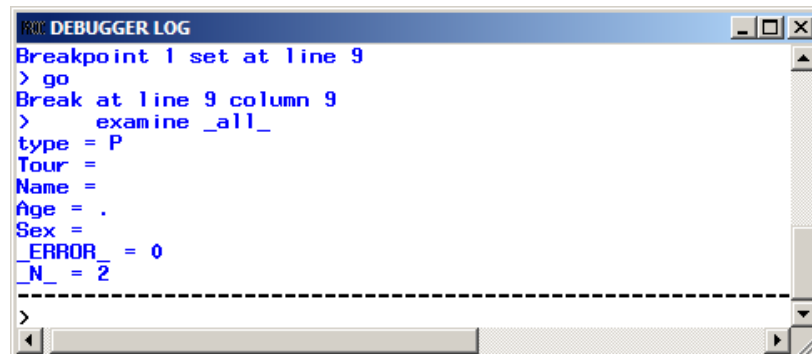
DEBUGGER SOURCE
3 data tours (drop=type) /debug;
4 input @1 type $ @;
5 if type='H' then do;
6   input @3 Tour $20.;
7   return;
8   end;
! 9 else if type='P' then do;
10   input @3 Name $10. Age 2. +1 Sex $1.;
11   output;
12   end;
13 datalines;

```

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:



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.

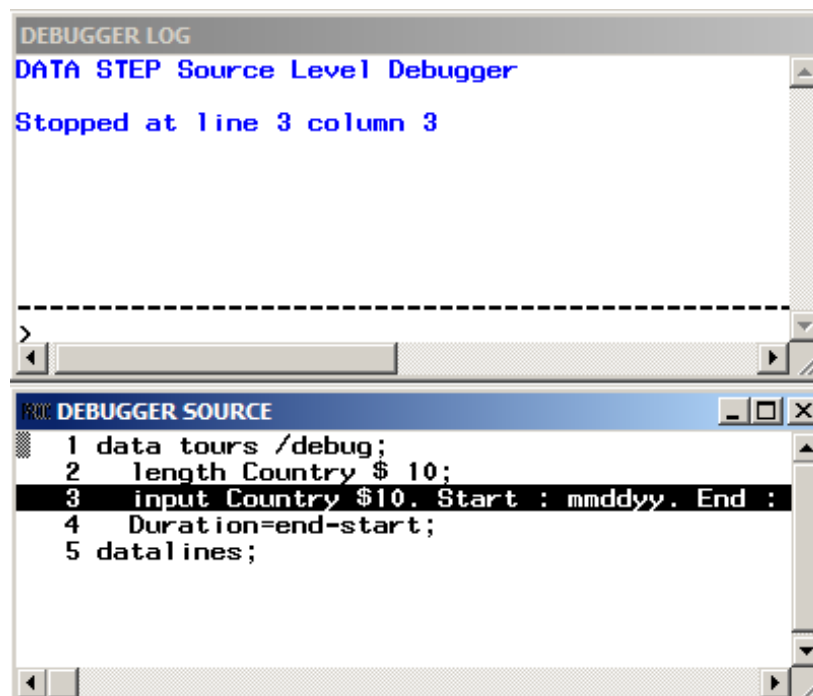
```
data tours;
  length Country $ 10;
  input Country $10. Start : mmddyy. End : mmddyy.;
  Duration=end-start;
datalines;
Italy      033012 041312
Brazil     021912 022812
Japan      052212 061512
Venezuela  110312 11801
Australia  122112 011513
;

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



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:



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_
```

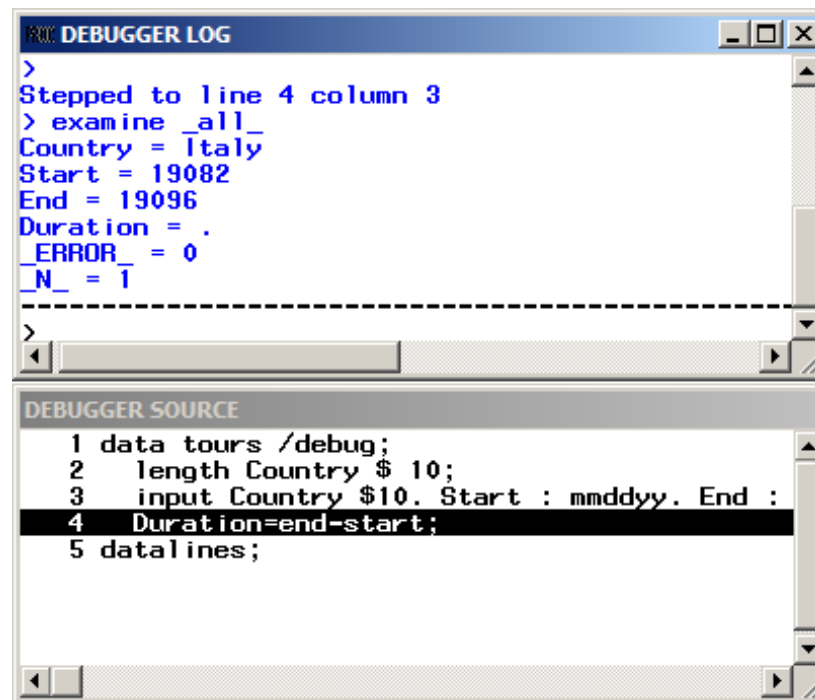
Initial values of all variables appear in the DEBUGGER LOG window. SAS has not yet executed the INPUT statement.

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:

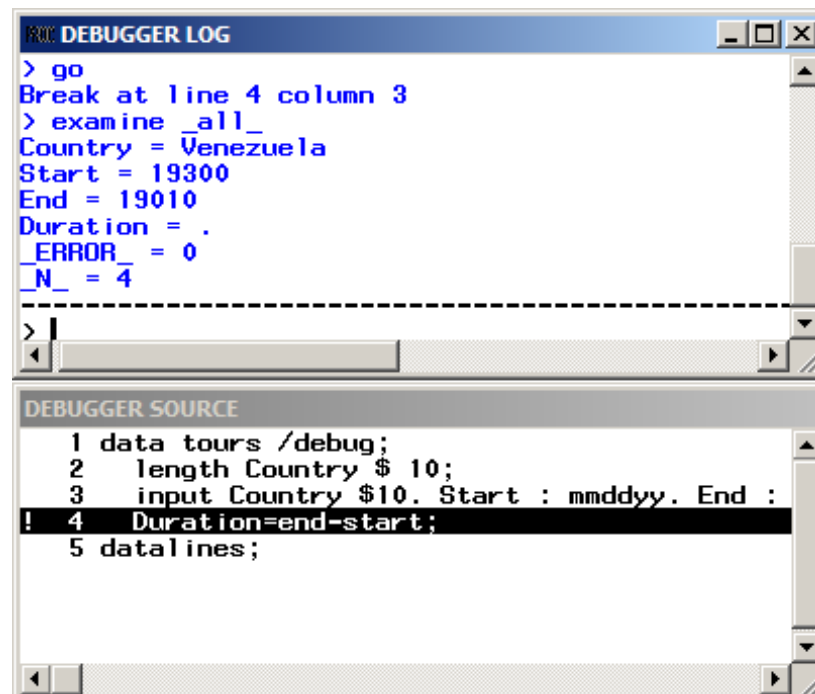
```
go
```

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
-----
> |

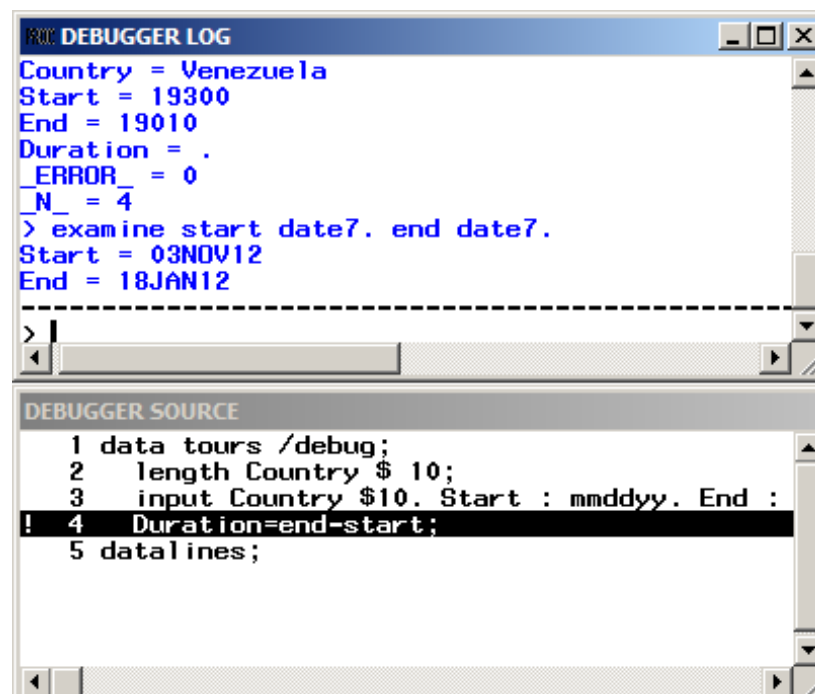
DEBUGGER SOURCE
1 data tours /debug;
2   length Country $ 10;
3   input Country $10. Start : mmddyy. End :
! 4   Duration=end-start;
5 datalines;

```

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, 20120, 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 DDMMYY_W. format:

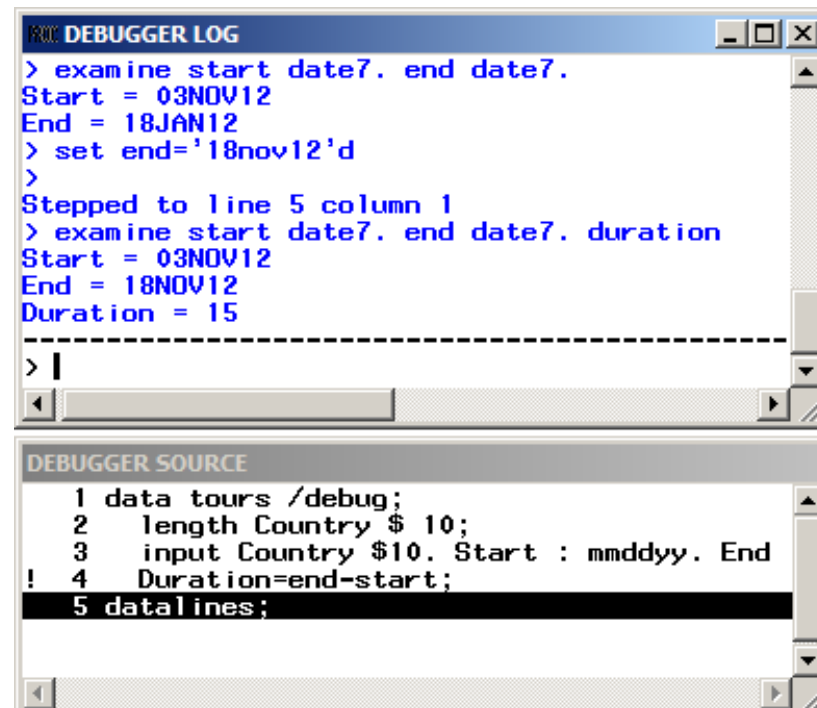
```
set end='18nov00'd
```

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

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

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

The following display shows the results:



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

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

```
/* corrected version */

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;

```



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 WEEKDATEw. 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
```

Chapter 3

Dictionary of DATA Step Debugger Commands

DATA Step Debugger Commands by Category	25
Dictionary	26
BREAK	26
CALCULATE	28
DELETE	29
DESCRIBE	30
ENTER	31
EXAMINE	31
GO	32
HELP	33
JUMP	34
LIST	35
QUIT	36
SET	36
STEP	37
SWAP	38
TRACE	38
WATCH	39

DATA Step Debugger Commands by Category

Category	Language elements	Description
Controlling Program Execution	GO (p. 32)	Starts or resumes execution of the DATA step.
	JUMP (p. 34)	Restarts execution of a suspended program.
	STEP (p. 37)	Executes statements one at a time in the active program.
Controlling the Windows	HELP (p. 33)	Displays information about debugger commands.
	SWAP (p. 38)	Switches control between the SOURCE window and the LOG window.
Manipulating DATA Step Variables	CALCULATE (p. 28)	Evaluates a debugger expression and displays the result.
	DESCRIBE (p. 30)	Displays the attributes of one or more variables.

Category	Language elements	Description
Manipulating Debugging Requests	EXAMINE (p. 31)	Displays the value of one or more variables.
	SET (p. 36)	Assigns a new value to a specified variable.
	BREAK (p. 26)	Suspends program execution at an executable statement.
	DELETE (p. 29)	Deletes breakpoints or the watch status of variables in the DATA step.
	LIST (p. 35)	Displays all occurrences of the item that is listed in the argument.
	TRACE (p. 38)	Controls whether the debugger displays a continuous record of the DATA step execution.
Tailoring the Debugger	WATCH (p. 39)	Suspends execution when the value of a specified variable changes.
	ENTER (p. 31)	Assigns one or more debugger commands to the ENTER key.
Terminating the Debugger	QUIT (p. 36)	Terminates a debugger session.

Dictionary

BREAK

Suspends program execution at an executable statement.

Category: Manipulating Debugging Requests

Alias: B

Syntax

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

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 non-zero, 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” on page 29
- “WATCH” on page 39

CALCULATE

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 11](#)

DELETE

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

Category: Manipulating Debugging Requests

Alias: D

Syntax

DELETE BREAK *location*

DELETE WATCH *variable(s)* | `_ALL_`

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” on page 26](#)
- [“WATCH” on page 39](#)

DESCRIBE

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

Assigns one or more debugger commands to the ENTER key.

Category: Tailoring the Debugger

Syntax

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

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

Displays the value of one or more variables.

Category: Manipulating DATA Step Variables

Alias: E

Syntax

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

EXAMINE ALL <*format*>

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” on page 30](#)

GO

Starts or resumes execution of the DATA step.

Category: Controlling Program Execution

Alias: G

Syntax

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

Without Arguments

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

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” on page 34](#)
- [“STEP” on page 37](#)

HELP

Displays information about debugger commands.

Category: Controlling the Windows

Syntax

HELP

Without Arguments

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

JUMP

Restarts execution of a suspended program.

Category: Controlling Program Execution

Alias: J

Syntax

JUMP *line-number* | *label*

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” on page 32
- “STEP” on page 37

LIST

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

Category: Manipulating Debugging Requests

Alias: L

Syntax

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

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” on page 26](#)
- [“DELETE” on page 29](#)
- [“WATCH” on page 39](#)

QUIT

Terminates a debugger session.

Category: Terminating the Debugger

Alias: Q

Syntax

QUIT

Without Arguments

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

Details

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

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

SET

Assigns a new value to a specified variable.

Category: Manipulating DATA Step Variables

Alias: None

Syntax

SET *variable=expression*

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

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” on page 32
- “JUMP” on page 34

SWAP

Switches control between the SOURCE window and the LOG window.

Category: Controlling the Windows

Alias: None

Syntax

SWAP

Without Arguments

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

TRACE

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

Category: Manipulating Debugging Requests

Alias: T

Default: OFF

Syntax

TRACE <ON | OFF>

Without Arguments

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

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
```

Index

Special Characters

%DS2CSV macro [3](#)

B

BREAK [26](#)

BREAK command

DATA step debugger [26](#)

C

CALCULATE [28](#)

CALCULATE command

DATA step debugger [28](#)

comma-separated value (CSV) files [3](#)

CSV files [3](#)

D

data sets

converting to CSV files [3](#)

DATA step debugger [9](#)

assigning commands to ENTER key [31](#)

assigning commands to function keys
[11](#)

assigning new variable values [36](#)

continuous record of DATA step
execution [38](#)

customizing commands with macros [12](#)

DATA step generated by macros [12](#)

debugger sessions [10](#)

debugging, defined [9](#)

debugging DO loops [23](#)

deleting breakpoints [29](#)

deleting watch status [29](#)

description of [10](#)

displaying variable attributes [30](#)

displaying variable values [31](#)

entering commands [11](#)

evaluating expressions [28](#)

examples [13](#)

executing statements one at a time [37](#)

expressions and [11](#)

formats and [18](#)

formatted variable values [24](#)

help on commands [33](#)

jumping to program line [34](#)

listing items [35](#)

macro facility with [12](#)

macros as debugging tools [12](#)

quitting [36](#)

restarting suspended programs [34](#)

resuming DATA step execution [32](#)

starting DATA step execution [32](#)

suspending execution [26, 39](#)

switching window control [38](#)

windows [11](#)

DEBUGGER LOG window [11](#)

DEBUGGER SOURCE window [11](#)

debugging

See [DATA step debugger](#)

DELETE [29](#)

DELETE command

DATA step debugger [29](#)

DESCRIBE [30](#)

DESCRIBE command

DATA step debugger [30](#)

DO loops

debugging [23](#)

E

ENTER [31](#)

ENTER command

DATA step debugger [31](#)

EXAMINE [31](#)

EXAMINE command

DATA step debugger [31](#)

expressions

DATA step debugger and [11](#)

F

formats

DATA step debugger and 18

G

GO 32

GO command

DATA step debugger 32

H

HELP 33

HELP command

DATA step debugger 33

J

JUMP 34

JUMP command

DATA step debugger 34

L

LIST 35

LIST command

DATA step debugger 35

LOG window

DATA step debugger 38

M

macro facility

DATA step debugger with 12

macros

as debugging tools 12

customized debugging commands with

12

debugging a DATA step generated by

12

Q

QUIT 36

QUIT command

DATA step debugger 36

S

SET 36

SET command

DATA step debugger 36

SOURCE window

DATA step debugger 38

STEP 37

STEP command

DATA step debugger 37

SWAP 38

SWAP command

DATA step debugger 38

T

TRACE 38

TRACE command

DATA step debugger 38

W

WATCH 39

WATCH command

DATA step debugger 39