ABSTRACT

In Release 6.07 of base SAS software for non-UNIX platforms, a full-screen source level debugger will be supplied for the DATA step on an experimental basis. This paper is an introductory discussion of the debugger’s capabilities.

INTRODUCTION

One of the top vote-getting items on the SASware Ballot in the last few years has been for the Institute to supply a source-level debugger for the DATA Step. With the Release 6.07 for non-UNIX platforms the debugger will be available in base SAS software as an experimental software feature. This means that there will be limited technical support and preliminary documentation available.

This paper is intended to provide an overview of the debugger's operation and command syntax, to demonstrate its usefulness compared with current debugging techniques and to offer some instructions on using the macro facility to write customized debugger commands that may prove useful in a particular user's environment.

DEBUGGING BACKGROUND: BUGS IN DATA STEP PROGRAMS

Anyone who has written a DATA step program knows that there is a chance that things will not go as planned. These problems fall into two basic categories: syntax errors and execution errors.

Syntax Errors

Syntax errors are errors in the specification of the DATA step language that prevent the DATA step from executing. The DATA step cannot do something that it cannot understand. The SAS System issues error messages from the compiler that tell the user which part of the program the compiler is confused about. For example, suppose you get the C programming language mixed up with the SAS System DATA step language:

```sas
data; x = y; run;
```

100
ERROR 180-322: Statement is not valid or it is used out of proper order.

NOTE: The SAS System stopped processing this step because of errors.

With the 180 message, the SAS System is telling you that it has no idea what your program was intending to do. These problems are easily fixed, usually by checking the SAS documentation and making corrections.

Execution Errors

Execution errors are errors that cause unexpected results from the DATA step execution. Some execution errors may be like syntax errors; they are errors in the DATA step language specification:

```sas
1? data; input x format x data7. ; cards;
2x 1000
3x 1100
4x 1101
5x ;
```

NOTE: LOST CARD.

The intention of this program was to have one variable, X, with the format DATE7. in the data set. But the LOST CARD message in the first NOTE statement indicates that something went wrong.

Other execution errors are mistakes made in the logic of the program. For example, suppose you need to determine when Washington's Birthday is celebrated. You might code:

```sas
if day = '2feb91'd then holiday = 1;
else holiday = 0;
```

But this is not correct since Washington’s Birthday is celebrated on the third Monday in February. Execution errors are the errors that the DATA step debugger can help find. The debugger will be the most helpful in finding logical errors.

Current Debugging Techniques

What techniques are employed currently to help find execution errors? These techniques can best be described as brute force. Probably what most people do is to scatter PUT statements at strategic locations and print the values of the offending variables to try to see what happens. For example, let’s say that you confuse the C and SAS languages again:

```sas
data;
  x = y = .;
...
1000 lines of code ....
run;
```

The intention is to initialize X to missing as well as Y. Then you execute the DATA step to find that X has not been set to missing. If you add the PUT X= statement after the assignment statement you’ll see that X’s value after the assignment statement is one and not missing. This is because the statement ‘y = .;’ is treated in this case as a logical operation that returns a true or false value and is not an assignment statement.

The solution to this problem is fairly simple once you’ve placed the PUT statement in the correct place. However, finding the correct place may be like the needle in the haystack problem since there is no telling where you will detect that the value of X has gone awry in the 1000 lines of code or after how many DATA step iterations. So, from the example above, you might start out with:

```sas
data;
  x = y = .;
...
500 lines of code ....
put x = y;
...
500 lines of code ....
run;
```

From there you would probably move the PUT statement closer to the top of the program and execute the program again and again.

The DATA Step Debugger: Bug Hunting Made Easy
Mark Schaffer, SAS Institute Inc., Cary, NC
Brenda Kalt, SAS Institute Inc.
until you have the PUT statement immediately following the second line.

Also, for each time that you wish to change the point where the values are dumped, you have to initiate a new execution. This method will consume a lot of CPU time and programming effort since you have to read over and try to make sense out of the values that are dumped.

Another technique available is the use of options on the FILE and INFILE statements that display information about the file or infile being processed. The FILE and INFILE statements have the FILENAME=, LINE=, LINESLEFT= and COLUMN= options. When debugging a problem with a FILE or INFILE statement, specify these options and print them using the PUT statement at strategic points in the program. This approach creates the same problems as described with using the PUT statement.

**OVERVIEW OF THE DEBUGGER**

**Description**

The DATA step debugger is a full-screen interactive environment that allows you to control the execution of a DATA step program. The DATA step debugger, with the use of windows, simultaneously displays the source that is being executed, accepts input that controls execution, and displays information about the program.

The DATA step debugger allows a DATA step user to:
- display the value of a variable (EXAMINE)
- examine the characteristics of a variable (DESCRIBE)
- check calculations of expressions (CALCULATE)
- suspend execution when a variable's value changes (WATCH)
- change execution flow by resuming execution at a different line number (JUMP)
- change a variable's value (SET)
- display the current state of FILE and INFILE processing (LIST).

Start the debugger with the DEBUG option specified in the RUN or CARDS statement. The debugger is valid when the SAS System is in display manager or non-display manager modes. The debugger's windows will cover up your other SAS windows. The debugger starts with two windows:

1. a source window where the current range of the source is displayed along with a highlighted line (the line to be executed next)
2. a dialog window that has a command pane where debugger commands are specified and a log pane where the commands are echoed along with the commands' responses.

How the DATA Step Executes Under the Debugger

When you submit a DATA step with the DEBUG option in the RUN or CARDS statement, the SAS System compiles the step, displays the debugger windows, and waits for input from the command pane of the dialog window. The commands are instructions that control and display information about the execution of the DATA step.

The DATA step iterates as many times as when executed without the DEBUG option. When last iteration has finished, a message appears in the DEBUGGER LOG window, stating that the DATA step has finished and there is no active program.

At this point you can still get information about variables but you cannot continue execution. To start the DATA step again you must issue a QUIT command and submit the DATA step again.

**An Example of Using the Debugger**

Suppose you are a programmer for a mail-order clothing company and you have written a program to check the current inventory of various items. Your current stock of items should match the initial stock after all adjustments for sales and other changes have been made. You write the following program to compare the starting and current inventories:

```sql
data temp;
  set screen.stock;
  return = start - sold + dislike;
  endstock = start - sold + return;
  proc print; run;
```

The output from the PROC PRINT step looks like the following:

<table>
<thead>
<tr>
<th>ITEM</th>
<th>START</th>
<th>SELL</th>
<th>SCREEN</th>
<th>DISLIKE</th>
<th>TODAY</th>
<th>RETURN</th>
<th>ENDS</th>
</tr>
</thead>
<tbody>
<tr>
<td>coat</td>
<td>150</td>
<td>50</td>
<td>5</td>
<td>5</td>
<td>105</td>
<td>5</td>
<td>105</td>
</tr>
<tr>
<td>dress</td>
<td>200</td>
<td>75</td>
<td>2</td>
<td>2</td>
<td>117</td>
<td>11</td>
<td>117</td>
</tr>
<tr>
<td>shoes</td>
<td>90</td>
<td>60</td>
<td>5</td>
<td>5</td>
<td>90</td>
<td>5</td>
<td>90</td>
</tr>
<tr>
<td>ties</td>
<td>100</td>
<td>60</td>
<td>0</td>
<td>0</td>
<td>10</td>
<td>50</td>
<td>50</td>
</tr>
</tbody>
</table>

The values for ENDS and TODAY do not match. So you decide to use the handy DATA step debugger to find the problem. First you need to resubmit the program with the DEBUG option on the RUN statement after the DATA step:

```sql
data temp;
  set screen.stock;
  return = start - sold + dislike;
  endstock = start - sold + return;
run debug;
```

You see that our current line is line number 2. At this point DATA step execution is waiting to begin. Your first hypothesis is that something is wrong with the value of the variable ENDSTOCK. So you set a break point at line 4 by issuing the debugger command `break 4`.

The line is marked in the DEBUGGER SOURCE window with an exclamation point (!) to the left of the line number. Next you start execution with the GO command.

Now the current line is pointing to line 4, which means that the debugger has stopped before line 4 has been executed. You examine all the pertinent variables:

```
examine endstock start sold return today
```

and the debugger displays

```
ENDSTOCK = 150
START = 101
SOLD = 50
RETURN = 0
TODAY = 101
```

Check the anticipated calculation of ENDSTOCK by using the CALCULATE command:

```
calc start=sold; calc start - sold + return;
```

The debugger displays

```
100
```
Issue the STEP command to execute line 4 and use the EXAMINE command again:

```
step; examine endstock;
```

The debugger displays
```
ENDSTOCK + 105
```

We see that ENDSTOCK seems to have been correctly calculated. So this has been a dead end. You try something else. The calculation of the variable RETURN should be examined next. So you get rid of the old breakpoint and set a new one:

```
delete break 4; break 3;
```

Repeat what you did for the ENDSTOCK variable:

```
go; examine return spoil size dislike;
```

The debugger displays
```
RETURN = 10
SPOIL = 2
SIZE = 10
DISLIKE = 2
```

The values for SPOIL, SIZE and DISLIKE have changed since you are now at the second observation. But you can still calculate the anticipated value:

```
calc spoil*size; calc spoil*size*dislike;
```

The debugger displays
```
12
14
```

We check the calculation against the value for RETURN after line 3 is executed:

```
step; examine return;
```

The debugger displays
```
RETURN = 14
```

and everything is still fine.

So now you decide to look further at the calculation of RETURN and notice that SPOIL + SIZE is the same as the difference between ENDSTOCK and TODAY:

```
calc spoil + size; calc endstock - today;
```

The debugger displays
```
12
12
```

You realize that spoiled goods are not returned to inventory, while items returned because they are the wrong size are replaced by items from inventory. Neither of these variables should count in the ENDSTOCK calculation. So you change the calculation of ENDSTOCK to:

```
endstock = start - sold + dislike;
```

and the problem is solved.

Other Commands

Now that you have seen an example of the debugger’s usefulness, here are some hypothetical errors where some additional debugging commands and features can be helpful.

The DESCRIBE Command

Suppose that you have the following OAT A step:
```
data temp;
set screen.10001;
input x;
cards;
```

You get the following message on the log:
```
ERROR: Invalid data for x in line 3.
```

```
01
02
03
04
05
06
07
08
09
10
```

```
x= 0
```

```
NOTE: 1 record was read from the input file.
The minimum record length was 1.
The maximum record length was 1.
```

Well, if you use the DESCRIBE command to see if the variable X has some sort of problem, as shown:
```
DESC X;
```

You see that your variable X has an informat associated with it:
```
Name = x, Type = WM, Length = 8
Informat = DATE7.
```

The EXAMINE Command

Suppose that you have the following program:
```
data a;
retain daypast 0;
input pastday date1.;
if pastday > '01jan90'd then pastday+1;
cards;
```

This program produces the following output:
```
Output 2 Errors in Dates
```

```
ERR=0001; ERR=0002;
```

You see that everything is input correctly, but the variable PASTDAY changes after the IF statement. You realize that you have mixed up your variable names, since their names are very similar.

The WATCH Command

Another way to debug the previous problem is to use the WATCH command. The WATCH command suspends the program’s execution when the specified variable’s value changes. If you specify

```
watch pastday;
go;
```

You see that everything is input correctly, but the variable PASTDAY changes after the IF statement. You realize that you have mixed up your variable names, since their names are very similar.

```
```

Of course you are concerned that the dates have changed from the way they appeared in the CARDS statement. You can use the EXAMINE command to output the value with a format if the associated format is not the desired one:
```
break 1; go;
```

```
proc print; format pastday date7.; run;
```

You see that everything is input correctly, but the variable PASTDAY changes after the IF statement. You realize that you have mixed up your variable names, since their names are very similar.

The WATCH Command

Another way to debug the previous problem is to use the WATCH command. The WATCH command suspends the program’s execution when the specified variable’s value changes. If you specify

```
watch pastday;
go;
```

The debugger suspends execution after the INPUT statement is executed and again after the statement PASTDAY+1 has been executed. This is something that you did not expect, and hopefully you will see your error.

The WATCH command will cause the program to be suspended at the top of each DATA step iteration since all nonretained variables are set to missing at this point. (A remedy for this will be discussed later.) Suppose in the example you were upset about the fact that the DAYPAST variable did not change and decided to watch it instead, as shown:
```
watch daypast; go;
```

The debugger would allow the DATA step to execute to completion since DAYPAST is specified in the RETAIN statement.

The CALCULATE Command

Both the SET and CALCULATE commands accept SAS datetime constants as valid values, which can sometimes be useful. Suppose you need to calculate the difference between two dates. If you want to know how many days before the next SUGI begins, use the following command:

\[
\text{calc '12apr92'd - '19feb91'd}
\]

You will see that it is 418 days.

Similarly, suppose you need to change a variable to a date that is 10 weeks from another date, you could use the following SET command:

\[
\text{set x='01jan91'd + 70;}
\]

This also works with SAS datetime constants.

The LIST FILE and LIST INFILE Commands

One thing that frustrates many people is finding bugs in columns and line-directed input and output. This occurs in DATA steps that use the FILE or INFILE statements with the column pointer modifiers (@ and #) and the line pointer modifier (#) in the INPUT and PUT statements. As previously mentioned the current way to debug these problems is to specify options in the FILE or INFILE statements and dump their values at strategic points in the program.

All this can be done by using the LIST FILES or LIST INFILES commands from the debugger:

\[
\text{list infiles}
\]

This displays the name, current column, current line, the line length and the contents of the line:

\[
\begin{align*}
\text{Name} & : x \\
\text{Column} & : 2 \\
\text{Line} & : 1 \\
\text{Length} & : 1 \\
\text{Current buffer} & : 1 2 3 4 5 6 7 8
\end{align*}
\]

The buffer is not read; therefore it is not available to the LIST INFILES commands until after the INPUT statement.

The IF-THEN/ELSE Commands

The debugger also enables you to execute commands conditionally with the IF-THEN/ELSE commands. This construct works the same as in the DATA step, except the statements conditionally executed are debugger commands and not DATA step statements. Suppose you want to check your data for errors as values are input in the following program:

\[
\begin{align*}
\text{data;}
\text{input x;}
\text{run;}
\text{run debug;}
\end{align*}
\]

To examine the contents of the input buffer when an error occurs, use the following commands:

\[
\begin{align*}
\text{b 5 do; if _error_ = 1 then list infiles; else ex x; ex _m_; end;}
\end{align*}
\]

When there are no errors the debugger outputs

\[
\begin{align*}
\text{Break at line 5 col 2}
\text{X = 1}
\text{_m_ = 1}
\end{align*}
\]

When an error occurs the debugger outputs

\[
\begin{align*}
\text{macro _m_ watch(x); watch x; go; if x = . then go; gend;} \\
\text{macro my.go(x); go; if x = . then go; gend;}
\end{align*}
\]

These macros would be specified in the debugger command line as

\[
\begin{align*}
\text{xmy.watch(daypast)}
\text{xmy.go(daypast)}
\end{align*}
\]

These macros could be included in an autocall library and could be written as command-style macros also. Macros are also available throughout your SAS System session, which means that macros defined in one DATA step debugger invocation are available in all subsequent debugger invocations until the end of your session.

HOW THE DEBUGGER RELATES TO THE REST OF THE SAS SYSTEM

Windows

All SAS window functions and commands are available on the debugger windows. The CATALOG, VAR, and LIBNAME window may be helpful during debugging sessions. These windows are available regardless of whether the debugger was initiated from a display manager or non-display manager session of the SAS System. If you get lost in the maze of windows, you can get back to the debugger's window by using the NEXT window command followed by the debugger's window name, either DEBUGGER SOURCE or DEBUGGER LOG. To use this in a key definition, specify:

\[
\text{next 'DEBUGGER LOG'}
\]

as the key definition with the window title in quotes.

Two additional window commands are supported. The RECALL command brings back the previous command entered into the debugger's command line. The debugger keeps a circular queue of the last 20 commands so you can travel back through your old commands, change and submit them.

A new window command called SWAP will switch control between the two windows of the debugger. This helps with accessing the DEBUGGER SOURCE window for scrolling and the DEBUGGER LOG window for entering commands.

Keys

The debugger also accepts input from a set of function keys that you define for the debugger and are accessed via the KEYS window. If you want to save yourself some typing, you can define a key to examine all the variables in the program:

\[
\text{dd examine _all_;}
\]

To examine and describe all the variables in the program, use the following:

\[
\text{dd examine _all_; describe _all_}
\]

The Debugger and the Macro Facility

Since the debugger commands are input through the same source manager as the rest of the SAS System, the debugger commands can use the macro facility to write commands that are customized for certain DATA step programs. One macro for general use is one that sets a watchpoint and continues execution when the program is suspended at the top of the DATA step where variables are set to missing:

\[
\begin{align*}
\text{macro my.watch(x); watch x; go; if x = . then go; gend;}
\end{align*}
\]

These macros could be included in an autocall library and could be written as command-style macros also. Macros are also available throughout your SAS System session, which means that macros defined in one DATA step debugger invocation are available in all subsequent debugger invocations until the end of your session.
Also available with Release 6.07 for non-UNIX platforms is the ability to use command-style macros in display manager command lines and the debugger. This allows you to build your own commands that look like real commands. The CMDMAC option needs to be in effect. The previous example would be specified as

```
%macro my_watch(s)/cmd; watch &s; go; if &s = THEN go; %end;
%macro my_go(s)/cmd; go; if &s = THEN go; %end;
```

It could be specified in the debugger command line as

```
my_watch daypast
and
my_go daypast
```

The %INCLUDE statement is available and can be used to include a series of commands that do not resume program execution. If there are commands that resume and stop program execution, like GO statements with break and watch points set, some of the commands are lost. However, %INCLUDE can be useful to include a series of macro definitions that the debugger will use.

### Pull-down Menus

The debugger has its own set of pull-down menus; one for the DEBUGGER SOURCE window and one for the DEBUGGER LOG window. These pull-down menus operate the same as other pull-down menus. Their definition is customized to the debugger.

### CONCLUSION

The DATA step debugger's inclusion into the SAS System provides users that write DATA step applications with an effective debugging tool that saves programming and computer time.