ABSTRACT

Release 6.11 of base SAS® software contains a full-screen, source-level debugger for the DATA step. 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 SAS Institute to supply a source-level debugger for the DATA step. An experimental version was introduced with Release 6.07 (for platforms other than UNIX). With Release 6.11, the DATA step debugger is now a production component with full technical support and documentation available.

This paper provides an overview of the debugger's operation and demonstrates its usefulness compared with current debugging techniques. An example of using the debugger and detailed information about the debugging commands used in the example are also given.

DEBUGGING: 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 DATA step compiler that tell the user which portion of the program is confusing. For example, suppose you get the C programming language mixed up with the SAS System DATA step language:

```
  (tata, z -= y; rw;i;
1.3 data, z -= y,:
  180
ERROR 1.80-322; Statement 1. DO ... 1i4 ot it i. out of propel;
  or4er.
```

With the 180 message, the SAS System is telling you that it has no idea what your program was intending to do. It can make no sense of the statement 'Z -= Y;'. (This is a C language construct which correlates to 'Z = Z-Y;'.) These problems are easily fixed, by checking the SAS documentation and making corrections.

Execution Errors

Execution errors occur after the language statements have been compiled into executable code. The compiler understood the specification of the DATA step language, but the execution of the language statements produces a result that you either do not expect or do not understand.

It is possible that an execution error is actually a hidden specification error. A statement may have meaning, but it may not be the meaning you had intended. Note the missing semicolon below:

```plaintext
  if data; input x format x date7.; cards;
  2 1999
  3 1100
  4 1101
  5 ;
  NOTE: LOST CARD.
```

```
ERROR 1.80-322; Statement 1. DO ... 1i4 ot it i. out of propel;
```

The intention of this program was to have one variable, X, with the format DATE7, in the data set. But the LOST CARD and 'SAS went to a new line...' messages in the two NOTE statements indicate that the INPUT statement attempts to process more than the one variable.

The most common execution errors, however, are mistakes in the logic of the program. For example, suppose you need to validate a given month specification before generating a date. Your code might look like this:

```plaintext
  data day.
  input .ol1th yea
  if (month < 1 or month > 12) then day = '01Jan60';
  else day = mdy(month,1,year);
  run;
```

Because of the incorrect IF logic, you will see 'invalid argument' notes from the MDY function, instead of dates of 01JAN60 for months specified incorrectly:

```
ERROR 1.80-322; Statement 1. DO ... 1i4 ot it i. out of propel;
```

These are the types of errors that the DATA step debugger can help isolate and find.

Current Debugging Techniques

Current techniques for debugging DATA step programs are limited in number and scope. These techniques can best be described as brute force. The most common is to scatter PUT statements at strategic locations and print the values of the offending variables. This allows you to see the flow of control and data in your program. For example, let's say that you confuse the C and SAS languages again:

```plaintext
  ... 1000 lines of code ...

  run;
```

The intention is to initialize X, as well as Y, to missing. Then you execute the DATA step to find that X has not been set to missing. If you add a "PUT X=" statement after the assignment statement you'll see that the value of X after the assignment statement is 1
and not missing. This is because the statement "x = y = ." treats
"y = ." not as an assignment, but as a logical operation that returns
a true or false value.

The solution to this problem is fairly simple once you have the PUT
statement in the correct place. However, finding the correct place
may be like the 'needle in the haystack' problem. There is no way
to tell where you can detect that the value of X has gone away in
the 1000 lines of code or after how many DATA step iterations.
So, from the example above, you might start out with:

data;
   x = y = .;
   ... 500 lines of code ...
   put x=y;
   ... 500 lines of code ...
end;

From there you would begin moving the PUT statement closer to
the top of the program, executing the program again and again 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 could consume a lot of CPU time and programming effort,
depending on the number of iterations of this process that are
required.

For debugging problems with file or infile processing, options are
available in the FILE and INFILE statements. These options return
information about the file or infile being processed. The
FILENAME=, LINE=, COLUMN=, LENGTH=, and LINESLEFT= options return
the name of the current file, the current line number in an N= block, the current column number, the current line length,
and the number of output lines left on the current page, respective­
ly. (There may also be other options specific to your host.) When
debugging a problem with a FILE or INFILE statement, specify any
of these options and print the returned information using PUT
statements at strategic points in the program. Of course, this
approach has the same problems as described earlier.

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, through the use of windows, simulta­
nously 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:

- suspend execution at certain lines of source (BREAK)
- 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 debug session (LIST).

You start the debugger with the /DEBUG option specified on the
DATA statement (DATA TEST / DEBUG). The /DEBUG option is
valid when the SAS System is in display manager or non-display
manager modes. The debugger consists of two windows:

- a DEBUGGER SOURCE window in which the current
  range of the source is displayed along with a highlighted
  line (the line to be executed next)
- a DEBUGGER LOG window with a dialog-command pane
  in which commands are specified and a log pane in which
  commands are echoed along with command responses.

How the DATA Step Executes under the Debugger

When you submit a DATA step with the /DEBUG option in the
DATA statement, the SAS System compiles the step, displays the
debugger windows, and waits for input from the dialog-command
pane of the DEBUGGER LOG 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 it does when executed
without the /DEBUG option. When the 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 resubmit the DATA step.

An Example of Using the Debugger

Suppose you are a tour director for a travel agency. In your data
files, some of the records contain tour information and some
contain customer information. You write a program to give you
information about your tour participants, including name, age, sex,
which tour they are taking, and any unpaid balance. Your program
might look like:

   1 data tour; keep-tour name age sex unpaid;
   2 input #1 type $ price $;
   3 if (type='B') then do;
   4   input tour $10;
   5   fare = price;
   6   end;
   7   else if type='P' then do;
   8   input name $10. age 2. +x sex $1.;
   9   unpaid = fare - price;
   10 output;
   11 end;
   12 cards;
         Tour 101  B  2505
         Maryann R 43 F  1300
         Rose W  38 F  200
         Edward R 37 M  2595
         Oliver R 71 M  2055
         Tour 102  B  1605
         Sylvia V 46 F  1809
         William B 22 M  300
         Barry J  41 F  200
         Shinkle B 30 F  540
   21 go to;
   22 proc print;
   23 run;

Output 1 on the following page shows the results.
As you can see, the program did not work very well—all values are missing! So you decide to use your trusty DATA step debugger to find the problem. First, you need to resubmit the entire DATA step program with the /DEBUG option in the DATA statement.

```
data tours (keep=tour name age sex unpaid) / debug;
```

After resubmitting the program with the /DEBUG option, the debugger windows appear. At this point, DATA step execution is waiting to begin. The debugger has paused before the first executable statement of the program (line number 2) and highlighted it in the DEBUGGER SOURCE window. You decide you want to see what happens when the variable TOUR is input. So, you set a breakpoint 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. You then start execution with the GO command. Note that while all examples use full command names, most commands do have aliases or abbreviations. (See Appendix 1, "Command Summary".)

Now line 4 is the current line, which means that the debugger has stopped before line 4 has been executed. You examine the pertinent variables and list the INFILE status:

```
examine type tour price; list infiles
```

and the debugger displays:

```
TYPE = H
TURD = TOUR
PRICE = 2595
INFILES = Logical name = CARDS
Column = 43, Line = 1, Length = 80
Current buffer = Tour 101
```

It looks okay, so you issue the STEP command to execute line 4, and then look at things again:

```
step; examine tour price; list infiles
```

The debugger displays:

```
Stepped to line 5 column 4
TOUR = TOUR
PRICE = 2595
INFILES = Logical name = CARDS
Column = 43, Line = 1, Length = 80
Current buffer = Tour 101
```

From this, you realize one of the problems. After the INPUT statement on line 4, TOUR is still blank. Looking at the LIST INFILES output, you see that the current column after INPUT is 43, and that TOUR should have been read from the first 20 columns of the record. You realize that you need to reposition the column pointer before reading TOUR because the previous INPUT statement (with the single-trailing @) leaves the column pointer in column 43.

Looking at the rest of your program, it looks like this same type of error could cause NAME, AGE, and SEX to be missing. To test this theory, you set a breakpoint after the next INPUT statement, at line 9. Because you will not be needing the first breakpoint anymore, you delete it.

```
break 9; delete break 4; go
```

Line 9 is now the current line. Look at the values for NAME, AGE, and SEX, and list the INFILE status:

```
examine name age sex; list infiles
```

The debugger displays:

```
NAME = 
AGE = 
SEX = 
INFILES
Logical name = CARDS
Column = 57, Line = 1, Length = 80
Current buffer = Maryann B 43 F  
P 1200
```

Your theory is proven correct. The third INPUT statement also needs to reposition the column pointer before reading NAME, AGE, and SEX.

The corrected program and its results (Output 2) appear below. As you can see, you are not finished debugging this program, as the TOUR and UNPAID variables remain missing.

```
data tours (keep=tour name age sex unpaid) / debug;
input 935 type $ price 91
if (type='B') then do;
  input 91 tour $:20.1
  fare = price;
  ,
  else if type='P' then do;
    input 91 D&lMI $10. age sex $1.
  unpaid = fare - price;
  9 output;
end;
cards;
proc print data=tours; run;
```

```
As before, you resubmit the program with the /DEBUG option on the DATA statement, and the debugger pauses before line 2. You want to see the variable TOUR after it is input. To do this, set a breakpoint on line 5 and issue GO. When the debugger pauses at line 5, enter:

```
examine tour price; list infiles
```

and the debugger displays:

```
TOUR = Tour 101
PRICE = 2595
```

Because TOUR and PRICE have the correct values, that portion of the problem has indeed been fixed. You decide to look at things just before UNPAID is computed and the observation is output.

```
break 9; go
```

Now, at line 9, you look at the variables in question. UNPAID is a combination of FARE and PRICE, so you examine TOUR, FARE, and PRICE. The resulting display is:

```
examine tour fare price; list infiles
```

As before, you resubmit the program with the /DEBUG option on the DATA statement, and the debugger pauses before line 2. You want to see the variable TOUR after it is input. To do this, set a breakpoint on line 5 and issue GO. When the debugger pauses at line 5, enter:

```
examine tour price; list infiles
```

and the debugger displays:

```
TOUR = Tour 101
PRICE = 2595
```

Because TOUR and PRICE have the correct values, that portion of the problem has indeed been fixed. You decide to look at things just before UNPAID is computed and the observation is output.

```
break 9; go
```

Now, at line 9, you look at the variables in question. UNPAID is a combination of FARE and PRICE, so you examine TOUR, FARE, and PRICE. The resulting display is:

```
examine tour fare price; list infiles
```

As before, you resubmit the program with the /DEBUG option on the DATA statement, and the debugger pauses before line 2. You want to see the variable TOUR after it is input. To do this, set a breakpoint on line 5 and issue GO. When the debugger pauses at line 5, enter:

```
examine tour price; list infiles
```

and the debugger displays:

```
TOUR = Tour 101
PRICE = 2595
```

Because TOUR and PRICE have the correct values, that portion of the problem has indeed been fixed. You decide to look at things just before UNPAID is computed and the observation is output.

```
break 9; go
```

Now, at line 9, you look at the variables in question. UNPAID is a combination of FARE and PRICE, so you examine TOUR, FARE, and PRICE. The resulting display is:

```
examine tour fare price; list infiles
```

As before, you resubmit the program with the /DEBUG option on the DATA statement, and the debugger pauses before line 2. You want to see the variable TOUR after it is input. To do this, set a breakpoint on line 5 and issue GO. When the debugger pauses at line 5, enter:

```
examine tour price; list infiles
```

and the debugger displays:

```
TOUR = Tour 101
PRICE = 2595
```

Because TOUR and PRICE have the correct values, that portion of the problem has indeed been fixed. You decide to look at things just before UNPAID is computed and the observation is output.

```
break 9; go
```

Now, at line 9, you look at the variables in question. UNPAID is a combination of FARE and PRICE, so you examine TOUR, FARE, and PRICE. The resulting display is:
TOUR and FARE (and thus UNPAID) are missing! This is when you realize that TOUR and FARE were given values in the previous iteration of the DATA step and were reset to missing at the beginning of this iteration. These variables need to be retained by using the RETAIN statement in order to keep their values across iterations.

If the program is fairly complex, you may want to test your hypothesis concerning the problem. To do this, you can use the SET command to correct the values of the TOUR and FARE variables, and let execution proceed.

```sas
set tour=r101; set fare=2995
```

The debugger displays:

```
TOUR = Tour 101
FARE = 2995
```

You may also want to validate that the UNPAID computation will give the expected results. You can either let execution proceed until UNPAID has been computed and examine the result, or you can use the CALCULATE command at the current location to compute the result. Using CALCULATE, you see the following:

```sas
calculate fare - price 1295
```

This is the expected result. You could exit the debugger at this point using the QUIT command. But you want to verify that the first observation has been generated correctly. Deleting all breakpoints, you let execution continue to completion and then exit the debugger:

```sas
delete break call; go: quit
```

Looking at the results, you see that the first observation is correct. With the added RETAIN statement, the program should now be correct. Below is the corrected DATA step program, and Output 3 displays the expected results:

```sas
1 data tours (keep=tour name age sex unpaid);
2 retain fare 0 tour,
3 input daytday create7,
4 if (type='P') then do;
5 input 0 name $10, age 3, <1 sex $1,
6 unpaid = fare - price;
7 output;
8 end;
9 else if type='F' then do;
10 input 0 name $10, age 3, <1 sex $1,
11 output;
12 end;
13 cards;
... data lines ...
24 ;
25 proc print data=tours;
26 run;
```

### Output 3 Data Set TOURS

<table>
<thead>
<tr>
<th>OBS</th>
<th>TOUR</th>
<th>NAME</th>
<th>AGE</th>
<th>SEX</th>
<th>UNPAID</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Tour 101</td>
<td>Maryann</td>
<td>45</td>
<td>F</td>
<td>1395</td>
</tr>
<tr>
<td>2</td>
<td>Tour 101</td>
<td>Rose A</td>
<td>38</td>
<td>F</td>
<td>2195</td>
</tr>
<tr>
<td>3</td>
<td>Tour 101</td>
<td>Edward P</td>
<td>37</td>
<td>M</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>Tour 101</td>
<td>Oliver S</td>
<td>75</td>
<td>M</td>
<td>0</td>
</tr>
<tr>
<td>5</td>
<td>Tour 102</td>
<td>Sylvia V</td>
<td>48</td>
<td>F</td>
<td>1395</td>
</tr>
<tr>
<td>6</td>
<td>Tour 102</td>
<td>William H</td>
<td>22</td>
<td>M</td>
<td>1435</td>
</tr>
<tr>
<td>7</td>
<td>Tour 102</td>
<td>Lucy J</td>
<td>41</td>
<td>F</td>
<td>1435</td>
</tr>
<tr>
<td>8</td>
<td>Tour 102</td>
<td>Winnie J</td>
<td>30</td>
<td>F</td>
<td>795</td>
</tr>
</tbody>
</table>

### Commands

Now that you have seen an example of the debugger's usefulness, here is more information on the available commands and some hypothetical errors where these debugging capabilities can be helpful. Also, see Appendix 1 for a complete list of commands and syntax.

#### The BREAK Command

One of the more powerful DATA step debugger commands is the BREAK command. As shown in the example above, execution can be suspended at specific points during execution using BREAK. You may also add conditions to the BREAK command, so that execution is suspended only under certain circumstances.

Suppose you have a large program, that processes thousands of observations. You know that problems begin appearing in the output after 6,000 observations have been processed. Using a BREAK command with an AFTER clause, you can direct the debugger to suspend execution only when it is time to debug the problem. You want to pause after a particular line has executed 6,000 times:

```sas
break 102 after 6000
```

Or, you may wish to suspend execution when a certain condition or expression is true. If you notice that something goes wrong when a variable contains a certain value, you can use a WHEN clause with the BREAK command to suspend execution at that point:

```sas
break 75 when country='Italy'
```

Another component of the BREAK command is the DO group. A DO group specifies one or more debugger commands that are to be executed when DATA step execution is suspended at the breakpoint. Suppose, in our previous example, you decide that you want to display the current values for IMPORT and EXPORT every time execution is suspended (when COUNTRY='Italy'). Use a DO group:

```sas
break 75 when country='Italy' do;
```

A DO group can span more than one line, assuming you want to do more things at your breakpoint:

```sas
break 75 when country='Italy' do;
```

A DO group can also contain IF-THEN-ELSE statement logic, as in the following:

```sas
break 75 when country='Italy' do;
```

#### The EXAMINE Command

The most commonly used command, EXAMINE, displays the current value of DATA step variables. EXAMINE also allows you to output a variable using a specified SAS or user-defined format. Suppose that you have the following program:

```sas
1 data s;
2 retain daypast 0;
3 input pastday date7;
4 if (export > import) then calculate (export-import) /pop; end;
```

```sas
6 cards;
7 '03jan95'
8 '03jan95'
9 '04jan95'
10 ;
11 proc print;
12 format pastday date7.;
13 run;
```

This program produces the following output:
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 the DATE7. format to check against the input data:

```
break 4; go; examine pastday date7.;
step; step; examine pastday date7.;
```

You see that everything is input correctly:

```
Break at line 4 column 1
PASTDAY = 02Jan95
Stopped at line 5 column 6
PASTDAY = 02Jan95
```

But as you step past the IF statement (lines 4 and 5), you see that the variable PASTDAY changes. You realize that you have mixed up your variable names, as 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=PASTDAY+1 has been executed. This is something that you did not expect, and hopefully you will see your error.

The WATCH command specified above suspends the program at the top of each DATA step iteration because all non-retained variables (PASTDAY) are set to missing at this point. 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 allows the DATA step to execute to completion because DAYPAST is specified in the RETAIN statement and is therefore not set to missing.

### The DESCRIBE Command

You can display a variable's attributes with the DESCRIBE command. Suppose that you have the following DATA step:

```
data temp;
  set screen.in0001;
  input x;
  card;
  1 /
```

You get the following message in the log:

```
NOTE: Invalid date for x in line 5 1=1.
NOTE: '-------'='-----------------'=-----------------
   _ERROR_x1 _N=x1
```

On the surface, this does not make sense. But you realize that the variable X could be defined in the data set SCREEN.IN0001. So, you use the DESCRIBE command to see the attributes of the variable X, as shown:

```
describe x;
```

You see that variable X has an associated informat, which causes the invalid data message:

```
Name = X. Type = NEW. Length = 8
Informat = DATE7.
```

In order to input the variable X as a numeric, you must override this associated informat. This can either be done temporarily with an INPUT statement,

```
set screen.in0001;
input x 5.;
```

or, permanently with an INFORMAT statement

```
set screen.in0001;
informat x; input x;
```

### The CALCULATE and SET Commands

Both the CALCULATE and SET commands handle expressions. CALCULATE displays the computed result of the given expression. SET assigns the computed result of the given expression to a DATA step variable. Both may be useful in your debugging sessions. Expressions may be simple or complex, but they may not contain function calls. For example,

```
calculate x * y;
calculate y = (x * y) / (x + y) - (x * 1.1)
```

Both commands also accept SAS DATE, TIME, and 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:

```
calculate '10mar96'd - '02apr95'd
```

You will see that it is 343 days.

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

```
set x='01jan95'd + 70;
```

### The STEP, GO, and JUMP Commands

You control the execution of the DATA step using STEP, GO, and JUMP. Execution can be started or suspended, and the flow of control can be modified.

When execution is suspended at a statement, the STEP command executes the current statement. Execution is then suspended at the following statement. By default, STEP executes one statement, but you can specify the number of statements to be executed. The following two command sequences achieve the same result:

```
step; step
```

The GO command starts or resumes execution of the suspended DATA step. By default, GO causes execution to continue until either a previously set breakpoint or watchpoint is encountered or the DATA step completes. You can, however, specify with GO a target location, where execution is to be suspended. For example, suppose you are debugging a large program. As you step through a problematic section of code, you encounter a DO loop:
The LIST Command

As seen previously in an example, the LIST command allows you to ascertain the status of infile or file processing. This can be a big help in an area that frustrates many people -- finding bugs in column- and line-directed input and output. This type of I/O occurs when a column-pointer modifier (@ or +) or a line-pointer modifier (@) is used in an INPUT or PUT statement. As previously mentioned, the current way to debug these problems is to specify options in the FILE or INFILE statements and to dump their values to the DEBUGGER LOG window before each statement is executed. Specify TRACE ON to enable this tracing facility and TRACE OFF to disable it.

The LIST Command also displays information concerning any SAS data sets being processed in the DATA step program. For example,

```sas
data .... / debug;
.... 100 lines of code ....
104 do x = 1 to n;
105 ....
106 ....
107 end;
108 by ....
109 .... 100 lines of code ....
211 run;
```

You have stepped to line 104 (DO statement) and don't see the need to step through every iteration of the loop. You can set a breakpoint at line 108, enter the GO command, and then delete the breakpoint when execution is suspended:

```
break 108; go delete break 108
```

This works fine, but requires many keystrokes. A shorter, more compact method would be to use the target specification with the GO command:

```
go 108
```

This essentially establishes an implicit breakpoint at line 108, thereby suspending execution when line 108 is reached. If execution is interrupted for any other reason (another breakpoint or a watchpoint) before this target is reached, this implicit breakpoint is removed.

You can alter the flow of execution in the DATA step program with the JUMP command. When execution is suspended at a particular line, specify the next line to be executed with JUMP. This changes the current line designation and moves execution to that point in the program without executing any intervening statements. JUMP does not resume execution; it only changes the point at which execution resumes when you enter GO or STEP.

The TRACE Command

You can see the flow of execution (without suspending execution) of your DATA step program with the TRACE command. TRACE causes a message to be output to the DEBUGGER LOG window before each statement is executed. Specify TRACE ON to enable this tracing facility and TRACE OFF to disable it.

The TRACE Command

You can see the flow of execution (without suspending execution) of your DATA step program with the TRACE command. TRACE causes a message to be output to the DEBUGGER LOG window before each statement is executed. Specify TRACE ON to enable this tracing facility and TRACE OFF to disable it.

The LIST Command

As seen previously in an example, the LIST command allows you to ascertain the status of infile or file processing. This can be a big help in an area that frustrates many people -- finding bugs in column- and line-directed input and output. This type of I/O occurs when a column-pointer modifier (@ or +) or a line-pointer modifier (@) is used in an INPUT or PUT statement. As previously mentioned, the current way to debug these problems is to specify options in the FILE or INFILE statements and to dump their values at strategic points in the program.

Much of this information can be obtained with the LIST INFILES or LIST FILES command from the debugger:

```
list infiles
```

This displays the name, the current column number, the current line number (relative to N=), the current line length, and the contents of the current line:

```
Logical Name = MYFILE
Column = 5, Line = 1, Length = 16
Current buffer = 1 2 3 4 5 6 7 8
```

If an INPUT statement has not been executed yet, the current buffer displays as all blanks. LIST will not read the current buffer.

The LIST Command also displays information concerning any SAS data sets being processed in the DATA step program. For example,

```
list dataset:
name = WORK.SAS, number of observations = 0
```

You can also see the active breakpoints and watchpoints in your debugging session by using the LIST BREAK or LIST WATCH commands.

THE DEBUGGER IN RELATION TO THE REST OF THE SAS SYSTEM

Windows

All SAS window functions and commands are available from 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

```
ext '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 on the debugger's command line. The debugger keeps a circular queue of the last 20 commands so that you can travel back through your old commands, change, and submit them.

A new window command, SWAP, switches control between the two debugger windows. The CATALOG, VAR, and LIBNAME window are only available during a display manager session of the SAS System. If you get lost in the maze of windows, you can get back to the debugger windows. The CATALOG, VAR, and LIBNAME window are only available during a display manager session of the SAS System.

Keys

The debugger also accepts input from a set of function keys that you define for the debugger. These keys are accessed via the KEYS window command, entered from the command line of either the DEBUGGER LOG or DEBUGGER SOURCE windows. This means that you can define a function key to perform a debugger command or series of commands. The format of the key specification is the string "DBD" followed by the debugger command. For example, if you want to save yourself some typing, you can define a key (through the KEYS window) to examine all the variables in the program:

```
exam all
```

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

```
ex 'exam _all_/ describe _all_'
or
des _all_;
```

164
The Macro Facility

Because debugger commands are input through the same source manager as the rest of the SAS System, you can use the macro facility to write debugger commands that are customized for certain DATA step programs, or for your taste. These macro commands can be aliases for longer, more complex commands or short-hand notation for a series of commands. For example, you can customize the BREAK and GO commands as shown:

\begin{verbatim}
%macro mbrk(line); break line do; examine all; end; %end;
%macro mgo(line); go line; examine all; %end;
\end{verbatim}

The macro MBRK sets a breakpoint on a specified line and instructs the debugger to examine all variables when this breakpoint is reached. The macro MGO directs the debugger to go to a specific line and to examine all variables when execution is suspended. These macros are specified from the debugger command line as

\begin{verbatim}
%mbrk(S2)
\end{verbatim}

and

\begin{verbatim}
%mgo(S2)
\end{verbatim}

You can expand MGO to trace execution upon command by using some additional macro logic such as

\begin{verbatim}
%macro mgo(line, tr); go line; trace on; examine all; go line; %end;
\end{verbatim}

MGO now traces statement execution if a second parameter of 't' is specified. MGO can be specified from the debugger command line by either of the following:

\begin{verbatim}
%mgo(S2) /* Will not trace execution... */
%mgo(S2,t) /* Will trace execution... */
\end{verbatim}

Macros are 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, macros defined outside of a debugger invocation are available to the debugger. These macros can be included in an autocall library.

These macros can also be defined as command-style macros. Command-style macros look like real commands and can be specified from the debugger command line. (The CMDMAC option needs to be in effect.) The previous examples would be defined as

\begin{verbatim}
%macro mbrk(line); break line do; examine all; end;
%macro mgo(line, tr) go line; trace on; examine all; go line; %end;
\end{verbatim}

and specified on the debugger command line as

\begin{verbatim}
mbrk 52
\end{verbatim}

and

\begin{verbatim}
mgo 52
\end{verbatim}

You can use the %INCLUDE statement to include a series of commands that do not resume program execution. If there are commands that resume and stop program execution, some of the commands are lost. However, %INCLUDE is 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 (PMENUs), one for the DEBUGGER SOURCE window and one for the DEBUGGER LOG window. These pull-down menus operate identically to other pull-down menus in the SAS System. Their definition is customized to the debugger.

CONCLUSION

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

ACKNOWLEDGMENTS

The authors greatly appreciate the contributions by Jeff Polzin for the original work on the debugger, Nancy Agnew for testing the debugger and providing examples, and Susan O’Connor and Judy Heffner for assistance in reviewing this paper.
APPENDIX 1: COMMAND SUMMARY

BREAK I B  Suspend program execution at given location

Break location <After count> <When expr> <DO_group>

location may be one or more of the following:
- line-number  program line of statement
- label       statement following the label
- *           current line

count is an integer, specifying the number times location is passed before execution is suspended.

eexpr is a DATA step expression, but cannot contain function calls.

DO_group takes the following form:

DO; command_1; ...; command_n> END;

command may be any of the following:
- A single debugger command
- IF expr THEN command; DO_group
- ELSE command; DO_group

CALCULATE I CALC  Evaluate expression and display result

CALCulate expression

eexpression is a DATA step expression, but cannot contain function calls.

DELETE I DEL I D  Delete breakpoint or watch status

Delete Break location

location may be one or more of the following:
- line-number  program line of statement
- label       statement following the label
- *           current line
- _ALL_       all active breakpoints

Delete Watch variable_1 < ... variable_n>
Delete Watch _ALL_

DESCRIBE I DESC  Display attributes of variable

DESCribe variable_1 < ... variable_n>
DESCribe _ALL_

ENTER  Assign one or more command to the ENTER key

ENTER <command_1 < ...; command_n>

By default, the ENTER key is assigned the command STEP.

EXAMINE I EXAM I EX I E  Display value of variable

Examine variable_1 <format_1> < ... variable_n <format_n>
Examine _ALL_ <format>

GO I G  Start or resume program execution

Go <location>

location may be one or more of the following:
- line-number  program line of statement
- label       statement following the label
- *           current line

JUMP I J  Alter flow of program execution

Jump location

location may be one or more of the following:
- line-number  program line of statement
- label       statement following the label

LIST I L  Display status

List _ALL_
List Break
List Datasets
List Files
List Infiles
List Watch

QUIT I Q  Terminate a debugging session

Quit

SET  Assign a value to a specified variable

SET variable=expression

expression is a DATA step expression, but cannot contain function calls.

STEP I ST  Execute specified number of statements in program

STEP <n>

STEP with no argument executes one statement.

TRACE I T  Display record of execution

Trace <ON I OFF>

Trace with no argument displays the current trace status.

WATCH I W  Suspend execution when value of variable changes

Watch variable_1 < ... variable_n>