An Animated Guide: The SAS® Data Step Debugger

Russ Lavery, Contractor

ABSTRACT

The Data Step Debugger (DSD) simplifies the debugging of individual Data Steps and not whole SAS programs. The SAS Debugger is not a "use everyday" tool, but is extremely useful when debugging complex data steps. Using the debugger requires some understanding of how SAS works because the DSD does not issue any error messages. When using the DSD, a programmer mentally compares what s/he sees, displayed in the Program Data Vector (PDV), with what s/he expected to see. Critical, to use of the DSD, is an understanding of the PDV and, as a side issue, the DSD is an excellent way to learn the PDV.

The goal of a programmer should be to learn to ***combine DSD commands*** and to link those combinations to a key or a macro. Linking a series of commands to a key, or a macro makes the DSD much more powerful.

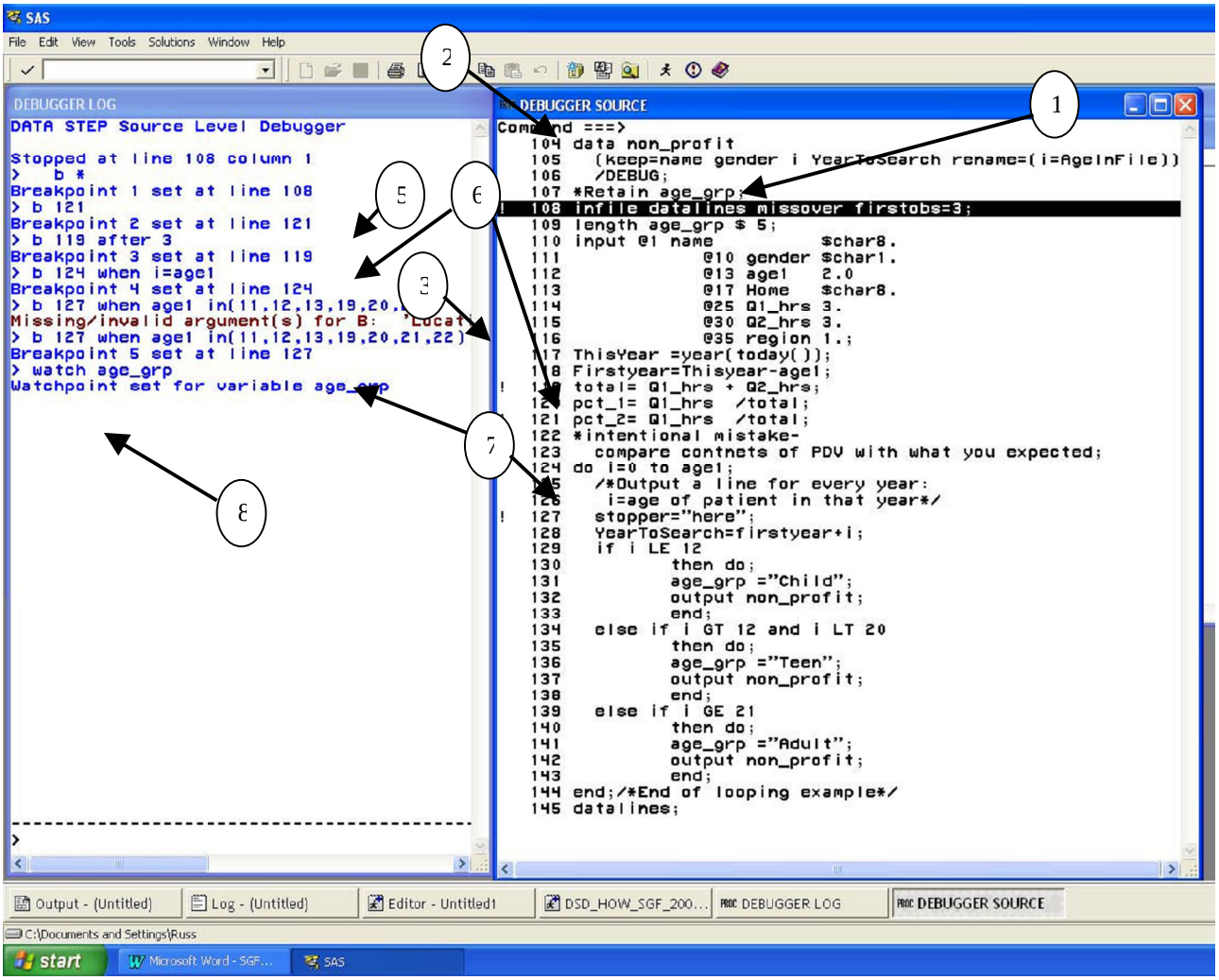

FIGURE 1

INTRODUCTION

This paper concentrates on using the DSD interactively to boost productivity. It will quickly cover DSD commands, and then concentrate on explaining the most useful **combinations of commands**, and the programming situations, where the DSD is likely to give a productivity boost. The first goal of a programmer starting to learn the DSD should be to link DSD commands to a function key or to a macro. Linking commands to a function key, or a macro, makes the DSD much more powerful.

USING THE DSD IN INTERACTIVE MODE

The debugger is most often used in interactive mode. To operate in this manner, SAS must be running with the SAS editor window active. The data step you are trying to debug *must not have syntax errors*, only logical errors. In short, syntax errors cause the DSD to not start (the usual situation) or to misbehave. For example, if you leave out a semicolon, and try to run the DSD, the DSD will not start and the DSD screens will not be displayed. Referencing non-existing variables might cause the DSD to not stop at break points.

The DSD is invoked by adding "/debug" to the data step line [circle (1) Figure 1], highlighting the data step code and then hitting <F3> (or run). Invoking the DSD will add two more windows to your computer screen, the DSD log and the DSD source. These windows are shown in Figure 1. Since the debugging work is done in these two windows, programmers typically hide the SAS log, SAS list and SAS editor windows so as to be easily able to see activity in the DSD log and DSD source [Figure 1].

A workable screen layout is shown in Figure 1. The messages shown in the DSD log are rather short and programmers often make the DSD source wider than the DSD log. Once you have found an effective layout, you should issue WSAVE on the command line [circle (2) Figure 1] in both windows (note that only the active window shows its command line) and SAS will open the DSD with that layout in future sessions.

To make the command line appear in a window, work through the following steps: Tools-Options-Preferences-Select the View tab – click on command line and OK. When the command lines appear issue the WSAVE command in each window. It should be noted that the log window must be wide enough so that commands can be typed. The Debugger log does not scroll, to the right, as commands become longer than the window can accept. The red message (circle (3) Figure 1), in the log in Figure 1, was caused be a command (the B121 … shown just above the red line) becoming too wide for the screen as it was typed. If the DSD stops accepting characters and returns a red message because the command has become too long, simply use the mouse to widen the window and re-enter the command.

Running the DSD is fairly simple. DSD commands are typed in a command line in the bottom of the log window (circle (4) Figure 1). The line about to execute is highlighted in black in the DSD source window [Figure 1]. The DSD only stops on (reverse highlights) executable lines. In the code in Figure 1, the DSD will never stop with line 107 in reverse highlight because line 107 is not executable. Because the DSD only stops on executable lines, it is sometimes useful to insert executable "nonsense lines" like:

```
Book="mark1";
```

into the SAS code so that you will have an executable line where you'd like to make the DSD to stop.

Progress messages [circles (5 & 6) Figure 1], as well as the results of DSD queries are written to the DSD log window.

BINDING/USING GROUPS OF DSD COMMANDS

There are only a handful of easy-to-learn DSD commands. However; using sequences of commands, not using individual
commands, is what makes the DSD a useful tool.

Binding a useful, and frequently used, series of commands to a key-combination, or including them in a macro, is the starting point for DSD power use. These two techniques should be a goal of everyone starting to learn the DSD. The DSD is slow if you are a bad typist and do not "bind" commands.

DSD COMMAND FUNCTIONAL GROUPS

The DSD commands are generally grouped by the functions they perform and will be grouped that way in this paper. The
commands have abbreviations to reduce typing and options to increase their power/flexibility. The most frequently used commands have been underlined below. Many commands have abbreviations that can be discovered in the SAS help pages.

Generally, binding commands to a key, or calling a macro of DSD commands into the DSD, minimizes the need to use the abbreviations.

Execution commands move the (reverse highlighted) "active line" of code- by executing steps or skipping steps.
The four commands are: **Step, Go, Jump, Quit**

Suspension commands tell SAS where to stop execution of code. Execution must be stopped to display the PDV.
The four commands are: **Break, Watch, Delete, Trace**

Display commands can show the PDV, and input buffer, in the DSD LOG.
The three commands are: **Examine, List, Describe**

Window commands toggle the active window between editor a/log. Only the active window has a scroll bar, and can be scrolled.
The two commands are: **Swap, mouse-click**

Other commands do miscellaneous things
The three commands are: **Calculate, Set, Help**

SPEED TRICKS make the DSD convenient to use
The tricks are: **Binding Commands to Keys, <F4>, Do-Loop Macro, Calling a macro of DSD commands**

DSD EXECUTION COMMANDS

Execution commands cause lines of code to execute or jump over lines of code without executing.

Step causes the DSD to execute lines. Pressing the return key is abbreviated <ret> in this paper.
The commands *Step<ret>* (or *st<ret>* or *step 1<ret>* or *<ret>*) all will cause the editor to execute one command and stop. *Step n* will cause the editor to execute n commands and stop. Usually, if you want to execute one, or a few lines, you will hit the return key, rather than typing *step*. If you want to execute a number of lines, the *Go linenumber<ret>* (Go to line number n, executing lines as you go) command is easier than *step n*. Line numbers are shown in the Debugger Editor window. In Figure 1, line numbers go from 104 to 145. Line numbers are counted as the number of lines executed since the SAS session started. If you re-run a DSD session on the same date step, you will see different line numbers in the editor.

Go has several options and does several things.
A Go <ret> will execute lines until the DSD encounters a condition for which you have coded a stop. If you have set a break point at a line, or instructed the DSD to watch a variable for value changes, issuing *Go<ret>* will cause the DSD to execute lines until the break line is reached or the watched variable changes value. If you have not set any stop/break points, GO will cause processing of all observations through all the lines in the data step, effectively running the data step “to a normal conclusion”.

Two useful Go options are:

***Go** linenumber <ret>*. This command will cause the DSD to execute all lines between the current line and the specified line number. Line numbers are shown in the left-hand side of the Data Step Source window. This can be more convenient than repeatedly typing Step<ret>.

***Go** label<ret>.* You can put a SAS label in your code and issue the command *Go label<ret>*.

Jump is a command that lets you skip lines of code. If your DSD window showed the screens in Figure 1, and you issued jump 124<ret>, you would not execute any of the lines of code between lines 108 and 124. The jump command is useful if you think one block of code is causing a problem, and you do not want the block of code to run but also do not want to exit the debugger to comment out the suspicious block of code.

QUIT ends the debugger session. It closes the debugger log and debugger editor.

DSD SUSPENSION/BREAK COMMANDS

Suspension/break commands simply specify where/when to stop execution and do not display any information. You can tell the DSD to stop (break) at a line number [circle (5) Figure 1], or stop on a line when a logical condition is true [circles (6 & 7) Figure 1]. As Figure 1 shows, you can set several break points, each having different logical conditions. You can also ask the DSD to stop immediately, if a variable changes value [circle (8) Figure 1]. When you issue a "suspension command" you will get a message in the DSD log and see an ! in the DSD source (see lines 119, 121,124, 127 in the source window in Figure 1) indicating the line has been tagged as a "break line". You usually break, or suspend execution, so that you can then issue "display commands" to check values of variables as execution waits on that line. Printing commands are separate from break commands.

Break tells the DSD to "stop on a line" and has useful options.
Most often control passes down from the top of the data step code to the run statement and the cycle repeats. This process happens for each observation, with little internal "cycles" for Do loops. Sometimes you will want to stop every time the DSD is about to execute a line. In that case, if the DSD is highlighting the line you wish to make into a "break every time this line is about to execute" line you can simply type b *<ret> [see first line in the debugger log, Figure 1]. This makes the line a "break every time you are about to execute" line. Generally, If you want to make a line that is not reverse highlighted a "break every time you are about to execute " line, you issue the command break linenumber<ret>. [circle (5) Figure 1]. When a break has been set successfully, the DSD writes a message to the DSD log and puts an exclamation point on that line in the DSD Source Window [Figure 1].

When your programming problem is to debug a do loop or to try to find "odd observations" (a missing value or that "one darn observation" that is giving you trouble), "break at this line when a condition is true" is a useful option. Often you want to see the last "pass through a loop" or you would like to see the processing of observations through the nesting of "IF-ELSE-IF" code. Detailed debugging (good programming practice) requires that we check the execution of every IF statement. Breaking when a variable reaches certain values is useful for debugging loops and IFs. It is also useful for examining what happens when certain observations pass through the data step [see the break commands in the DSD log in Figure 1].

You might want to check that each level in an IF statement executes correctly (see the IF series in Figure 1 for background). We might like to check the proper routing of observations that are close to the critical values of the IF statements (when age1 equals 11, 12,13,19, 20, 21, 22). We know that observations with age1 equal eleven should be classified as a child. Observations with ages twelve through twenty should go to teen. Observations with age equal Twenty-one should be classified as adult.

A useful breakpoint for the example above might be *Break (at) 124 when age in (11,12,13,19,20,21,22)<ret>* followed by a *GO*. This would process observations, possibly very many observations, and stop processing on line *124* when an observation with an age1 close to the "IF breakpoints" is about to be processed. After breaking execution on an interesting observation you could then proceed to "step" that observation through the data step – one line at a time.

The debugger log also shows the issuance of a "break (at) line-number after n" command [circle (6) Figure 1]. This command is useful in debugging loops. You can tell the DSD to stop processing after a line has been passed n (a certain number of) times. This means you can instruct the DSD to cycle through a do loop and then stop when it is just about to leave the loop. You can then step through the last cycle and see what happens as you leave the loop. [see Figure 1,especially "break 124 when i=age1"].

You have options in debugging loops. Stopping execution at a particular stage of a loop can be done with a break after or with a break when.

Watch tells the DSD to stop when a variable changes its value, *regardless of what line is executing*. The command is *watch var.-name<ret>* and does not take a line number argument [circle (6) Figure 1],. The watch command is very useful when you are trying to debug the setting of a flag, especially when the flag can be set in several places in your code.

Delete is maintenance command that "clears" break and watch points. Sometimes, you can use the DSD to examine two, or three, problems. If you resolve the first problem you will want to clear the break points and the watch variables associated with that first problem. You might then set new breaks/watch variables to investigate the second problem.

Delete will clear both breakpoints and watchpoints. The command to clear a break is `Delete B linenumber<ret>.`

The command to clear a watchpoint is `Delete W varname<ret>.` You can save time with `D B _all_<ret>` or `D W _all_<ret>`.

Trace can be toggled on and off. When trace is on, messages are written to the log that show the lines that were executed. This can be useful if you issue a go and want to be able to determine what path execution took through an IF-ELSE section. Its use is limited to tracking execution through line numbers and does not provide any information about variables changing values

DSD DISPLAY COMMANDS

Display commands show values, or attributes, of variables. The reason you set break points on particular lines of the data step is so that you can, later, issue display commands at particular points in your program. Once you have halted execution with a break you will issue the display commands, explained below, to show the PDV and input buffer.

EXAMINE
Examine is the most useful display command and displays values from the PDV in the DSD log.
The syntax is `examine variable(s) format<ret>` . `Examine <varnames> format` will show, in the DSD log, current PDV values for the variables listed.

The format is not a required option but, if used, applies a format and can be useful in making output more readable.

To save keystrokes you can issue the command as `E _all_<ret>`. This command will examine (show in the DSD log) all variables in the PDV. While this is an "easy to type" command it usually provides more output than is desired, or convenient. Repeatedly typing this command with a list of many variables (e.g. `examine var1 var12 var13 var24 var5`) gives you just what you want to see, but can be tedious. To save typing, the examine command is usually bound to a function key (explained below) or the programmer can use the <F4> key.

Examine does not accept the following SAS abbreviations:
`Examine _numeric_` ***or*** `Examine _character_` ***or*** `e var_nm1-var_nm3` ***or*** `e varnm1--varnmchar`

LIST
List is a DSD display tool and writes messages to the DSD log to describe your DSD session. `List d<ret>` provides information about the SAS data set you are creating. `List I<ret>` will list the current infile (the source of data) and display the input buffer.

Imagine, you are reading a text file and have made a mistake in coding the input statement (maybe typing the wrong column number in the input statement). You can use `list I<ret>` to see the input buffer and then use `examine varname<ret>` to see if what you read into the PDV is what SAS has in the input buffer. The input buffer shows what is in the data file and the PDV shows what you "got" into your variables.

`List b<ret>` writes a message that tells you what lines have breakpoints set, but it produces very sparse messages. Remember that you can associate conditions with the breakpoints when you set the breakpoints. You might set breakpoints with associated conditions like when var=" and "after n". However, *`list b<ret>`* will not report on the conditions that are associated with the breakpoints. It just reports the line number, not the associated condition.

`List W<ret>` will write a message to the DSD log that indicates the variables you are watching and tells the current values of the variables.

`List _all_<ret>` will report on all the list options.

DESCRIBE writes information about variables to the DSD log. The format is `Describe firstvar secondvar thirdvar<ret>`. Describe causes SAS to write the variable name, type and length to the DSD log. The output is similar to that of a PROC CONTENTS, but just on the variables specified.

DSD WINDOW COMMANDS

The window command toggles the active window between editor and log (using a mouse is usually faster). Only the active window has a scroll bar and a command line. If your data step does not fit on one screen, you might want to scroll up/down to see code that is not currently displayed. You will issue the `WSAVE` command on the command line of every window whose shape you want SAS to remember. You can change the active window with *`Swap<ret>`* or by left

mouse clicking on the window you wish to make active.

DSD SPEED TRICKS

Using speed tricks makes the DSD a powerful tool and a pleasure to use. The most important speed trick is binding a set of commands to a key (or pair of keys) on your keyboard.

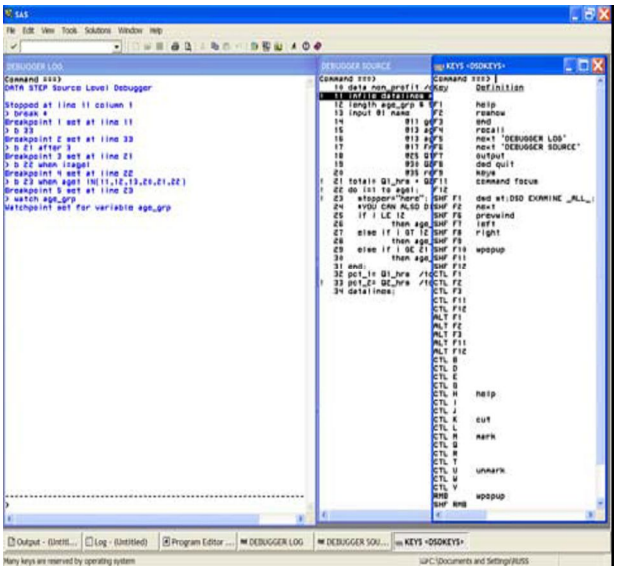

FIGURE 2

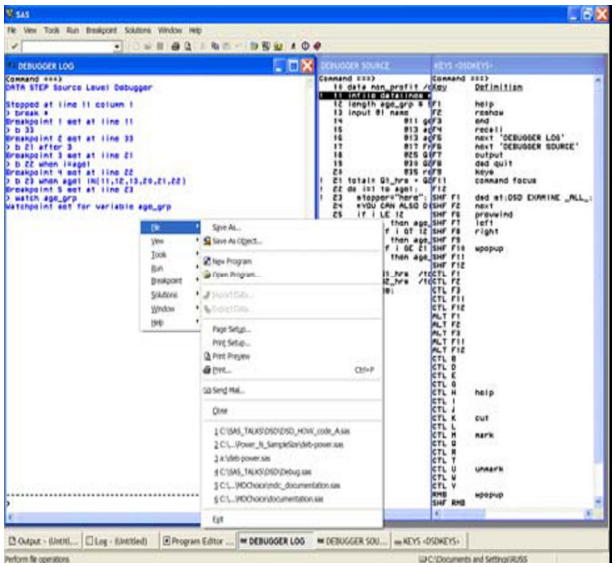

FIGURE 3

If the DSD is active and you press <F9> the DSD keys window will be displayed [see Figure 2]. Figure 2 shows that two types of commands have been linked to keys: command line commands (like help, reshow, end, recall, etc) and DSD commands (see `shift <F1>` and `shift<f11>`). Note that <F9> has already been linked to the keys command. That is why pressing <F9> displays the DSD keys window.

The keys window, in Figure 2, shows what key bindings that are in effect when the author's DSD is running and your key bindings may differ. These key bindings differ from bindings in effect when regular SAS is running. Pressing <F9>, when in the regular SAS editor, will also display a keys window. However, the key bindings would be different from the DSD key bindings shown above.

Please note that binding commands to keys is a critical productivity technique for using the DSD. Looking in Figure 2, `shift <F1>` shows a variation of one of the most useful DSD command strings. Pressing `Shift<F1>` will execute one line and then display the full PDV (`Shift <F1>` is linked to: `DSD st; DSD examine _all_;`). If there were repeated need to examine four or five variables you could save time by binding the examine command (e.g. `DSD Step; DSD examine var1 varB var 23 varAA;`) to a key and you would never need to type the command again.

Key binding is easy to set up. Open the key window and use the mouse to position the cursor on a blank line [circle (1) Figure 2] or a line that holds a command you wish to modify. Type in the DSD commands, remembering two rules. 1) DSD commands must begin with the string DSD and 2) DSD commands must end with a semicolon. Several commands can be linked to one keystroke.

(You can see examples of these rules in Figure 2. – note `shift-<F1>` and `shift-<F11>`) To finish the key binding process, close the keys window to have a better view of events. Finally, make the DSD log window active. You can then press the key(s) to which you have bound the commands and watch the DSD execute them. *NOTE: Your key bindings will only work when the DSD Log window is active.*

By default, the <F4> key is usually bound to the recall command. It has the same function as DOSKEY does in DOS. Pressing <F4> will recall the last twenty DSD commands back to the DSD log command line. Once the proper command is on the DSD log command line, it can be executed by hitting `<ret>`. This can be a small timesaver, if you do not want to bind a command string to a key.

Figure 3 (above) shows how the DSD can be run from pull down menus. Right clicking on the DSD editor, or DSD log, will produce a menu of DSD commands. DSD commands (examine, list, break, delete etc) can be executed from this menu. A useful feature of the menu is the ability to save the DSD Log, and/or DSD Editor, to a file. From the menu shown in Figure 3, select File - Save As. This "easy save" is the most useful command in the menu.

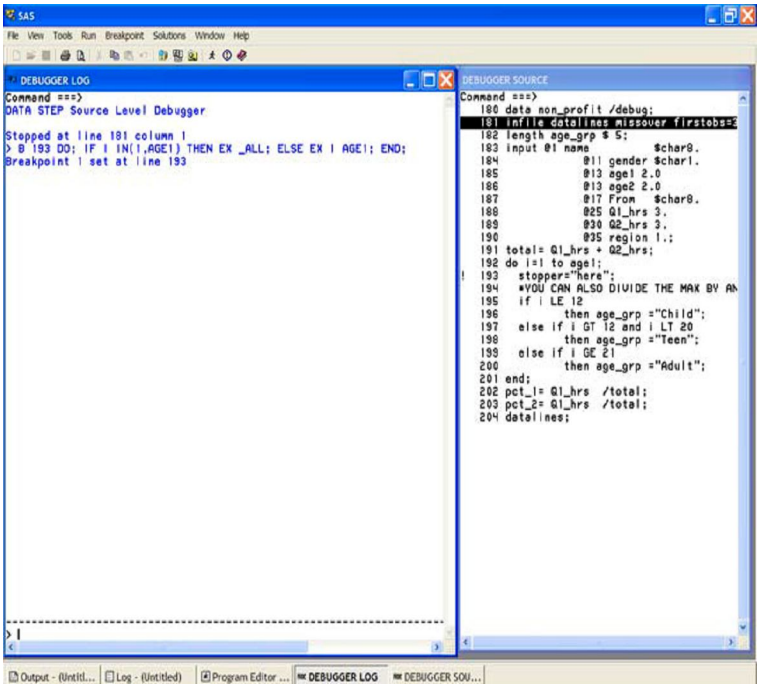

FIGURE 4

```
*************************************************************************;
**  SPEED TRICKS                                                       **;
*************************************************************************;
# A way of avoiding the poor editing capability of the Key binding process is ;
# to write a macro -OF DSD COMMANDS - outside the DSD and call the macro from within the DSD;
#these macros are stored in the macro catalog and are available at anytime during the session;
# common macros can be called at startup * please execute the following code;


        /* NO COMMENTS INSIDE MACRO DEFINITION! THE NAME IS G FOR "GO MACRO";
          this macro issues the command - go to a line number and examine all variables*/
%macro G(lineno);
        go &lineno;
           examine _all_;
%mend G;

        /* NO COMMENTS INSIDE MACRO DEFINITION!  THE NAME IS lp FOR "LooP Macro"
          this macro sets a break at the current line - line must be first executable line in loop
          LINENO IS THE LINE ON WHICH WE WANT THE BREAK,
        UPVAR IS THE VARIABLE THAT CONTAINS THE UPPER LIMIT OF THE LOOP*/
%macro lp(lineno,Upvar);
             B &lineno do;
                  if (i=1 or i=&upvar ) then Ex _all_;
                  else  ex I &upvar;
               end;
%mend lp;

*NOW RUN THE CODE BELOW AND CALL THE MACROS;
options nocenter;
data non_profit /DEBUG;
infile datalines missover firstobs=3;
length age_grp $ 5;
input @1 name          $char8.
            @11 gender $char1.
            @13 age1 2.0
            @13 age2 2.0
            @17 From   $char8.
            @25 Q1_hrs 3.
            @30 Q2_hrs 3.
            @35 region 1.;
total= Q1_hrs + Q2_hrs;

do i=1 to age1;
  stoppers="here";
  *YOU CAN ALSO DIVIDE THE MAX BY AN INTEGER AND USE IN THE BREAK;
  if i LE 12
          then age_grp ="Child";
  else if i GT 12 and i LT 20
          then age_grp ="Teen";
  else if i GE 21
          then age_grp ="Adult";
end;
```

FIGURE 5

Multiple DSD commands can be "set to execute" after a break by enclosing them in a Do-End as can be seen in FIGURE 4.

This requires a lot of typing on the command line (often resulting in typos for me). If's are supported inside the Do-End.and this feature has a use in debugging loops.

The complex command in Figure 4 can be bound to a key but is easier to type if put in a macro.

Often, there is an interest to see values both at the start of the loop and at the end of the loop. The code in FIGURE 4 (combined with issuing the Go command) will EXamine the full PDV during the first, and last pass, through the loop.

On other passes, it will EXamine just the loop counter and age1. A user, at any time, could issue additional examine commands to EXamine other variables in the PDV.

The SAS editor is better than the DSD command line editor.

Saving a series of DSD commands, in the SAS macro catalog and recalling the commands into the DSD is another speed trick.

This trick allows you to use the SAS editor (a editor than the debugger 'editor") when creating a complex bit of code.

The tow macros in FIGURE 5 are created and run outside the DSD in order to compile the macro in the normal SAS Macro Catalog.

The DSD is then started.

To call the macro named G inside the DSD, and have the DSD stop on line 134, the user would type `%G(134)` on the DSD log command line. [circle (4) Figure 1]

To execute the Lp macro from inside the DSD, type `%lp(127,age1)` `<ret>` and then a series of `Go` `<ret>` commands (Or one `Go` `<ret>` and a series of `<F4>` `<ret>`) commands

RUNNING THE DSD ON CODE GENERATED BY A MACRO

Macros are often difficult to debug and often do unintended things. A programmer is left to wonder if the macro evaluated incorrectly and gave "poor SAS code" or if the macro evaluated correctly and the underlying logic of the source code was wrong (and is therefore a candidate for analysis using the DSD). In this situation, the programmer needs to have clean code (the result of the macro execution) to be "put into" the DSD.

In the past, programmers occasionally turned on MLOGIC, ran the code, copied the log into an editor and then manually removed SAS comments and notes from the code to create code to be then run through the DSD. Clean code, generated by macro execution, can now be captured with the following easy method that uses MPrint MFile and a Filename statement.

IMAGINE A MACRO DEFINED AS BELOW. THIS MACRO HAS NO BUSINESS REASON, IT JUST ILLUSTRATES A FEW POINTS.

```
%MACRO series(DSET=);
 data new_&Dset;
   total= %do i=6 %to 12;
   &i %str(+) %end; 0 ; run;
  proc print data=new_&Dset; run;
  proc means data=new_&Dset; run;
%MEND series;
```

The file series_txt.txt contains the following text.

```
data new_PLACE;
total= 6 + 7 + 8 + 9 + 10 + 11 + 12 + 0 ;
run;
proc print data=new_PLACE;
run;
proc
```

The programmer could issue the commands below.

```
options mprint mfile;
filename mprint "c:\temp\series_txt.txt";
%series(DSET=PLACE); *this runs the macro;
options nomfile;
run;
```

The `%series` statement runs the macro and `options nomfile` closes the file being creted. Note that if the macro is run again, the results of the new run will be APPENDED to the old run. There is no option that will cause SAS to overwrite the old file. Appending to the collecting file (series_txt.txt) is the only option. The file series_txt.txt shows how the macro evaluated the %do loop and all the other statements inside the macro. A programmer desiring to step through this code with the debugger need only cut and paste the data into SAS, change "`new_PLACE;`" to "`new_PLACE /DEBUG;`" and re-run.

ENDING THE DSD SESSION

The debugger session is ended by typing quit<ret> in the DSD command line of the DSD log.

CONCLUSIONS

The DSD is a powerful tool for debugging the Data Step parts of programs. It should be in the toolkit of every SAS programmer. The basic output is a listing of values in the PDV. Effective use of the DSD requires understanding the PDV.

REFERENCES

Heffner and Schaffer. "The Data Step Debugger ", Proceedings of the Twentieth Annual SAS Users Group International Conference", 20

Riba, S. Davids, "How to use The Data Step Debugger " Proceedings of the Twenty-Third Annual SAS Users Group International Conference", 25, paper 52

Schaffer and Kalt, "The Data Step Debugger: Bug hunting made Easy ", Proceedings of the Sixteenth Annual SAS Users Group International Conference", 16

CONTACT INFORMATION

Your comments and questions are valued and encouraged. Contact the author at:

Russ Lavery Bryn Mawr, PA Email: Russ.Lavery@verizon.net Web page for all papers: Russ lavery.com