

HW-08

"An Animated Guide©: The Data Step Debugger"

Russell Lavery, Contractor for Numeric, LLC. Chadds Ford, PA

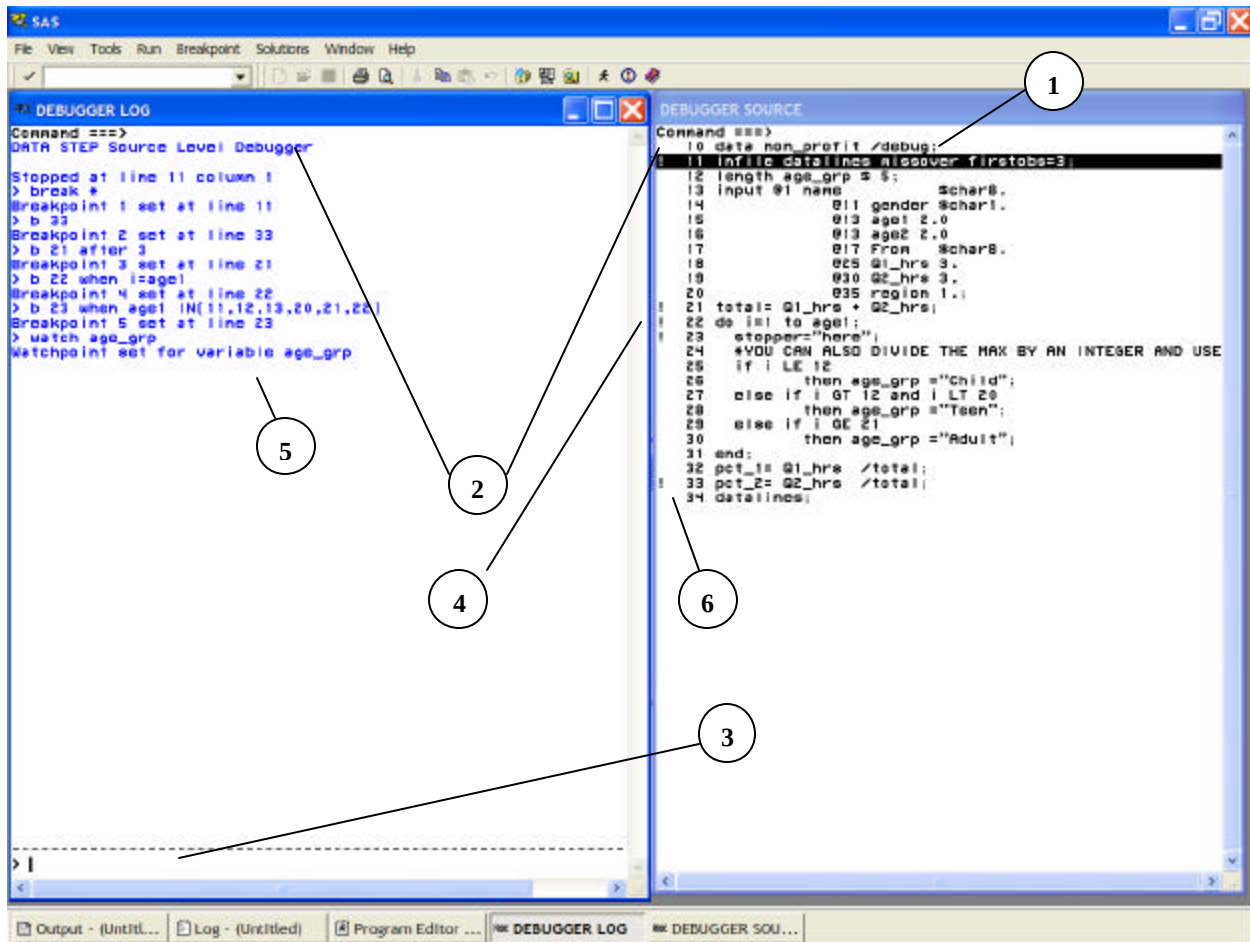


Figure 1

ABSTRACT

Use of the Data Step Debugger (DSD) simplifies debugging code and, while the data step debugger (DSD) was introduced in SAS® version 6.11, it is not widely known. It is a powerful tool for finding logical errors in the data step. It gives you control over execution of "data step" code and allows you to watch your program execute- line by line. The DSD allows you to watch if-else statements execute to check if the code is correct *while the code is running*, rather than checking for correctness by writing Proc Print statements after the Data Step has finished. The DSD allows you to not only watch code execute, but will allow you to MANUALLY CHANGE the values in the program data vector as your job is running. The DSD does not issue any error messages as you use it. The programmer mentally compares what s/he sees, 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 the DSD is an excellent way to learn the PDV.

The goal of a programmer learning the DSD should be to learn the DSD commands and how to link them to a key or to call a macro of commands. Linking a series of commands to a key, or calling a macro makes the DSD much more powerful.

INTRODUCTION

This paper concentrates on using the DSD interactively and on how the DSD can be used to boost productivity. It will quickly cover commands, and then concentrate on explaining the combinations of commands, and the programming situations, where the DSD is likely to give a productivity boost. The goal of a programmer learning the DSD should be to learn the DSD commands and

how to link them to a key. Linking a series of commands to a key 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. If you leave out a semicolon and try to run the DSD, DSD screens will not be displayed. Referencing non-existing variables might cause SAS to display the DSD windows but not stop at break points. In short, syntax errors cause the DSD to misbehave.

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 the screen, the DSD source and the DSD log. Since the debugging work is done in these two windows, programmers typically hide the SAS log, list and editor windows so as to be easily able to see activity in the DSD log and 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 makes the DSD source wider. 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, 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.

Running the DSD is fairly simple. DSD commands are typed in a command line in the bottom of the log window (circle (3) Figure 1). The line *about* to execute is highlighted in black in the DSD source window [ciFigure 1]. The DSD only stops on (reverse highlights) executable lines. In the code in Figure 1, the DSD will never stop with line 12 in reverse highlight because line 12 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 code so that you will have an executable line where you'd like the DSD to stop. Progress messages [circle (5) Figure 1], as well as the results of DSD queries are written to the DSD log window.

DSD COMMANDS

There are only a handful of easy-to-learn DSD commands. However; using sequences of commands, not 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 painfully slow if you are a bad typist and have to type in every command.

DSD COMMAND GROUPS

The DSD commands are generally grouped by the functions they perform and are grouped that way below. 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 into the DSD, minimizes the need to use the abbreviations.

Execution commands move the "active line" (the reverse highlighted line) of the SAS code- by executing steps or skipping steps.

Step Go Jump Quit

Suspension/Break commands specify where to stop execution of your SAS code

Break Watch Delete Trace

Display commands show the PDV and input buffer in the DSD LOG.

Examine List Describe

Window command toggle the active window between editor/log

Swap mouse-click

Other commands do miscellaneous things

Calculate Set Help

SPEED TRICKS make the DSD convenient to use

Binding Commands to Keys <F4> **Do-Loop Macro** **Calling a macro**

DSD EXECUTION COMMANDS

Execution commands cause lines of code to execute or jump over lines without executing. Little is printed in the DSD log.

Step causes the DSD to execute lines. Return is abbreviated <ret> in this paper. *Step<ret>* (or *st<ret>* or *step 1<ret>* or *<ret>*) will all cause the editor to execute one command and stop. *Step n* will cause the editor to execute n commands. 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>* command can be easier than *step n*, when n is a large number. The line numbers are shown in the Debugger Editor window. In figure 1 they go from 10 to 34.

Go has several options and does several things. A *Go <ret>* will execute lines until the DSD reaches a stop point. The stop point might be a line on which you have set as a stop point [circle (6) Figure 1] or the "end of the data step". If you have not set any stop/break points, go will cause processing of all observations through all the lines in the data step. It will run the data step "to a normal conclusion". 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.

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. You will likely use *Go linenumber <ret>* rather than *step many-n<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 31<ret>, you would not execute any of the lines of code between lines 11 and 31. The jump command is useful if you think one block of code is causing a problem, and you do not want to have it run but do not want to take the effort to comment it out.

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

DSD SUSPENSION/BREAK COMMANDS

Suspension/break commands specify where/when to stop execution. You can tell the DSD to stop (break) at a line number, or stop on a line when a logical condition is true. 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. When you issue a "suspension command you will get a message in the DSD log and see an ! in the DSD editor (see line 21 Figure 1) on the "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.

Break tells the DSD to "stop on a line" and has useful options.

Most often control passes from the top of the data step code to the run; for each observation with little cycles for looping. Sometimes you will want to stop every time the DSD is about to execute the line. If the DSD is highlighting the line you wish to make a "break every time line" you can simply type *break *<ret>* [see Debugger log, Figure 1] and the line will become a "break every time you are about to execute" line. Generally, If you want to make a line a "break every time you are about to execute " line, you issue the command *break linenumber<ret>*. 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 [circle (6) figure 1].

When your programming problem is to try to find "odd observations" (a missing value or the one darn observation that is giving you trouble), or you are trying to debug a do loop, "break at this line when" 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 requires that we check the execution of every if statement. You can have DSD stop execution of code at the break line only when the condition is true. The "Break at this line when" is useful for debugging loops and 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 (11, 12, 20, 21). We see that observations with age 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 23 when age in (11,12,13,19,20,21)<ret>* followed by a *GO*. This would process observations, possibly very many observations, and stop processing on line 23 when an observation with an age 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, in Figure 1, also shows the issuance of a "break line-number after n" command. This is also useful in debugging loops. You can tell the DSD to stop processing after a line has been passed n 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 23 when i=age1"].

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 any arguments (Figure 1). The watch command is especially 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 linenum<ret>*. The command to clear a watchpoint is *Delete W varname<ret>*. You can save time with *DB _all_<ret>* or *DW _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. You set break points so that you can issue display commands at interesting points in your program. Once you have halted execution with a break you will use display commands to show the PDV and input buffer.

Examine is the most useful display command. It 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 can be very useful in making output more readable. It applies a format to the value in the PDV.

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*) can be tedious. To save typing, the examine command is usually bound to a function key (explained below) or the programmer uses <F4>.

Examine does not accept the following SAS abbreviations:

Examine _numeric_ **or** Examine _character_ **or** e var_nm1-var_nm3 **or** e varnm1--varnmchar

List is a DSD display tool and writes messages to the DSD log to describe your DSD session. It has several options. *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 you have typed in the wrong column number). 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 we can associate conditions with the breakpoints when we set breakpoints. We 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 secvar thirdvar<ret>*. Describe causes SAS to write the variable name, type and length to the DSD log like a Proc Contents on the variables specified.

DSD WINDOW COMMANDS

The window command toggles the active window between editor and log. This is useful because only the active window has a scroll bar and a command line. You will issue the WSAVE command on the command line of every window whose shape you want SAS to remember. To scroll a window, it must be active. 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 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.

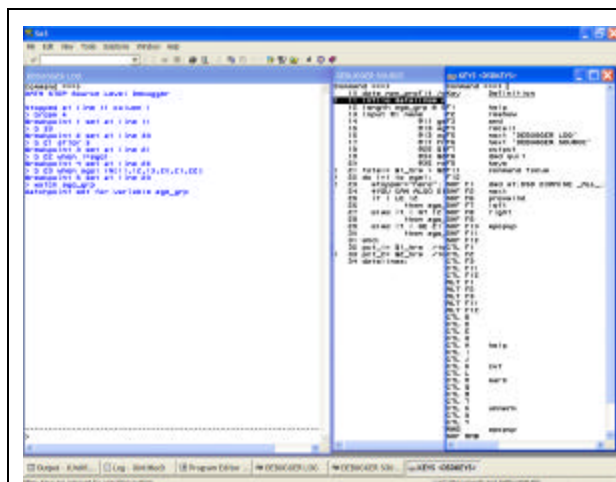


FIGURE 2

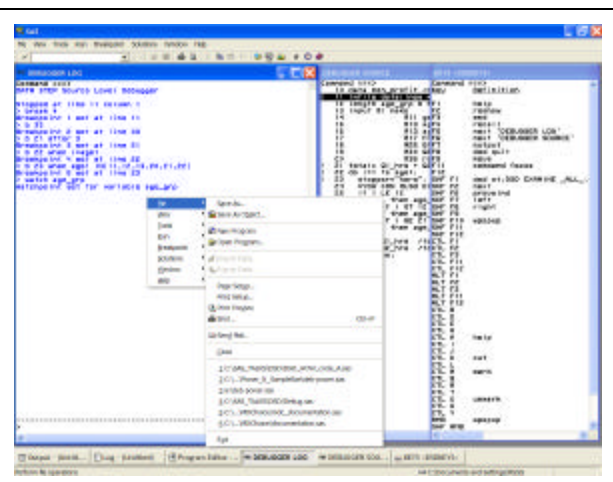


Figure 3

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 "command series" (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.

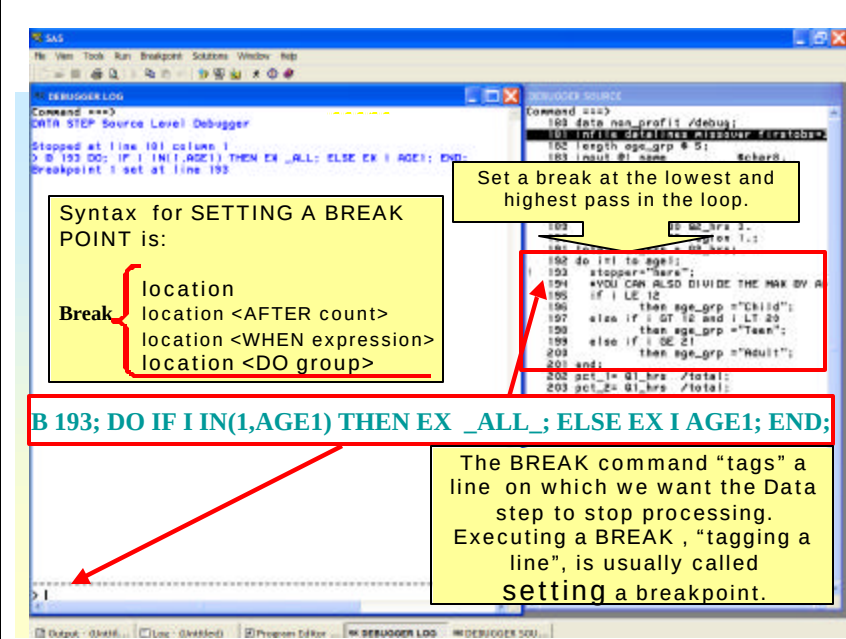
The keys window shows the key bindings that are in effect when the author's DSD is running. These key bindings are not valid when regular SAS is running. If I were to press <F9>, when in the regular SAS editor, a keys window would be displayed. However, the key bindings would be different from the DSD key bindings shown above. 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 (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. Open the key window and use the mouse to position the cursor on a line. 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. – see 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.

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. 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 very 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 feature on the menu. For most DSD commands, typing commands is usually faster than using a mouse and menu.



Multiple DSD commands can be "set to execute" after a break by enclosing them in a Do-End. This has a use in debugging loops. If's are supported.

Often, there is an interest to see values 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 the loop counter and age1. A user, at any time, could issue additional examine commands to EXamine other variables in the PDV.

Syntax for SETTING A BREAK POINT is:

Break { location
location <AFTER count>
location <WHEN expression>
location <DO group>

B 193; DO IF I IN(1,AGE1) THEN EX _ALL_; ELSE EX I AGE1; END;

The BREAK command "tags" a line on which we want the Data step to stop processing. Executing a BREAK, "tagging a line", is usually called setting a breakpoint.

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);
  %&lineno do;
    if (i=i 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 new_profit /debug;
infile datalines missover firstobs=3;
length age_grp $ 5;
input @1 name
      @11 gender $char1;
      @13 age1 2.0;
      @13 age2 2.0;
      @17 From $char5;
      @25 Q1_hrs 3.;
      @30 Q2_hrs 3.;
      @35 region 1.;
total= Q1_hrs + Q2_hrs;
do i=1 to age1;
  stopper="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

Typing a series of DSD commands, saving them in the macro catalog and recalling then into the DSD is another speed trick.

Code in Figure 5 will create macros in the normal SAS Macro Catalog that can be called from the DSD.

To call the macro named G and have the DSD stop on line 193, the user would type on the DSD log instruction line (see red arrow in Figure 4) command %g(193).

To use the LP macro on the code in Figure 4, the user might type

%lp(193,age1),ret>
and then a series of Go commands Or one go and then a series of <F4> <ret> commands.

CLEAN CODE FROM RUNNING A MACRO – FOR INPUT INTO THE DSD

Sometimes it is difficult to debug macro code. There are occasions where the programmer is unsure if the macro code evaluated to something unintended or if the macro code evaluated as intended, but the logic that was programmed into the macro is flawed (and a candidate for analysis using the DSD). The programmer needs to have clean code to be put into the DSD. In the past, programmers occasionally copied the log into an editor and manually cleaned SAS comments and notes from the code to create code for the DSD. Clean code can now be captured with the following easy method.

Imagine a macro defined as:

```

%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 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 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 means data=new_PLACE;
run;
```

The options and filename statements open a file that collects the evaluated macro code. The % series statement runs the macro and closes the file. 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. 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 DEBUGGER SESSION

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

CONCLUSION

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 and effective use of the DSD requires understanding the PDV.

REFERENCES

Heffner and Schaffer, “The Data Step Debugger “, Proceedings of the Twentyth 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:

Russell Lavery Email: russ.lavery@verizon.net
or c/o Numeric, LLC
5 Christy Drive
Bldg.2 Suite 107
Chadds Ford, PA, 19317 (610)642-0700

SAS is a registered trademark or trademark of SAS Institute, Inc. in the USA and other countries. ® indicates USA registration.

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

*****BELOW, PLEASE FIND SAMPLE CODE THAT CAN BE PASTED INTO SAS AND USED AS A VEHICLE FOR LEARNING*****

THE CODE IS ORGANIZED INTO SECTIONS FOR EASY PRACTICE. THE SECTION HEADINGS SUGGEST DSD COMMANDS THAT MIGHT BE USED ON CODE IN THAT SECTION.

First: make sure you have stored, at least, some DSD code in a key. This is a basic speed trick. Good code is:

DSD step 1; DSD examine _all_;

```
*****;
/* macros for calling from the DSD */
/* 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;
```

```
*****;
*****;
**                                     **;
** This is the section for introducing the Datasets **;
**                                     **;
*****;

OPTIONS NOCENTER;
proc format;
value born_in 1="NE"
              2="SE"
              3="NW"
              4="SW"
              5="Intl";

run;

data humans (sortedby=name);;
infile datalines missover firstobs=2;
input @1 name $char8. @10 time_W_co @15 sex $char1. @18 job $char4. @25 line_no;
datalines;
123456789012345678901234567890
Art      1      M  Stat   1
Brent    6      M  Stat   2
Cathy    6      F  Stat   3
Debbie   5      F  Stat   4
Jen      13     F  Stat   5
Russ     2      M  Prog   6
Saad     3      M  Prog   7
Tom       6     M  Stat   8
Zim       6     M  Stat   9
;
run;
```

```

data pets (sortedby=name);
infile datalines missover firstobs=2;
input @1 name $char8. @9 repeat @15 type $char3. @20 Pet_nm $char8. ;
datalines;
123456789012345678901234567890
Art      1      Dog  Catcher
Brent    1      N
Cathy    1      Dog  R.A.
Debbie   1      Dog  Dusty
Jen       1      Cat  Princess
Jen       2      Cat  Gabbie
Jen       3      Cat  India
Mary     1      Dog  Rao
Saad     1      N
;
run;

```

```

*****
*0      Step return jump go quit ;
*****

*Snippet 0A Step return jump go quit ;
*****
**
** Example of a how knowledge of the PDV is essential
** There are no error messages, just a list of values that you must
** interpret according to your expectations
** use the f4 key and binding to keys
** look at when num_h is created- before the set statement executes
**
*****
* use this code for issuing step <RET> jump go and quit;
* This is going to take a 1 out of 2 sample from work.humans ;
data sample/debug; /*focus on _N_, values in the PDV and num_h*/
  do obsnum=1 to num_h by 2;
    set humans point=obsnum nobs=num_h;

    if _error_ then abort;
    output;
  end;
  stop;
run;

*****
*1      Examine, list and describe ;

```

```

*****
*Snippet 1A Examine, list and describe;
*****
**
** Example of a how knowledge of the PDV is essential
** There are no error messages, just a list of values that you must
** interpret according to your expectations
** use the f4 key and binding to keys
** look at when num_h is created- before the set statement
**
*****
*one way to select a subset of the data - use pointer control and a do ;
*use this to practice examine list and describe commands;
data sample/debug; /*focus on _N_, values in the PDV and num_h*/
*just a non-executable comment;
  do obsnum=1 to num_h by 2;
    *another non-executable comment;
    set humans point=obsnum nobs=num_h;
    if _error_ then abort;
    output;
  end;
  stop;
run;

*Snippet 1B Examine, list and describe;
*****
**
** Danger of recoding in a one to many merge

** see the first. and last. variables in a t of values that you must
** interpret according to your expectations
** use the f4 key and binding to keys
** look at when num_h is created- before the set statement
**
*****
*the danger of recoding in a one to many merge - first. and last. variables;
*use jump to skip the calculation of time _w_co;
* there is a danger, in one to many merges, in modifying variables on the one;
data matching/debug;
  merge humans(in=h) pets(in=p);
  *stop_pt=1;
  by name;
  time_w_co=time_w_co*12;
  stop_pt=1;
run;

proc print;
run;

*Snippet 1C Examine, list and describe;
*****

```

```

*;

**
** Look at the input buffer - not just the PDV
** list I, b W _all_
** notice that the input buffer contains Brent
** but the variable name only contains rent
** also see the list command as a tool for managing break and watch points
** ISSUE THE DESCRIBE _ALL_ COMMAND -Does show formats - see someday

*****
** you can compare the PDV and the input bufffer to see what is wrong with art;
data pets2 (sortedby=name)/debug;
infile datalines missover firstobs=2;
input @2 name $char8. @9 repeat 4.2 @15 type $char3. @20 Pet_nm $char8.
      @9 someday ;
format someday date9.;
datalines;
123456789012345678901234567890
  Art    1      Dog  Catcher
Brent    1      N
Cathy    1      Dog  R.A.
Debbie   1      Dog  Dusty
Jen       1      Cat  Princess
Jen       2      Cat  Gabbie
Jen       3      Cat  India
Mary     1      Dog  Rao
Saad     1      N
;
run;

*Snippet 1D Examine, list and describe;
*****
**
** USE THIS TO SHOW EXAMINE, SHOWING DATES WITH FORMATS, FORMATS FOR NUMBER
** use the f4 key and binding to keys
** USE EXAMINE ON THIS_
**
*****
**use this code to show effect of keep on data step statement vs set statement;
**This effect show up on the mergeing of left and right on line 233;
data left;
infile datalines missover firstobs=3;
input @1 name $char4. @8 gender 1. @10 enroll DATE9.;
THIS_DATE=TODAY();
ENROLLED=THIS_DATE-ENROLL;
datalines;

      1      2      3      4      5
12345678901234567890123456789012345678901234567890
Sai    1 01JAN2003
Yung   1 05FEB2002
Yong   1 12MAR2002
ODD    3 22MAR2004
Sue    2 22SEP2001
Bob    1 18AUG1998

```

```

RUSS 1 19JUN1997
LIJI 1 30SEP1995
MUSA 1 20JUL2001
DEB 2 11MAR2002
;
RUN;

```

```

data RIGHT;
infile datalines missover firstobs=3;
input @1 name $char4. @8 CHILDREN 1. @10 SCHOOL $CHAR1. @15 HAS_BOY $CHAR1. @20 HAS_grl
$CHAR1.;
THIS_DATE=TODAY();
datalines;
          1          2          3          4          5
12345678901234567890123456789012345678901234567890
Sai 1 NO Y N
Yung 2 YES Y Y
Yong 1 NO Y N
ODD 3 YES Y Y
RUSS 0 N N
Sue 2 YES N Y
Bob 2 NO Y N
LIJI 2 NO Y Y
DEB 3 YES Y Y
;
RUN;

```

```

*SNIPPET 1D_A
**SHOW A KEEP ON THE INPUT VS A KEEP ON THE OUTPUT DATA SET;
data show_keep1(keep=F_NAME wks_enr) /debug;
set left (RENAME =(NAME=F_NAME));
wks_enr=enrolled/7;
run;

```

```
PROC PRINT;
```

```

TITLE "WE HAD MANY VARIABOLES ON THE pdv , BUT FEW IN OUTPUT DATASET";
RUN;

```

```

**A SNIPPET FOR INDEPENDENT EXPLORATION;
*CODE BELOW WILL CAUSE AN ERROR - RENAMED VARIABLE IS ON THE pdv AND CAN NOT BE KEPT ON
OUTPUT;
data show_keep2(keep=name wks_enr) /debug;

set left(keep=name wks_enr RENAME =(NAME=F_NAME));
wks_enr=enrolled/7;
run;

```

```

**ANOTHER SNIPPET FOR INDEPENDENT EXPLORATION;
** the PDV is created before the data is read, so the first definition of has_a_boy sets
length;

** the order of Y and N, in the data does not matter;
Data Problem/DEBUG;
set right;

```

```

if has_boy="N" then has_a_boy="NO";*use describe to see has_a_boy will be length 2;
  else has_a_boy="YES";

run;

Data OK_length/DEBUG;
set right;
  if has_boy="Y" then has_a_boy="YES";*use describe to see has_a_boy will be length 3;
  else has_a_boy="NO";
run;

```

```

*****
*      BREAK, Delete Watch Trace
*****

```

```

*Snippet 2A  BREAK, Delete Watch Trace ;
**This is a GOOD TRICK. Variables from input are not retained - watch will have problems;
** variables from a set statement are retained - so less useful for vars from data set;
** run PBLM_Pets and set a watch for repeat;
**then uncomment the retain on line 290 and run again;
data Prblm_pets (sortedby=name)/debug;
infile datalines missover firstobs=2;
*retain repeat;
input @2 name $char5. @9 repeat 4.2 @15 type $char3. @20 Pet_nm $char8.
      @9 someday ;
format someday date9.;
datalines;
123456789012345678901234567890
  Art    1      Dog  Catcher
  Brent  1      N
  Cathy  1      Dog  R.A.
  Debb   1      Dog  Dusty
  Jen    1      Cat  Princess
  Jen    2      Cat  Gabbie
  Jen    3      Cat  India
  Mary   1      Dog  Rao
  Saad   1      N
;
run;

```

```

*Snippet 2B  BREAK, Delete Watch Trace ;
*****
** Set break point
** Look at the input buffer - not just the PDV
** list I, b w _all_
** notice that the input buffer contains Brent
** but the variable name only contains rent
** also see the list command as a tool for managing break and watch points
** ISSUE THE DESCRIBE _ALL_ COMMAND -Does show formats - see someday
*****

```

```

*Snippet 2A  BREAK, Delete Watch Trace ;
*use for debugger for a do loop;
*break when I = 1 and when I = exit condition to check start and end of looping;

```

```

*you can divide the max by an integer and use in the break;
*put in a non-ececutable line;
*break with an in to break just before / after the cutpoints;
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;
PUT _all_;
do i=1 to age1;
  stopper="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;
pct_1= Q1_hrs /total;
pct_2= Q1_hrs /total;
datalines;
      1          2          3          4
1234567890123456789012345678901234567890
Sue      F  10  Florida 017  022  2
Mike     M  12  Conn   027  042  1
Erin     F  13  Ireland 021  012  5
Ya-Ching F  20  China   017  024  5
Ann      F  21  Mass    022  014  1
Jeff     M  45  Calif   078  099  4
Yuan     F  27  Nevada  026  046  4
Suna     F  31  India   017  011  5
Eric     M  45  Oregon  041  023  3
;
run;

proc print data=non_profit;
run;

*****
**  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;
  stopper="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;
pct_1= Q1_hrs /total;
pct_2= Q1_hrs /total;
datalines;
      1          2          3          4
1234567890123456789012345678901234567890
Sue      F 10 Florida 017 022 2

```

```

Mike      M  12  Conn    027  042  1
Erin      F  13  Ireland 021  012  5
Ya-Ching  F  20  China   017  024  5
Ann       F  21  Mass    022  014  1
Jeff      M  45  Calif   078  099  4
Yuan      F  27  Nevada  026  046  4
Suna      F  31  India   017  011  5
Eric      M  45  Oregon  041  023  3
;
run;

```

```

*****
**                                                                 **
**  PRACTICE ON THESE EXAMPLES                                  **
**                                                                 **
*****
*****
**                                                                 **
**  COLLECT HOW A MACRO EVALUATES AND STORES CLEAN CODE IN A TEXT FILE **
**                                                                 **
*****

```

```

%MACRO series;
data new;
total= %do i=6 %to 12;
&i %str(+) %end;  0 ;
run;

```

```

proc print data=&dset;
run;

```

```

proc means;
run;

```

```

%MEND series;

```

```

/* the mprint mfile commands are for version 8 and up.
If you have an earlier version of SAS use
    options mprint reservedb1;
    filename mac_list 'c:\temp\lookSAS.txt';
    %the_macro_call;
    x fclose(c:\temp\ lookSAS.txt);
*/

```

```

options mprint mfile;
filename mprint "c:\temp\series_txt.txt";

```

```

%series; *this runs the macro and
options nomprint;

```

```

*****
**
** Example of if _N_=1 then set statement
**
** do this in steps, first look at data no_read
** use the f4 key and binding of commands to keys
** USE THIS, w/ break at LOOK, TO SHOW THAT SET LAST IS VALUED BEFORE THE SET
** SHOW THAT LAST IS NOT OUTPUT TO THE DATA SET & THEN APPLY THE KEEP
** SOW AUTOMATIC RETAIN FOR VARIABLES FROM A SET STATEMENT
*****

```

```

data no_read/debug;
*data no_read(keep=obs_perm) /debug;
*look="look"; *comment and uncomment this line - examine _all_ here;
if _N_ = 0 then set left nob=last;
obs_perm=last;
run;

```

```

proc print;
run;

```

```

data show_N(keep=name wks_enr) /debug;
if _n_=1 then set no_read;
set left;
wks_enr=enrolled/7;
run;

```

```

*****
** NO RETAIN IF YOU USE AN INPUT STATEMENT
** This is from the SAS help file
** This illustrates how there is NO retain for a input statement
** the problem show up when _n_=2
**
** USE THIS TO SHOW:
** BREAKS & DELETE BREAKS, EXAMINE ALL
** SET A BREAK WHEN TOUR CHANGES AS AN ILLUSTRATION
** us the f4 keys and binding to keys
*****

```

```

/* first execution */
data tours (drop=type);
*RETAIN TOUR;
infile datalines missover firstobs=3;
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;
1 2 3 4 5
1234567890123456789012345678901234567890

```

```

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;

```

```

*****
**                                MERGE EXAMPLE                                **
** one to one, two to 1, one to two, three to two and two to three merge **
**                                                                **
** This is mostly a view of the SAS merge, and to see in=variables          **
** use breaks examine                                                       **
** break where r=0                                                           **
** use f4 and keys                                                           **
*****

```

```

DATA L_MEDS;
INFILE DATALINES MISSOVER firstobs=4;
/*KEEP THE UNREAD DATA IN FOR A WHILE - MAYBE WE'LL GET MORE VARIABLES*/
input @1 name $char8.
      @10 LEFT_LINE 1.
      @13 L_OB_CD   $CHAR4.
      @19 gender    1.
      @22 INF_con_frm $CHAR3.
      @27 VNUM      1.
      @32 AM_PM     $CHAR2.
      @34 EXER_TM   TIME8.
      @44 LAB       $CHAR7. ;

```

```

format EXER_TM time6.2;
datalines;
name  LEFT_LINE sex ICF VNUM AM/PM TIME LAB
      1      2      3      4      5
12345678901234567890123456789012345678901234567890
_1_1*1M 1 L1_1 1 YES 1 AM 2:20 ACME
_2_2*1M 2 L2_1 1 YES 1 AM 2:20 ACME
_2_2*1M 3 L2_2 1 YES 1 PM 2:40 GENERIC
_3_1*2M 4 L3_1 2 YES 1 AM 2:10 GENERIC
_4_2*3M 5 L4_1 2 YES 1 PM 2:19 ACE
_4_2*3M 6 L4_2 1 1 AM 1:58 ACE

_5_3*2M 7 L5_1 1 1 PM 2:02 ACME
_5_3*2M 8 L5_2 1 YES 1 AM 2:50 ACME

_5_3*2M 9 L5_3 1 YES 2 PM 2:02 ACME
;

```

```

RUN;
PROC SORT DATA=L_MEDS;

```

```

BY NAME LEFT_LINE;
RUN;

```

```

DATA R_RUN_TIMES; /*SUBJECTS A HALF MILE AND RUN TWICE A DAY*/
INFILE DATALINES MISSOVER FIRSTOBS=4;

```

```

input @1 name $char8.
      @11 RIGHT_LINE 1.

      @14 R_OB_CD $CHAR4.

      @21 RAIN_YN 1.

      @26 RUN_ALONE $CHAR3.
      @33 SPLITS_YN $CHAR3.
      @37 ELAPS_TM TIME10.
      @50 TRACK $CHAR12.
      @64 WARMUP $CHAR3.

```

```

;
datalines;
NAME RIGHT_LINE RAIN ALONE? SPLITS ELAPS_TM TRACK WARMUP
      1 2 3 4 5 6 7
123456789012345678901234567890123456789012345678901234567890
_1_1*1M 1 R1_1 1 NO NO 2:20 mi/4 CINDER YES
_2_2*1M 2 R2_1 1 NO YES 2:40 mi/4 TARTAN YES
_3_1*2M 3 R3_1 2 YES NO 2:19 mi/4 TARTAN NO
_3_1*2M 4 R3_2 2 YES NO 2:19 mi/4 TARTAN NO
_4_2*3M 5 R4_1 0 NO NO 1:58 1/8 MI NO
_4_2*3M 6 R4_2 0 NO NO 2:02 1/8 MI NO
_4_2*3M 7 R4_3 0 NO NO 2:02 1/8 MI NO
_5_3*2M 8 R5_1 0 NO NO 2:02 1/8 MI YES
_5_3*2M 9 R5_2 1 NO YES 2:50 mi/4 TARTAN YES
;
RUN;

```

```

PROC SORT DATA=R_RUN_TIMES;
BY NAME RIGHT_LINE;
RUN;

```

```

*1 use a where r=0 to stop processing when we fail to read from r; *r always is 1;
* use a watch on to jump to the next value of name - then step through;
*mention that one ds has 8 obs and another has 9 then examine for _n_ ;
OPTIONS LS=120;
DATA BOTH1 ;
MERGE L_MEDS (IN=M ) END=ENDMED
      R_RUN_TIMES (IN=R ) END=ENDRUN;
BY NAME;
PERM_M=m;
PERM_r=r;
P_ENDMED=ENDMED;
P_ENDRUN=ENDRUN;
DUMMY="XX";
RUN;

```

```

PROC PRINT DATA=BOTH1;

```

```

var      name L_OB_CD R_OB_CD LEFT_LINE RIGHT_LINE  PERM_r      PERM_M  P_ENDMED  P_ENDRUN
TRACK;
RUN;

```

```

*****
**                                                                 **
**              SHOW ISSUES WITH LAG VARAIBLES                    **
**                                                                 **
*****

```

```

data oops;
infile datalines missover firstobs=3 ;
input  @1 name $char3. @5 vnum 1. @8 pulse 2. @12 TST_CTRL $CHAR1.;
datalines;

```

```

      1      2
12345678901234567890
bob 0  72  T
bob 1  72  T
bob 2  74  T
bob 3  75  T
SUE 0  72  C
SUE 1  72  C
SUE 2  74  C
SUE 3  75  C
Lun 0  72  T
Lun 1  72  T
Lun 2  74  T
Lun 3  75  T

```

```

;
run;

proc sort data=oops;
by name vnum;
run;

```

```

proc print data=oops;
run;

```

```

data oops2;
set oops;
by name vnum;
if first.name =0 then old=lag(pulse);
run;

```

```

proc print data=oops2;
run;

```

```

*****
**                                                                 **
**              SHOW END=EOF AND DIFFERENT FUNCTIONING OF WHERE VS IF          **
**                                                                 **
*****

```

```

data oops2;
infile datalines missover firstobs=3 ;
input  @1 name $char3. @5 vnum 1. @8 pulse 2. @12 TST_CTRL $CHAR1.;
datalines;

```

```

      1      2

```

12345678901234567890

```
bob 0 72 T
bob 1 72 T
bob 2 74 T
bob 3 75 T
SUE 0 72 C
SUE 1 72 C
SUE 2 74 C
SUE 3 75 C
Lun 0 72 T
Lun 1 72 T
Lun 2 74 T
Lun 3 75 T
```

```
;
run;
```

```
proc sort data=oops2;
by name vnum;
run;
```

```
proc print data=oops2;
run;
```

```
*USE A DATA STEP TO COUNT THE NUBER OF OBS;
DATA _NULL_/debug;
SET OOPS2 END=EOF;
BY name vnum;
stopper="here";
*IF upcase(TST_CTRL)="C";
where upcase(TST_CTRL)="C";

counter+1;
if EOF=1 then
do;
put "this is the end and we had " counter "observations";
end;
run;
```

```
*****
**
** MAYBE SHOW BOB VIRGILS COOL WHERE ON PAGE 87 OF NESUG 93
**
*****
*not yet in place;
```

```
*****
**
** maybe SHOW attempts to use macro variables in DSD
**
*****
```

```

OPTIONS MPRINT MLOGIC SYMBOLGEN;
*create a couple of macro variables so we can toggle back and forth;
*execute one or the other of the macro statements below;
%LET TRACK=mi/4 TARTAN;
%LET TRACK=1/8 MI;

OPTIONS LS=120;*THIS RUNS AND uses the value of track to produce output;
DATA BOTH2 /DEBUG;
MERGE L_MEDS      (IN=M ) END=ENDMED
      R_RUN_TIMES (IN=R WHERE=( TRACK="&TRACK")) END=ENDRUN;
BY NAME;
MACRO_IS="&TRACK"; *JUST A WAY TO LOOK AT THE MACRO VARIABLE;
PERM_M=m;
PERM_r=r;
P_ENDMED=ENDMED;
P_ENDRUN=ENDRUN;

DUMMY="XX";
  TRACK_COUNT+1;
IF ENDRUN THEN PUT "SUBJECTS USED &TRACK TRACKS " TRACK_COUNT " TIMES";

RUN;

```

```

OPTIONS LS=120;*THE IF STATEMENT DELETES THE OBS WHEN ENDRUN =1;
DATA BOTH3 /DEBUG;
MERGE L_MEDS      (IN=M ) END=ENDMED
      R_RUN_TIMES (IN=R ) END=ENDRUN;
BY NAME;
MACRO_IS="&TRACK"; *JUST A WAY TO LOOK AT THE MACRO VARIABLE;
IF TRACK=: "PUT(SYMGET('TRACK'))";
PERM_M=m;
PERM_r=r;
P_ENDMED=ENDMED;
P_ENDRUN=ENDRUN;

DUMMY="A STOP POINT";
  TRACK_COUNT+1;

IF ENDRUN THEN
  DO;
    PUT;PUT;
    PUT "SUBJECTS USED &TRACK TRACKS " TRACK_COUNT " TIMES";
  END;
DUMMY2="A STOP POINT AFTER THE COMMAND THAT INTERESTS ME";
RUN;

```