Paper 121-2007

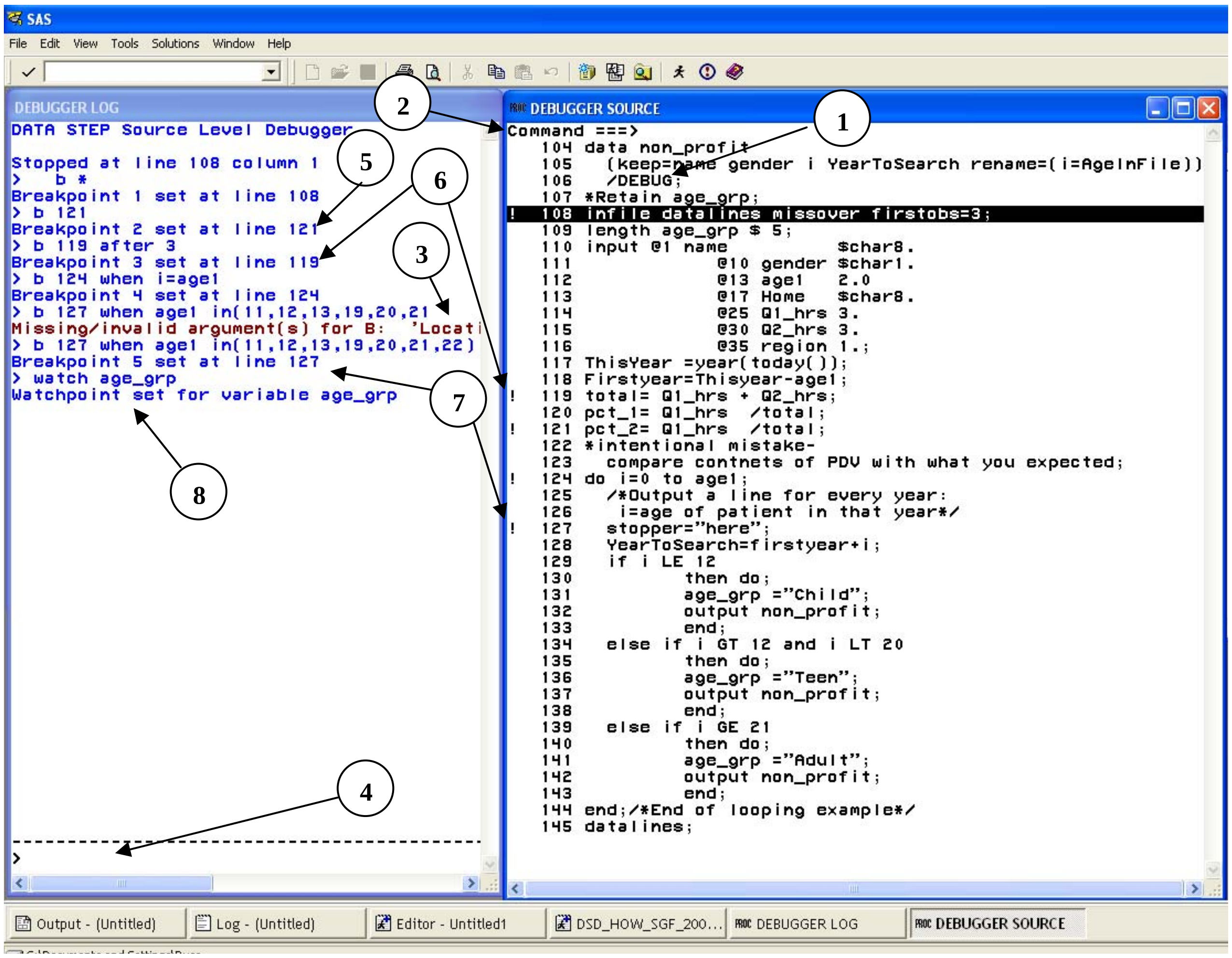
# An Animated Guide: The SAS® Data Step Debugger

Russ Lavery: Contractor for Numeric Resources LLC, Chadds Ford PA

## ABSTRACT

The Data Step Debugger (DSD) simplifies debugging Data Steps and not whole programs. 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, 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.



```
DATA STEP Source Level Debugger

Stopped at line 108 column 1
>   b *
Breakpoint 1 set at line 108
> b 121
Breakpoint 2 set at line 121
> b 119 after 3
Breakpoint 3 set at line 119
> b 124 when i=age1
Breakpoint 4 set at line 124
> b 127 when age1 in(11,12,13,19,20,21
Missing/invalid argument(s) for B:   'Locati
> b 127 when age1 in(11,12,13,19,20,21,22)
Breakpoint 5 set at line 127
> watch age_grp
Watchpoint set for variable age_grp



-----------------------------------------------
>
```

```
Command ===>
     104 data non_profit
     105    (keep=name gender i YearToSearch rename=(i=AgeInFile))
     106    /DEBUG;
     107 *Retain age_grp;
!    108 infile datalines missover firstobs=3;
     109 length age_grp $ 5;
     110 input @1 name          $char8.
     111             @10 gender $char1.
     112             @13 age1    2.0
     113             @17 Home    $char8.
     114             @25 Q1_hrs 3.
     115             @30 Q2_hrs 3.
     116             @35 region 1.;
     117 ThisYear =year(today());
     118 Firstyear=Thisyear-age1;
!    119 total= Q1_hrs + Q2_hrs;
     120 pct_1= Q1_hrs  /total;
!    121 pct_2= Q1_hrs  /total;
     122 *intentional mistake-
     123    compare contnets of PDV with what you expected;
!    124 do i=0 to age1;
     125    /*Output a line for every year:
     126     i=age of patient in that year*/
!    127    stopper="here";
     128    YearToSearch=firstyear+i;
     129    if i LE 12
     130            then do;
     131            age_grp ="Child";
     132            output non_profit;
     133            end;
     134    else if i GT 12 and i LT 20
     135            then do;
     136            age_grp ="Teen";
     137            output non_profit;
     138            end;
     139    else if i GE 21
     140            then do;
     141            age_grp ="Adult";
     142            output non_profit;
     143            end;
     144 end;/*End of looping example*/
     145 datalines;
```

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 learning the DSD should be to link DSD commands to a function key or a macro. Linking commands to a function key, or 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. If you leave out a semicolon, and try to run the DSD, the DSD screens will not be displayed. Referencing non-existing variables might cause the DSD to 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 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 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.

### 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 slow if you are a bad typist.

## DSD COMMAND GROUPS

The DSD commands are generally grouped by the functions they perform and will be 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 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.

**<u>Step</u>** **<u>Go</u>** **<u>Jump</u>** **<u>Quit</u>**

Suspension commands tell SAS where to stop execution of code. Execution must be stopped to display the PDV.

**<u>Break</u>** **<u>Watch</u>** **<u>Delete</u>** **Trace**

Display commands can show the PdV, and input buffer, in the DSD LOG.

**<u>Examine</u>** **List** **Describe**

Window commands toggle the active window between editor a/log. Only the active window has a scroll bar, and can be scrolled.

**Swap** **mouse-click**

Other commands do miscellaneous things

**Calculate** **<u>Set</u>** **Help**

*SPEED TRICKS make the DSD convenient to use*

**<u>Binding Commands to Keys</u>** **<u><F4></u>** **<u>Do-Loop</u>** **<u>Macro</u>** **<u>Calling a macro of DSD commands</u>**

## 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.

**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 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 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 whith 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. 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 the display commands, explained below, to show the PDV and input buffer.

**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 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 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. 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.

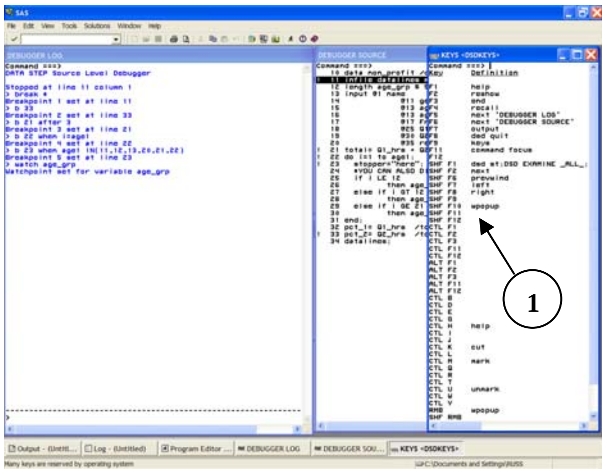
**Describe** writes information about variables to the DSD log. The format is *Describe firstvar secondvar thirdvar<ret>*. Describe causes SAS to write the variable name, type and length to the DSD log. This is like a Proc Contents, but just on the variables specified.

## 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.


SAS
File Edit View Tools Solutions Window Help
DEBUGGER LOG
DEBUGGER SOURCE
KEYS <DSDKEYS>
1
Output - (Untitl... Log - (Untitled) Program Editor ... DEBUGGER LOG DEBUGGER SOU... KEYS <DSDKEYS>
Many keys are reserved by operating system
C:\Documents and Settings\RUSS

```
Command ===>
DATA STEP Source Level Debugger

Stopped at line 11 column 1
> break *
Breakpoint 1 set at line 11
> b 33
Breakpoint 2 set at line 33
> b 21 after 3
Breakpoint 3 set at line 21
> b 22 when i=age1
Breakpoint 4 set at line 22
> b 23 when age1 IN(11,12,13,20,21,22)
Breakpoint 5 set at line 23
> watch age_grp
Watchpoint set for variable age_grp
```

```
Command ===>
Key       Definition

F1        help
F2        reshow
F3        end
F4        recall
F5        next 'DEBUGGER LOG'
F6        next 'DEBUGGER SOURCE'
F7        output
F8        dsd quit
F9        keys
F11       command focus
F12
SHF F1    dsd st;DSD EXAMINE _ALL_;
SHF F2    next
SHF F6    prevwind
SHF F7    left
SHF F8    right
SHF F9
SHF F10   wpopup
SHF F11
SHF F12
CTL F1
CTL F2
CTL F3
CTL F11
CTL F12
ALT F1
ALT F2
ALT F3
ALT F11
ALT F12
CTL B
CTL D
CTL E
CTL G
CTL H     help
CTL I
CTL J
CTL K     cut
CTL L
CTL M     mark
CTL Q
CTL R
CTL T
CTL U     unmark
CTL W
CTL V
RMB       wpopup
SHF RMB
```

FIGURE 2

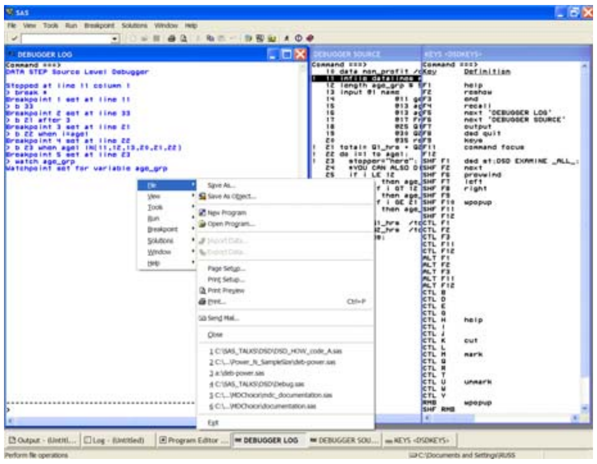

SAS
File View Tools Run Breakpoint Solutions Window Help
DEBUGGER LOG
DEBUGGER SOURCE
KEYS <DSDKEYS>
File
View
Tools
Run
Breakpoint
Solutions
Window
Help
Save As...
Save As Object...
New Program
Open Program...
Import Data...
Export Data...
Page Setup...
Print Setup...
Print Preview
Print... Ctrl+P
Send Mail...
Close
1 C:\SAS_TALKS\DSD\DSD_HOW_code_A.sas
2 C:\...\Power_N_SampleSize\deb-power.sas
3 a:\deb-power.sas
4 C:\SAS_TALKS\DSD\Debug.sas
5 C:\...\MDChoice\mdc_documentation.sas
6 C:\...\MDChoice\documentation.sas
Exit
Output - (Untitl... Log - (Untitled) Program Editor ... DEBUGGER LOG DEBUGGER SOU... KEYS <DSDKEYS>
Perform file operations
C:\Documents and Settings\RUSS

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.

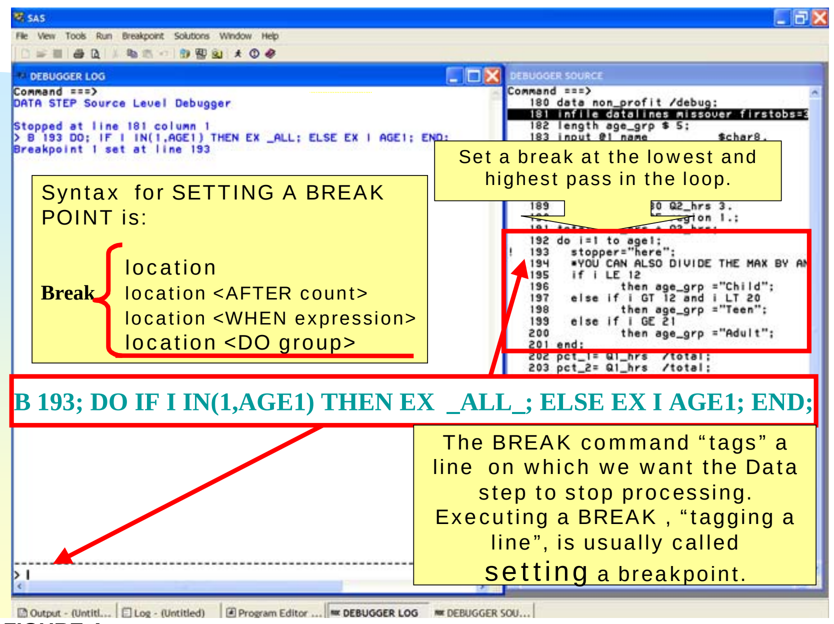
The keys window, in Figure 2, shows what key bindings that are in effect when the author's DSD is running and your key bindings may differ. These key bindings differ from bindings in effect when regular SAS is running. Pressing <F9>, when in the regular SAS editor, displays a keys window. However, the 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. 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. Your key bindings will only work when the DSD Log window is active.

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 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.


SAS
File View Tools Run Breakpoint Solutions Window Help
DEBUGGER LOG
DEBUGGER SOURCE
Output - (Untitl... Log - (Untitled) Program Editor ... DEBUGGER LOG DEBUGGER SOU...

```
Command ===>
DATA STEP Source Level Debugger

Stopped at line 181 column 1
> B 193 DO; IF I IN(1,AGE1) THEN EX _ALL; ELSE EX I AGE1; END;
Breakpoint 1 set at line 193
```

```
Command ===>
   180 data non_profit /debug;
   181 infile datalines missover firstobs=3
   182 length age_grp $ 5;
   183 input @1 name              $char8.
   189                 0 Q2_hrs 3.
                         gion 1.;
   192 do i=1 to age1;
!  193   stopper="here";
   194   *YOU CAN ALSO DIVIDE THE MAX BY AN
   195   if i LE 12
   196          then age_grp ="Child";
   197   else if i GT 12 and i LT 20
   198          then age_grp ="Teen";
   199   else if i GE 21
   200          then age_grp ="Adult";
   201 end;
   202 pct_1= Q1_hrs  /total;
   203 pct_2= Q1_hrs  /total;
```

Syntax for SETTING A BREAK POINT is:

**Break**
- location
- location <AFTER count>
- location <WHEN expression>
- location <DO group>

Set a break at the lowest and highest pass in the loop.

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

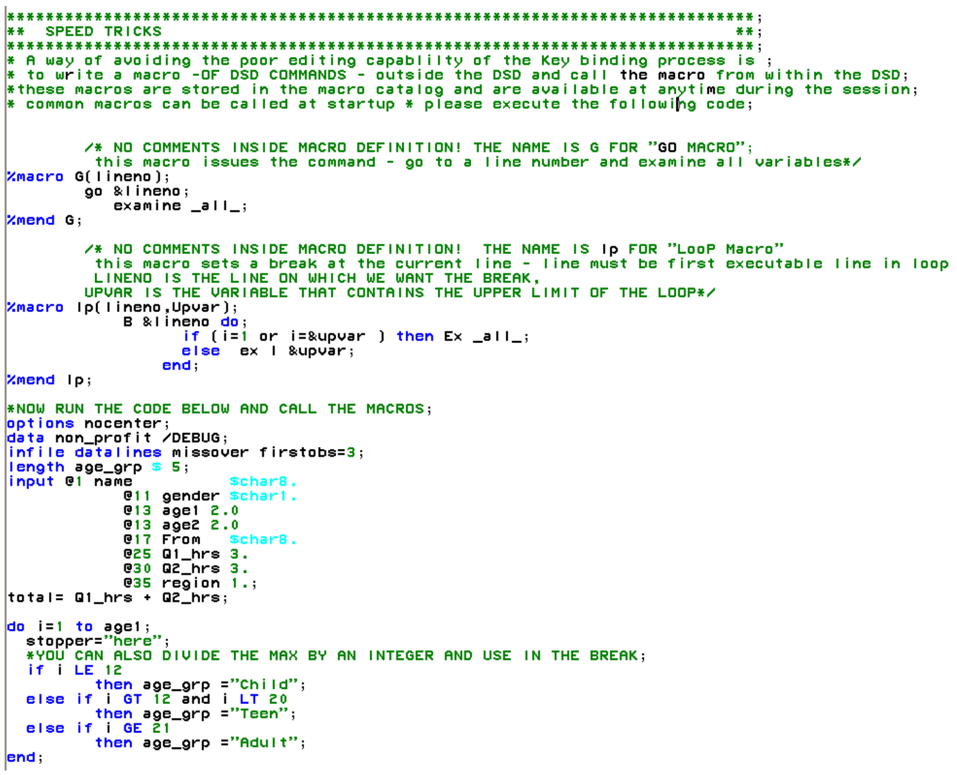
The BREAK command "tags" a line on which we want the Data step to stop processing. Executing a BREAK , "tagging a line", is usually called setting a breakpoint.

**FIGURE 4**

Multiple DSD commands can be "set to execute" after a break by enclosing them in a Do-End. 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 command in Figure 4 can be bound to a key but is easier to implement 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.



```
*************************************************************************;
**  SPEED TRICKS                                                       **;
*************************************************************************;
* A way of avoiding the poor editing capablilty 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 l &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;
```

FIGURE 5

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

Saving a series of DSD commands, in the SAS macro catalog and recalling then into the DSD is another speed trick. This trick gives you access to a good editor.

Code in Figure 5 is run outside the DSD 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 commands (Or one go and 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 therefore a candidate for analysis using the DSD). In this case, 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 clean code to be then run through the DSD. Clean code, generated by macro execution, can now be captured with the following easy method.

Imagine a macro defined as below. This macro bas 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 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 (called series_txt.txt) that collects the evaluated macro code. The %series statement runs the macro and options nomfile 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 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 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@russ-lavery.com
or c/o Numeric, LLC 5 Christy Drive Bldg.2 Suite 107 Chadds Ford, PA, 19317 (610)642-0700

***BELOW, IN A TINY FONT, FIND CODE YOU CAN PASTE INTO SAS AS A VEHICLE FOR LEARNING THE DSD***

**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.
Linking DSD commands in a key is a basic speed trick. Good first code to link is: DSD step 1; DSD examine _all_;

```
*******************************************;
*****************************************************************************;
**  April 24 -- DSD_HOW_SGF_2007  russ@russ.lavery.com                     **;
**  This is the section for introducing the Datasets                       **;
**                                                                         **;
*****************************************************************************;
OPTIONS NOCENTER;
proc format;
value born_in 1="NE"
              2="SE"
              3="NW"
              4="SW"
              5="Intnl";
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;
12345678901234567890123456789 0
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;
      *just another non-executable comment;
      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 commands to keys to reduce typing.            **;
** Look at when num-h is created- before the set statement.                 **;
** the creation of Num-h is, before the execution of any statemets, is      **;
**  evidence that there are commands that are "for" the compiler            **;
**;
******************************************************************************;
*One way to select a subset of the data - use Direct Pointer Control (point=) and a do ;
*Use this example 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;
*******************************************************************************;
**                                                                          **;
**  Shows the DANGER of recoding inside a data step doeing a one2many 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                                     **;
**  Notice when num-h is created- before the set statement                 **;
**                                                                          **;
*******************************************************************************;
* Shows the danger of recoding in a one to many merge                         ;
* Shows the 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                            **;
**  Try thse commands: 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 DESCRIBE _ALL_ COMMAND -Does show formats - see variable someday  **;
*******************************************************************************;
** You can compare the PDV and the input bufffer
to see what is wrong with reading the observations for Art and Brent;
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;
*******************************************************************************;
**                                                                          **;
** Issue the command Examine, SHOW 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 shows 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 truncover 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;
**Use Ex _all_ to see the PDV  ;
data show_keep1(keep=F_NAME wks_enr) /debug;
*keep on output brings all vars into PDV;
*the PDV shows the variable F_name and not NAME;
set left (RENAME =(NAME=F_NAME));
wks_enr=enrolled/7;
run;

PROC PRINT data=show_keep1;
TITLE "we had many variables on the PDV , but few in output dataset";
RUN;

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


set left(keep=name enrolled RENAME =(NAME=F_NAME));
wks_enr=enrolled/7;
FrstLetr=substr(name,1,1);
*Name was renamed on set--caracters are sent to F_name;
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 (datalines or text file) are NOT retained
**   they are set to missing at the top of the data step and watch variable command 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 298 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 = the exit condition.  Allows you 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
  (keep=name gender i YearToSearch rename=(i=AgeInFile))
  /DEBUG;
*Retain age_grp;
infile datalines missover firstobs=3;
length age_grp $ 5;
input @1 name          $char8.
            @10 gender $char1.
            @13 age1   2.0
            @17 Home   $char8.
            @25 Q1_hrs 3.
            @30 Q2_hrs 3.
            @35 region 1.;
ThisYear =year(today());
Firstyear=Thisyear-age1;
total= Q1_hrs + Q2_hrs;
pct_1= Q1_hrs  /total;
pct_2= Q1_hrs  /total;
*intentional mistake-
  compare contnets of PDV with what you expected;
do i=0 to age1;
  /*Output a line for every year:
   i=age of patient in that year*/
  stopper="here";
  YearToSearch=firstyear+i;
  if i LE 12
         then do;
         age_grp ="Child";
         output non_profit;
                  end;
  else if i GT 12 and i LT 20
         then do;
         age_grp ="Teen";
         output non_profit;
             end;
  else if i GE 21
         then do;
         age_grp ="Adult";
         output non_profit;
                  end;
end;/*End of looping example*/
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                                                           **;
*****************************************************************************;

*****************************************************************************;
**                                                                         **;
**  Most Importantly --- STORE AN ABBREVIATION IN A KEY                     **;
**                                                                         **;
*****************************************************************************;



*STORE A MACRO IN A KEY  The command below is useful;
/*  DSD STEP 1; DSD EXAMINE _ALL;  */




**Collect code in a file;
%let dsn=WhoWHo;

%MACRO series;
data &dsn;
total= %do i=6 %to 12;
&i %str(+) %end;  0 ;
run;
proc print data= &dsn;
title "Macro lookat";
run;
%MEND series;

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

options nomfile;




* 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;

options nocenter;
data non_profit (keep=name gender i YearToSearch rename=(i=AgeInFile)) ;
infile datalines missover firstobs=3;
length age_grp $ 5;
input @1 name          $char8.
            @10 gender $char1.
            @13 age1   2.0
            @17 Home   $char8.
            @25 Q1_hrs 3.
            @30 Q2_hrs 3.
            @35 region 1.;
*PUT _NUMERIC_;
ThisYear =year(today());
Firstyear=Thisyear-age1;
total= Q1_hrs + Q2_hrs;
pct_1= Q1_hrs  /total;
pct_2= Q1_hrs  /total;

do i=0 to age1; /*Output a line for every year: i=the age of the person in that year*/
  stopper="here";
  YearToSearch=firstyear+i;
  *YOU CAN ALSO DIVIDE THE MAX BY AN INTEGER AND USE IN THE BREAK;
  if i LE 12
         then do;
         age_grp ="Child";
         output non_profit;
                  end;
  else if i GT 12 and i LT 20
         then do;
         age_grp ="Teen";
         output non_profit;
              end;
  else if i GE 21
         then do;
         age_grp ="Adult";
         output non_profit;
                  end;
end;/*End of looping example*/
/*Short example of finding errors with the debugger*/
*intentional mistake-
   compare what is in the PDV with what you expect to see;
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;

/*******************************************************************
Section __: Cool if you can get to it
********************************************************************/
proc sort data=sashelp.class out=class;
by sex name;
run;
proc print data=class;
run;

data drat /DEBUG;
set class;
by sex;

*if AGE LT 15;
WHERE AGE LT 15;

if first.sex=1 then counter=0;
counter+1;
if last.sex then output;
run;

proc print data=drat;
run;

PROC SQL;
DROP TABLE DRAT;
QUIT;

**********************************************************************;
**********************************************************************;
**                                                                  **;
**                 The End - Thanks for attending                   **;
**                                                                  **;
**********************************************************************;
**********************************************************************;
```

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


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

proc print data=&dsn;
run;

proc means;
run;

%MEND series;


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

%let dsn=WhoWhoo;
options mprint mfile;   *sends data to file;
filename mprint "c:\temp\series_txt.txt";

%series;                *this runs the macro;
options nomprint;       *closes the file;



*****************************************************************************;
**                                                                         **;
**  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="look",                                      **;
**    Note THAT LAST IS VALUED BEFORE THE SET executes                     **;
**     SHOW THAT LAST IS NOT OUTPUT TO THE DATA SET & THEN APPLY THE KEEP  **;
**     SHOW AUTOMATIC RETAIN FOR VARIABLES FROM A SET STATEMENT            **;
*****************************************************************************;
*Example below is very short.  Run the code and issue ex _all_.  Note value of last;
*if you want header info in the output file you must copy it into another var see 531;
data no_read/debug;
*data no_read(keep=obs_perm) /debug;
*look="look"; *comment and uncomment this line - issue examine _all_ here;
if _N_ = 0 then set left nobs=last;
obs_perm=last;
run;

proc print data=no_read;
title "No Last here, but You can get information from the file header on/when compiling";
run;
title "";

data show_N_/debug;
if _n_=1 then set no_read;
set left;
wks_enr=enrolled/7; /* No business sense*/
run;
PROC PRINT ; title "Note that we have added a var to each line";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 shows 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
12345678901234567890123456789012345678901234567890
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
1234567890123456789012345678901234567890123456789012345678901234567890
_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 /DEBUG ;
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 VARIABLES                        **;
**                                                                           **;
*******************************************************************************;
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 /debug;
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;







*****************************************************************************;
**                                                                         **;
**   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";
bo="gus";
RUN;


*****************************************************************************;
**                                                                        **;
**  This shows an uncommon error with the IF                              **;
**                                                                        **;
*****************************************************************************;

%LET TRACK=1/8 MI;

OPTIONS LS=120;*THE IF STATEMENT DELETES THE OBS WHEN ENDRUN =1;
DATA Odd_if /DEBUG;
Set  R_RUN_TIMES (IN=R ) END=ENDRUN;
MACRO_IS="&TRACK"; *JUST A WAY TO LOOK AT THE MACRO VARIABLE;
Symget_macro_is=symget("track");
IF left(TRACK)= left(SYMGET('TRACK'));
PERM_r=r;
P_ENDRUN=ENDRUN;

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

IF ENDRUN=1 THEN
       DO;
       PUT;PUT;
       PUT "SUBJECTS USED &TRACK TRACKS " TRACK_COUNT " TIMES";
       END;
DUMMY2="A STOP POINT AFTER THE COMMAND THAT INTERESTS ME";
RUN;


**********************************************************************;
**********************************************************************;
**                                                                 **;
**                  Not in slides - see paper                      **;
**                     IORC MERGE DSD                              **;
**********************************************************************;
**********************************************************************;
data dAY_1;
infile datalines missover;
input @1  name   $char4.
      @10 Runner $char1.;
datalines;
Bob      Y
russ     N
Sue      N
AJ       N
Fred     Y
Glenn    N
KL       Y
; /*<- 7 observations need updating*/
run;

Data UpDt;
infile datalines;
input @1  name    $char4.
      @10 ShinSpl $char1.
      @15 T_elbow $char1. ;
label Shinspl=Shinsplints;
datalines;
Bob      S    N
eric     S    N
Sue      N    N
Fred     S
MarK     Y        <-NOTE MISSING VALUES
```

```
WalT     N
KL       N    Y
;
run;

proc datasets lib=work;
modify UpDt;
index create name/unique;
quit;

**no check of IORC ;
data new1;
set DAY_1;
set UpDt key=name/unique;
run;
/** you get errors ONLY for the folks in LEFt who do not match
name=russ Runner=N ShinSpl=S T_elbow=N _ERROR_=1 _IORC_=1230015 _N_=2
name=AJ Runner=N ShinSpl=N T_elbow=N _ERROR_=1 _IORC_=1230015 _N_=4
name=Glen Runner=N ShinSpl=S T_elbow=  _ERROR_=1 _IORC_=1230015 _N_=6
There are no error for mark and walt;
***/

* do the normal stuff- if _iorc_ then delete;
data new2;
set Day_1;
set UpDt key=name/unique;
if _iorc_  NE 0 then
  do;
   _error_=0;
   delete;
  end;
run;


data new3;
set Day_1;
set UpDt key=name/unique;
array setmiss(*) $
       ShinSpl--T_elbow;
if _iorc_ then
  do;
   _error_=0;
   ** we know that the fileds from UPDT are on the RHS of the PBV;
   ** watch out for mixed num & char ;
    do i= 1 to dim(setmiss);
     setmiss(i)="";
      end;
  end;
run;
```