SAS Data Step Debugger: Your Liberator from Logic Errors
Brandi Rhoads, Research Bureau of the Franchise Tax Board, Sacramento, CA

ABSTRACT
Have you ever written an intricate data step only to find that when you run the program the results are not as you expect? The SAS log is free of errors meaning you have rid your program of syntax errors; however, you know something is wrong. It must be a frustrating logic error! Using a simple example, this paper will illustrate basic debugger features you can use to find logic errors even in the most complex data steps. These techniques are useful to any programmer of SAS data steps, particularly those without prior knowledge of the data step debugger.

INTRODUCTION
One of the major tasks I have completed using SAS was to program a micro-simulation model of California state income taxes. To someone familiar with SAS, this actually means I programmed a massive data step that mimics all of the calculations in state tax policy to enable modeling of alternate policies using a weighted sample of tax returns as my input data set. In all, this involves almost 800 lines of code for the data step alone. You can imagine my frustration when the log contains no errors, however, the results are not as I expect. This is what is referred to as a logic error. The SAS software does not detect an error because all syntax is correct, but the lines of code do not process as the programmer intended. Learning basic features of the data step debugger enabled me to execute my data step code systematically to identify logic errors. Learning to use the debugger increased my efficiency and decreased my frustration. In the following paper I will use a simple example to teach and illustrate the usefulness of many of these features.

WHAT IS THE DEBUGGER?
The data step debugger allows you to control the execution of your data step code line by line, observation by observation and review variable values at any point during execution. Recall that by default, SAS processes data steps one statement at a time with one observation at a time. When the end of a data step is reached SAS will start the entire data step over with the next observation and repeat this process until all observations have been processed. Usually this occurs in rapid succession and all you can see are the final results as the completed data set or output from a procedure using the data set created. When the data step debugger is enabled an interactive screen will pop up enabling you to control the execution of the data step. There are additional commands you can apply in this environment that allow you to execute your code until a condition you set is met or a certain variable changes in value. The ability to “slow down” the execution of the code enables you to observe how values are affected by each line of code and therefore identify logic errors.

HOW TO EXECUTE AND CONTROL THE DEBUGGER
Starting the debugger is simple. You just need to add the DEBUG option to the first line of the data step as shown below:

```
DATA TAX / DEBUG;
```

Recall that adding this command will cause an interactive window to open within your base SAS session. If you wish to execute your SAS code as normal, you must exit the debugger interface and remove the DEBUG option.

The debugger screen has 2 main windows. The lower window will show the source code from your data step. Each line of code is numbered and the next line of code to be executed is highlighted. The upper window shows the debugger log and contains the command line. To execute the highlighted line of code you press the ENTER key. I will introduce a simple example to illustrate the “examine” and “watch” functions of the debugger.

ILLUSTRATING THE “EXAMINE” AND “WATCH” FUNCTIONS
The following code creates a dataset about taxpayers including an identification variable, their tax liability, and any credits they have available to offset this liability. The task is to create 2 new dummy variables to identify which taxpayers will need to make a payment and which will be receiving a refund.

```
DATA TAX;
INPUT ID TAX CREDITS;
```
DIFFERENCE=TAX-CREDITS;
IF DIFFERENCE>0 THEN DO;
  PAYMENT=1;
  REFUND=0;
END;
ELSE IF DIFFERENCE<0 THEN DO;
  REFUND=1;
  PAYMENT=0;
END;
DROP DIFFERENCE;
DATALINES;
1  500  300
2   300  .
3   250  500
4  1000  750
5   400  .
;
RUN;
PROC PRINT DATA=TAX;
RUN;

When you run this program there are no errors in the SAS log, however, the value for REFUND comes out as 1 for observations 2 and 5 who clearly have more liability than credits and should therefore owe a payment, not receive a refund. Because this data step is short you have likely already identified the problem, but if this line of code was vested in hundreds of other lines of code the debugger would be extremely valuable.

To initiate the debugger we will add the DEBUG option to the DATA line. The following screen will appear:

Figure 1. Results of Submitted Code
To execute the input statement for the first observation press ENTER. The highlighted bar will now move to the DIFFERENCE variable calculation indicating this is the next line of code to execute. At this point we would like to examine the values of the variables to make sure the input statement worked correctly. On the command line enter EXAMINE _ALL_ and press ENTER. Note that semi colons are not necessary after commands in debugger mode. Immediately the debugger log will show the current values of all of the variables in the program data vector for this point in execution. The value for ID, TAX, and CREDIT show up appropriately for the first observation and the values for DIFFERENCE, REFUND, and PAYMENT are missing since the code to calculate their values has not yet been executed. Press ENTER again to execute the line of code calculating the DIFFERENCE variable. To check that it calculated correctly we will enter EXAMINE (or E in shorthand) DIFFERENCE into the command line. Now it is displayed that DIFFERENCE = 200 which is correct for the first observation.

We can continue to press ENTER to execute the code line by line and then examine the variable values until we find the logic mistakes, however, there are additional commands to make this process easier. We know there is a problem with the calculation of the REFUND variable. It would be helpful to automatically be notified whenever the execution of a line of code causes the REFUND variable to change value. This is exactly what the WATCH function does. We will enter WATCH REFUND into the command line and select ENTER. Now as we continue to press ENTER to execute lines of code a message will appear in the log whenever REFUND changes value. You will be notified of the current value as well as the previous value. In the execution of the first observation you will be notified that REFUND has again been set to missing and the old value was 0.

This is an important time to note the behavior of the debugger when there is conditional logic in the data step. The debugger executes statements one at a time. Recall that if a conditional statement is not met, the related statements are not executed by SAS. For that reason if a conditional statement, such as IF DIFFERENCE>0 is not met, then the highlighted line of code (which is always the next one to execute) will automatically go to the next statement not pertaining to this conditional clause.

Now let’s start the execution of the second iteration of the data step. For the second observation we would expect DIFFERENCE to be greater than zero therefore causing PAYMENT to be set to 1 and REFUND set to 0, however, from the output we know this is not occurring correctly. When we press ENTER to execute line 4 we get a message in the debugger log that says “Stepped to line 8”. As I just explained this means the condition of line 4 was not met so lines 5 through 7 were not executed. We did not receive a message for watch point we set for REFUND so we know it did not change value as we would like. If we press ENTER twice more we see that the second conditional clause, DIFFERENCE<0, is met and our watch point for REFUND gives us a notification that the value has been changed to 1 at line 9. This clearly illustrates that our logical condition is failing to function as we intended. We would...
like to see the value of DIFFERENCE at this point in execution to see if it helps determine the cause. Enter E DIFFERENCE into the command line. The debugger log shows us that the value is missing! As the programmer, it was intended that the person has no credit available so their DIFFERENCE should be equal to their tax liability or $300. Recall that in SAS programming if one value is missing in an addition equation the answer will be missing. In addition SAS treats missing values as the smallest possible value in logical expressions so this explains why the missing value in DIFFERENCE was causing the conditional logic to execute in a way we did not expect. Using the debugger to analyze variable values as we executed the code enabled us to find this mistake. The calculation of the DIFFERENCE variable can be modified as follows to utilize a SUM function to correct for this mistake:

DIFFERENCE=SUM(TAX, -CREDITS);

ILLUSTRATING THE “GO” AND “BREAK” FUNCTIONS

We were able to identify the logic errors using only the “examine” and “watch” functions, but it involved repeatedly pressing ENTER to get through lines of code that weren’t actually part of the issue. If the data step had been long and more involved this could quickly get tedious. The solution for this is to set “break” points and use the “go” function to continuously execute the data step (as it would in normal SAS operation mode) until the break point is reached, at which time you can resume complete line by line control as previously explained. There are numerous criteria you can use to set break points including:

- A certain line of code to stop at as identified by the column numbers in the debugger mode
- A certain amount of times executing a statement before breaking (which is generally once per observation unless the statement resides inside a do-loop or conditional logic statement)
- A logical condition be met such as a variable being a certain value

The general formats for these conditions, respectively, are:

- BREAK line-number
- BREAK line-number AFTER number-of-iterations
- BREAK line-number WHEN expression

In our example we could have utilized a break point after a certain number of iterations since we knew the problem resided with the second and fifth observations. Let’s take a closer look at how this would work. As before, we would submit our code with the DEBUG option. Let’s intentionally stop the code before the execution of the calculation of DIFFERENCE when the 5th observation is loaded. Once the debugger interface came up we would enter the following into the command line: “BREAK 3 AFTER 5” and then hit enter. This has set a break point at line 3 (which corresponds to the DIFFERENCE calculation as shown in Figure 2) after 5 executions (which will be the beginning of the 5th observation). Once a break point has been set a note will appear in the debugger log stating what line it was set at and an exclamation mark will appear to the left of the line number in the debugger source code window. Once we have set where we would like the program to break we need to start the execution of the program. This is what the GO (or G for short) command is for. Simply enter G into the command line and press ENTER to submit it. The program will execute as in normal SAS mode until the break point is reached. When it stops we can enter EXAMINE _ALL_ to verify it has done as we wish. Indeed, the ID=5, the TAX and CREDITS variable correspond to the 5th observation and the other values are all missing since these statements have not yet executed. Now that we have quickly jumped to one of the observations in question, we can continue as before using the WATCH and EXAMINE functions. The GO command can also be used with the WATCH function. If you set the debugger to WATCH one or multiple variables and then enter GO the debugger will continuously execute lines of code until it encounters a change in one of the variables to be watched. At this point it will stop execution and allow user control once again.

CLEARING COMMANDS AND EXITING THE DEBUGGER

If you set multiple variables to be watched or break points you may find that some of them are not helpful or you would like to modify conditions based on what you discovered through first loops through the data step. The general form to clear a break point is DELETE BREAK line-number. To clear a watched variable enter DELETE WATCH variable-name. You can also specify _ALL_ if you would no longer like to watch any variables.

If you have determined the cause for your logic error and would like to exit the debugger interface so that you can correct your code you simply need to enter QUIT, or Q, into the command line followed by ENTER. If a data step reaches the end of its execution, meaning there are no more observations to process, you will also need to QUIT the debugger and resubmit the data step with the debugger option if you have not yet determined your mistake.
CONCLUSION

The debugger facility can be extremely useful for identifying logic errors in data step code. Using four fairly simple commands you can strategically execute and examine the values for your programming. Examine, watch, break, and go are the most useful commands in my experience when used in strategic combinations to control execution at key points in your program. There are additional debugger commands and ways to use the ones discussed in this paper that can be discovered in the SAS documentation.

CONTACT INFORMATION

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

   Name: Brandi Rhoads  
   Enterprise: Economic & Statistical Research Bureau, Franchise Tax Board  
   Address: MS A351, P.O. Box 1468  
   City, State ZIP: Sacramento, CA, 95812-1468  
   Work Phone: 916.845.7119  
   Fax: 916.845.5472  
   E-mail: brandy.rhoads@ftb.ca.gov

SAS and all other SAS Institute Inc. product or service names are registered trademarks or trademarks of SAS Institute Inc. in the USA and other countries. ® indicates USA registration.

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