Errors, Warnings, and Notes (Oh My)
A Practical Guide to Debugging SAS® Programs

Susan J. Slaughter, University of California Extension, Davis, CA
Lora D. Delwiche, IT/ANSA, University of California, Davis, CA

Why a paper on debugging SAS programs?

Most of the documentation about the SAS System doesn't even mention bugs, as if debugging wasn't worth talking about. This paper, on the other hand, is based on the belief that debugging is a good way to get insight into how SAS works. Once you understand why you got an error, you'll be better able to avoid it in the future. In other words, people who are good debuggers are good programmers.

Bugs can have different origins; some are accidentally built into the software by developers, others are introduced by programmers. Recently, one of the authors of this paper had a conversation about this topic with her father who is an aerospace engineer but not knowledgeable about the SAS System. The conversation went like this:

Susan: I'm writing about how to debug SAS programs.

Father: I thought they would have gotten the bugs out of SAS by now.

Fortunately, SAS Institute has done a good job of getting the bugs out of SAS software. Unfortunately, nobody has yet figured out how to get the bugs out of people.[1]

The SAS System even fixes some mistakes made by programmers. For example, SAS has gotten so smart over the years that it is now almost impossible to get an error by misspelling a keyword. If you misspell a keyword in a SAS program, SAS will almost always figure out what you meant to say and run the statement correctly in spite of your poor typing skills. But SAS can't fix all programming errors, so this paper discusses some of the most common bugs and how to exterminate them.

[1] "Debugging Myself" by Hayes (1995) contains an entertaining discussion of human bugs.

What is a bug?

Scientists have identified approximately 1 1/4 million species of animals. Of those about 3/4, or 932,000, are insects. However, only the 82,000 species belonging to the order Hemiptera are considered by scientists to be "true bugs" (McGavin, 1993). Fortunately, a taxonomy of SAS bugs would not identify nearly so many species.

Entomology aside, a bug is an error in a computer program that causes an undesirable, usually unexpected, result. One way of classifying computer bugs is to divide them into three types of errors: syntax, data, and logic. Syntax errors result from failing to follow SAS's rules about the way keywords are put together to make statements. With data errors you have a program that is syntactically sound but fails because of data values that do not fit the program as it was written. With logic errors you have a program that runs, and data that fits, but the result is wrong because the program does something different than you intended.

The bugs discussed in this paper can be classified as:

Syntax
- missing semicolon
- uninitialized variable and variable not found

Data
- missing values were generated
- numeric to character conversion
- invalid data
- character field is truncated

Logic
- DATA step produces wrong results but no error message.

Listen to the SAS Log

The first and most important rule in debugging SAS programs is to always, always check the SAS log. After running a SAS program many people turn immediately to the output. This is understandable,

but not advisable. It is entirely possible—and sooner of later it happens to all of us—to get output that looks fine but is totally bogus. Often checking the SAS log is the only way to know whether a program has run properly.

SAS logs contain 3 types of messages: errors, warnings, and notes.

Errors

If you get an error message in your program, you will know it. Error messages get your attention because SAS will not run a job with one of these bugs. Error messages are not quiet, discrete, or subtle; they are the loud, rabble-rousers of SAS messages. This message, for example:

```
ERROR: No CARDS or INFILE statement.
```

stops a program dead in its tracks. This message tells you that SAS could not find any data to read with the INPUT statement.

Warnings

Warnings are less dire than errors. SAS prints warnings in your log and then goes ahead and runs the job anyway. Many people, including some professional programmers, try to ignore warnings. Don't you be one of them. Sometimes the situations that result in warnings are indeed harmless; other times they indicate grave problems which, if unresolved, will render your results worthless. You should check all warnings to see if they are harmless or hazardous. This message:

```
WARNING: The data set WORK.SPECIES may be
incomplete.  When this step was stopped there
were 0 observations and 3 variables.
```

tells you that SAS did run a DATA step, but for some reason there are zero observations. This could be OK, but generally speaking when you go to the trouble of creating a data set, you want some data in it.

Notes

Notes are the most innocuous messages that SAS writes in your SAS log. They simply inform you of the status of your program. Notes contain information such as the number of records input from an external file, or the number of observations written in a SAS data set. Don't be fooled by demure little notes; they are a critically important way of catching errors. These messages:

```
NOTE: 29 records were read from the infile
      'bugspeci.dat'.
      The minimum record length was 27.
      The maximum record length was 27.
NOTE: The data set WORK.SPECIES has 14
      observations and 3 variables.
```

tell you that while 29 records where read from a raw data file, the resulting SAS data set contains only 14 observations. If you were expecting only 14 observations, then this would be fine. But if you were expecting 29 observations, one observation for each input record, then this would tip you off that something went wrong.

Another type of note can help you write efficient programs. At the end of every step SAS prints a note similar to this:

```
NOTE: The PROCEDURE PRINT used 6.98 seconds.
```

If you are running a one-time report, you may not care, but if you run the same program over and over then you may want to check your notes to see which steps can benefit the most from streamlining.

The species data

The data for the next few examples appear in Table 1. Each observation contains data about one order in the class Insecta (La Plante, 1996). The variables are the name of the order (ORDER), the number of species in that order found in North America (NASP), and the number of species found outside North America (OUTSP).

Table 1 Species data.

ORDER	NASP	OUTSP
Thysanura	20	230
Diplura	30	370
Protura	30	70
Collembola	325	1675
Ephemeroptera	550	950
Odonata	425	4575
Plecoptera	34	1266
Grylloblattodea	.	6
Saltatoria	110	21890
Phasmida	.	.
Dictyptera	.	.
Isoptera	45	.
Dermaptera	20	1080
Embioptera	10	140
Psocoptera	150	950
Zoraptera	2	17
Mallophaga	320	2280
Anoplura	65	285
Thysanoptera	625	2375
Hemiptera	8750	46250
Neuroptera	350	4350
Mecoptera	70	280
Trichoptera	950	3550
Lepidoptera	10500	189500
Diptera	16700	68300
Siphonaptera	250	850
Hymenoptera	14600	90400
Coleoptera	27000	530000
Strepsiptera	120	180

The missing semicolon

Even the newest of SAS programmers knows that every SAS statement ends with a semicolon; so it is ironic that one of the most common bugs is the missing semicolon.

While most SAS error messages are clear and easy to understand, the hallmark of a missing semicolon is confusion. Missing semicolons often produce a long stream of baffling messages. In the following example, the absence of a semicolon at the end of the DATA statement causes two error messages, three warnings, and a suspicious note.

```
1   DATA species
2      INFILE 'bugspeci.dat';
             --------------
             200
3      INPUT order $ 1-15 nasp outsp;
4   RUN;

ERROR 200-322: The symbol is not recognized.
ERROR: No CARDS or INFILE statement.
NOTE: The SAS System stopped processing this
    step because of errors.
WARNING: The data set WORK.SPECIES may be
    incomplete.  When this step was stopped
    there were 0 observations and 3 variables.
WARNING: Data set WORK.SPECIES was not
    replaced because this step was stopped.
WARNING: The data set WORK.INFILE may be
    incomplete.  When this step was stopped
    there were 0 observations and 3 variables.
```

The message “No CARDS or INFILE statement” is especially odd since there obviously is an INFILE statement. Without a semicolon the DATA statement becomes concatenated with the INFILE statement. SAS then interprets the keyword INFILE as a data set name in the DATA statement resulting in the warning “data set WORK.INFILE may be incomplete.”

If you find that the messages in your log make no sense, check for missing semicolons.

Uninitialized variable and variable not found

These two related messages tell you that SAS was unable to find one of your variables. The first time you see one of these messages you will probably wonder what SAS is babbling about, after all you remember creating the variable.

In the following SAS log, the INPUT statement reads the species data using the variable name NASP for the number of species in North America. Then a subsetting IF statement contains the misspelled variable name NASPEC.

```
1   DATA species (KEEP = order worldsp);
2      INFILE 'bugspeci.dat';
3      INPUT order $ 1-15 nasp outsp;
4      IF naspec > 100;
5      worldsp = nasp + outsp;
6   RUN;

NOTE: Variable NASPEC is uninitialized.
```

When SAS is unable to find a variable in a DATA step, SAS prints the variable-is-uninitialized message. Then SAS creates the variable, sets its values to missing for all observations, and runs the DATA step. It’s nice that SAS runs the DATA step, but you probably don’t want the variable to have missing values for all observations.

A more serious problem ensues when SAS is unable to find a variable in a PROC step. In the following example, SAS cannot find the variable NASP. This variable did exist, but was accidentally dropped in the previous DATA step because it was not listed in the KEEP option. SAS prints the variable-not-found message and does not run the procedure at all.

```
7   PROC PRINT DATA=species;
8      VAR order nasp worldsp;
ERROR: Variable NASP not found.
9   RUN;
```

Another version of the variable-not-found message appears as a warning when the problem occurs in a less critical statement such as a LABEL statement. Because this is a warning, not an error, SAS runs the step.

Possible causes of the variable-is-uninitialized and variable-not-found messages include:

- A misspelled variable name.
- Using a variable that has been dropped.
- Using the wrong data set.
- A logic error, such as using a variable before it is created.

Missing values were generated

The missing-values-were-generated note tells you that SAS was unable to compute the value of a new variable because of existing missing values in your data. This may not indicate a problem, but it warrants an investigation.

In the following example, SAS computes the variable WORLDSP by adding together NASP and OUTSP.

```
1    DATA species;
2       INFILE 'bugspeci.dat';
3       INPUT order $ 1-15 nasp outsp;
4       worldsp = nasp + outsp;
5    RUN;

NOTE: The infile 'bugspeci.dat' is:
       FILENAME=C:\BUGSPECI.DAT,RECFM=V,LRECL=256

NOTE: 29 records were read from the infile
       'bugspeci.dat'.
       The minimum record length was 27.
       The maximum record length was 27.
NOTE: Missing values were generated as a result
       of performing an operation on missing
       values.
       Each place is given by: (Number of times)
       at (Line):(Column).
       4 at 4:19
NOTE: The data set WORK.SPECIES has 29
       observations and 4 variables.
NOTE: The DATA statement used 5.42 seconds.
```

The missing-values-were-generated note tells you that SAS assigned missing values to four observations at line 4 column 19 of the program. A quick look back at the species data shows that 4 observations have missing data for NASP or OUTSP.

The SUM function and its cousin the MEAN function can lessen this problem because they use only non-missing values. For the preceding program you would use this statement:

```
worldsp = SUM(nasp, outsp);
```

However, if you have an observation with missing values for all of the variables named in the function, then the result is missing too and you will still get the missing-values note for that observation.

The insecticide data

Data about the effectiveness of insecticides appears in Table 2 (based on Conklin, 1996). The variables are the name of the insecticide (INSTCIDE), its residual effect on insects (TOXINSCT), and its toxicity to mammals (TOXMAMML). The toxicity to insects and mammals are both rated on a scale of 0 to 6 where 0 indicates no effect and 6 means extreme toxicity.

Table 2 Insecticide data.

INSTCIDE	TOXINSCT	TOXMAMML
DDT	6	3
Malathion	4	1
Pyrethrins	2	1
Sulfur	6	0
Chlordane	6	4
Diazinon	4	4
Heptachlor	6	5
Nicotine sulfate	1	6

Nicotine sulfate is a lousy insecticide. It has almost no residual effect against insects, but is extremely toxic to people.

Numeric to character conversion

If you accidentally mix numeric and character variables, SAS will convert the data from one type to the other, run the program anyway, and print the values-have-been-converted note.

In the following example, the variables TOXINSCT and TOXMAMML are input as character variables and then used in an arithmetic expression. The note tells you that SAS converted data at columns 10 and 21 in line 4 of the program. These columns correspond to the variable names TOXINSCT and TOXMAMML.

```
1   DATA poisons;
2      INFILE 'bugtox.dat';
3      INPUT instcide $ 1-16 toxinsct $
             18 toxmamml $ 20;
4      dif = toxinsct - toxmamml;
5   RUN;

NOTE: Character values have been converted to
    numeric values at the places given by:
    (Line):(Column).
    4:10   4:21
```

It's nice that SAS tries to fix the problem for you, but this doesn't mean that you can ignore the message. If you let SAS convert your variables, it can come back to haunt you at a later time when the variable that you think is numeric is now character or vice versa. If a variable needs to be converted, you should do it yourself, explicitly, so there are no surprises.

There is a bonus to doing the conversion yourself. Your programs will run faster because it takes less time for SAS to do an explicit conversion than for SAS to figure out how to handle it.

To convert from character to numeric you use the INPUT function. To convert from numeric to character, you use the PUT function. The basic forms of these statements are:

character to numeric:
```
newvar = INPUT(oldvar, informat.);
```

numeric to character:
```
newvar = PUT(oldvar, format.);
```

In either case, the informat or format must be numeric. To convert the TOXINSCT and TOXMAMML in the program above, you could add these statements:

```
newtoxi = INPUT(toxinsct, 1.);
newtoxm = INPUT(toxmamml, 1.);
```

The new variables will have a length of 8 bytes.

Possible causes of the values-have-been converted message include:

- Setting a variable equal to another variable of a different type.
- Using a variable with the wrong type of function.
- Using a character variable in an arithmetic expression.

Invalid data

Once you know how to read the invalid-data message, you'll know exactly what the problem is every time. Whenever SAS encounters invalid data while reading with an INPUT statement, SAS sets the problematic variable to missing for that observation and then prints a detailed message like this:

```
1   DATA invalid;
2      INFILE 'bugtox.dat';
3      INPUT toxname $ 1-15 toxinsct toxmamml;
4   RUN;

NOTE: The infile 'bugtox.dat' is:
      FILENAME=C:\bugtox.dat,RECFM=V,LRECL=256
NOTE: Invalid data for TOXINSCT in line 8 16-16
RULE:     ----+----1----+----2----+----3----+--
8         Nicotine sulfate 1 6
TOXNAME=Nicotine sulfat TOXINSCT=. TOXMAMML=1
    _ERROR_=1 _N_=8
```

The first line of this message is a note telling you which variable had a problem, TOXINSCT, in this case; the line of the raw data file at which the problem occurred, line 8; and the columns SAS was trying to read, column 16. Next SAS prints a line labeled RULE which is a handy ruler for counting columns. On this ruler 1 indicates the 10th column, 2 the 20th, and so on. Then SAS dumps the actual line of raw data so you can see the little troublemaker for yourself. Finally, SAS prints the values of variables as it has read them.

In this case, you can see that column 16 contains the letter "e" as the value for TOXINSCT which is a numeric variable. The problem is that the INPUT statement tells SAS to read TOXNAME from columns 1-15, but it should say 1-16.

Occasionally programmers get invalid-data messages because they are trying to read unprintable or other non-standard characters such as carriage returns. At those times SAS prints two more lines of data labeled ZONE and NUMR. These lines are the hexadecimal representation of the raw data. You don't have to be able to read hexadecimal to be able to interpret this. SAS prints the data this way because the normal 10 numerals and 26 letters don't provide enough values to represent all computer symbols uniquely. Hexadecimal uses two characters to represent each symbol. To read hexadecimal, take a digit from the ZONE line together with the corresponding digit from the NUMR line.

The following invalid-data message is the result of trying to read a Microsoft Word file without first saving it as a text file. Two characters look like periods, but in fact “2E” is a standard period while“11” is some other non-standard character.

```
NOTE: Invalid data for BADDATA in line 1 1-9.
RULE:     ----+----1----+----2----+----3----+---

1   CHAR  65._Ï.à¡±
    ZONE  332DC1EAB
    NUMR  65E0F1011
BADDATA=. _ERROR_=1 _N_=1
```

Possible causes of the invalid-data message include:

- Forgetting to specify that a variable is character (SAS assumes it is numeric).
- Incorrect column specifications producing embedded spaces in numeric data.
- Incorrect column specifications producing character values for a numeric variable.
- List-style data with two periods in a row and no space in between.
- Failing to mark a missing value with a period in list-style input, causing SAS to read the data for the next variable.
- Using the letter O instead of the number zero.
- Special characters such as carriage-return-line-feed and page-feed.
- Invalid dates (such as September 31) read with a date informat.

Character field truncated

This bug does not generate any error messages or suspicious notes, but you know that you have this problem when you print your data and find the end of a character variable has been lopped off.

The length of a character variable is set when SAS first encounters the variable, typically in an INPUT or assignment statement. If you use list-style input, the default length for character variables is eight bytes. With column-style input it is the number of columns you specify. With formatted-style input it is the length of the informat. If you create a new variable with assignment statements, SAS sets its length based on the first occurrence of the variable.

In the following example, the variable TOXICITY is first set equal to “high”. Therefore SAS gives TOXICITY a length of four bytes, and any subsequent longer values will be truncated.

```
1   DATA poisons;
2      INFILE 'bugtox.dat';
3      INPUT instcide $ 1-16 toxinsct toxmamml;
4      IF toxmamml >= 5 THEN toxicity = 'high';
5      ELSE IF toxmamml >= 3 THEN
          toxicity = 'moderate';
6      ELSE IF toxmamml >= 1 THEN
          toxicity = 'low';
7      ELSE toxicity = 'no effect';
8   RUN;
```

Using a PROC PRINT you can see the truncated values for TOXICITY.

The SAS System 1

OBS	INSTCIDE	TOXMAMML	TOXICITY
1	DDT	3	mode
2	Malathion	1	low
3	Pyrethrins	1	low
4	Sulfur	0	no e
5	Chlordane	4	mode
6	Diazinon	4	mode
7	Heptachlor	5	high
8	Nicotine sulfate	6	high

You could fix this problem by padding the value “high” with blanks, but a more elegant and explicit solution is to use the LENGTH statement. Insert this statement in the DATA step before the first occurrence of the variable TOXICITY.

```
LENGTH toxicity $9;
```

The new output is not truncated.

The SAS System 2

OBS	INSTCIDE	TOXMAMML	TOXICITY
1	DDT	3	moderate
2	Malathion	1	low
3	Pyrethrins	1	low
4	Sulfur	0	no effect
5	Chlordane	4	moderate
6	Diazinon	4	moderate
7	Heptachlor	5	high
8	Nicotine sulfate	6	high

Perhaps some future release of SAS will warn you when character values are truncated, but for now you are on your own.

The moth flight data

Data about the flight of 15 individual moths appears in Table 3 (Callahan, 1971). The variables are the moth's species(SPECIES), its family (FAMILY, where n=noctid and s=sphingid), sex (SEX), weight in grams (WEIGHT), and lift in grams at three, six, and twelve degrees of pitch (LIFT3, LIFT6, and LIFT12).

Table 3 Moth flight data.

corn earworm	n	m	0.107	0.118	0.168	0.249
corn earworm	n	m	0.226	0.131	0.186	0.281
corn earworm	n	f	0.161	0.108	0.150	0.232
corn earworm	n	f	0.239	0.154	0.218	0.327
corn earworm	n	f	0.279	0.158	0.222	0.336
fall armyworm	n	m	0.140	0.113	0.159	0.241
fall armyworm	n	m	0.139	0.113	0.159	0.241
fall armyworm	n	f	0.156	0.131	0.186	0.277
white-lined	s	m	0.600	0.322	0.458	0.681
white-lined	s	m	0.322	0.313	0.436	0.654
white-lined	s	f	0.660	0.276	0.386	0.581
white-lined	s	f	0.853	0.336	0.472	0.708
tobacco hornworm	s	f	1.199	0.721	1.017	1.525
tobacco hornworm	s	f	1.604	0.617	0.872	1.307
satellite	s	f	1.726	0.767	1.076	1.616

DATA step produces wrong results but no error

Sometimes a DATA step can seem like a "black box". You know what goes in, and you know what comes out, but what goes on in the middle can be a mystery. If what comes out is not what you want then you have a bug. Problems like this are really logic errors. Somewhere along the way SAS got the wrong instruction—a classic case of the computer doing what you tell it to do, not what you want.

An example

For a moth, a bird, or even a supersonic jet, flight occurs when lift exceeds weight. Using the moth flight data and a series of IF-THEN/ELSE statements, the following program finds the angle of attack at which each moth can sustain flight. The new variable ANGLE equals 3, 6, or 12 depending on the angle at which the moth's lift exceeds its weight.

```
DATA moths;
   INFILE 'bugwing.dat';
   INPUT species $ 1-16 family $ sex $ weight
      lift3 lift6 lift12;
   IF lift3 >= weight THEN angle = 3;
   ELSE IF lift6 >= weight THEN angle = 6;
   ELSE IF lift12 >= weight THEN angle = 12;

PROC PRINT DATA=moths;
    TITLE 'Angle of Attack to Sustain Flight';
    VAR species angle;
RUN;
```

This program runs fine (without errors, warnings, or suspicious notes), but looking at the following output you can see several observations have missing values for ANGLE.

```
Angle of Attack to Sustain Flight     1

OBS    SPECIES                      ANGLE

 1    corn earworm                   3
 2    corn earworm                  12
 3    corn earworm                  12
 4    corn earworm                  12
 5    corn earworm                  12
 6    fall armyworm                  6
 7    fall armyworm                  6
 8    fall armyworm                  6
 9    white-lined                   12
10    white-lined                    6
11    white-lined                    .
12    white-lined                    .
13    tobacco hornworm              12
14    tobacco hornworm               .
15    satellite                      .
```

One way to figure out what went wrong is just to look at the program and the output from PROC PRINT. When that doesn't work, then there are two ways to solve the mystery: the traditional method using PUT statements, and the new DATA step debugger.

Using PUT statements

PUT statements are like INPUT statements in reverse. Instead of reading data, they write it. The basic idea behind using PUT statements to debug a DATA step is to print data values at intermediate points in the DATA step. When used without a FILE statement, PUT statements write values in the log, a handy place for them to be for debugging. The following statement tells SAS to print the values of selected variables for every observation with a missing value for ANGLE.

```
IF angle = . THEN PUT weight= lift3= lift6=
lift12=;
```

After inserting this statement in the program and rerunning it, the log looks like this:

```
1 DATA moths;
2    INFILE 'bugwing.dat';
3    INPUT species $ 1-16 family $ sex $ weight
4       lift3 lift6 lift12;
5    IF lift3 >= weight THEN angle = 3;
6    ELSE IF lift6 >= weight THEN angle = 6;
7    ELSE IF lift12 >= weight THEN angle = 12;
8    IF angle = . THEN PUT weight= lift3=
9       lift6= lift12=;
10 RUN;

NOTE: The infile 'bugwing.dat' is:
      FILENAME=C:\bugwing.dat,RECFM=V,LRECL=256

WEIGHT=0.628 LIFT3=0.236 LIFT6=0.331
LIFT12=0.504
WEIGHT=0.66 LIFT3=0.276 LIFT6=0.386
LIFT12=0.581
WEIGHT=0.853 LIFT3=0.336 LIFT6=0.472
LIFT12=0.708
WEIGHT=1.604 LIFT3=0.617 LIFT6=0.872
LIFT12=1.307
WEIGHT=1.726 LIFT3=0.767 LIFT6=1.076
LIFT12=1.616
NOTE: 15 records were read from the infile
      'bugwing.dat'.
      The minimum record length was 48.
      The maximum record length was 48.
NOTE: The data set WORK.MOTHS has 15
      observations and 9 variables.
NOTE: The DATA statement used 1.86 seconds.
```

Looking at the data values in the log, you can see that lift never exceeds weight for these moths. Apparently, these moths need an angle of attack greater than 12 to get off the ground. The IF-THEN/ELSE series should be rewritten so that it takes into account the possibility that some moths may not sustain flight at 12 degrees.

In this example, the problem was simple enough that you could have solved it by using a PROC PRINT after the DATA step. In real life, the PUT statement technique is most useful when you have long and convoluted DATA steps, especially if that DATA step was written by someone else and you are handed the whole step rather than having the luxury of building it piece by piece.

Using the DATA step debugger

The DATA step debugger offers SAS programmers a new way to investigate logic errors. Available on an "experimental" basis in releases 6.06 to 6.10, the debugger became officially supported with release 6.11.

SAS runs programs in two phases. First SAS compiles your program, then SAS executes your program. Syntax errors and some data errors such as numeric to character conversions occur at compile time. Other errors such as logic errors and some data errors compile just fine, but cause you to get bad results. These are the errors that the DATA step debugger can help identify.

Space limits do not allow for a detailed discussion of the DATA step debugger, but the information here should be enough to get you started. For more information see *SAS Software: Changes and Enhancements, Release 6.11*.

To invoke the debugger, add "/ DEBUG" to the end of your DATA statement. Then run the DATA step in display manager. For the preceding example you would submit this.

```
DATA moths / DEBUG;
   INFILE 'bugwing.dat';
   INPUT species $ 1-16 family $ sex $ weight
      lift3 lift6 lift12;
   IF lift3 >= weight THEN angle = 3;
   ELSE IF lift6 >= weight THEN angle = 6;
   ELSE IF lift12 >= weight THEN angle = 12;
RUN;
```

After you submit the DATA step, two windows will appear. These are the DEBUGGER LOG window and the DEBUGGER SOURCE window. The DEBUGGER LOG window contains messages from the debugger and a command line. The SOURCE window contains your DATA step statements with the current line highlighted. By watching the highlighting move, you can see how SAS executes your program. SAS executes each-line of your program for the first observation, then returns to the top of the DATA step for the second observation, and so on.

Figure 1 DATA Step Debugger screen.

```
                DEBUGGER LOG
DATA STEP Source Level Debugger
Stopped at line 2 column 1


> STEP
```

```
                DEBUGGER SOURCE
1 DATA moths / DEBUG;
2    INFILE 'bugwing.dat';
3    INPUT species $ 1-6 family $ sex $ weight
lift3 lift6 lift12;
4    IF lift3 >= weight THEN angle = 3;
5    ELSE IF lift6 >= weight THEN angle = 6;
6    ELSE IF lift12 >= weight THEN ANGLE = 12;
```

By using commands you can control how many lines SAS executes, and you can print the current values of variables you specify. Some of the basic commands that you can issue appear in the following table.

Commands	
EXAMINE *variable-list*	Prints the values of specified variables. Must specify variable names or _ALL_.
STEP	Executes one statement.
<return>	Executes one statement.
SET *variable = value*	Assigns a new value to a specified variable.
BREAK *linenumber*	Tells SAS to execute statements up to the line number specified. Use the GO command to begin execution.
GO	Starts or resumes execution of the DATA step
QUIT	Ends a debugger session.

To get a feel for the debugger, you may want to start by stepping through your DATA step line by line. When you want to know the current values of variables, issue an EXAMINE command such as

```
EXAMINE _ALL_
```

for all variables, or

```
EXAMINE weight lift3 lift6 lift12 angle
```

to choose specific variables. To end your debugging session enter the command

```
QUIT
```

DATA step debugger vs. PUT statements

Some programmers will probably find the DATA step debugger very useful, others may choose to stay with the traditional PUT statement method.

The debugger is designed to be used in display manager, so people who normally use SAS in batch will probably prefer the PUT statement method. The debugger can work in batch in some environments (by popping you into an interactive window), but it makes more sense for people who work interactively.

Since the DATA step debugger is more interactive, it is better suited to an exploratory approach, printing a few data values here and there, making decisions as you go. If you have a general idea of which part of your DATA step is causing the problem, then you may find it simpler to use PUT statements.

Some logic errors may be easier to debug by looking at more than one observation at a time. In those cases the observation-by-observation nature of the debugger may give less insight than the PUT statement method.

One nice bonus of the DATA step debugger is the ability to watch SAS execute a DATA step line-by-line and observation-by-observation. For a beginner, this alone could be very enlightening.

CONCLUSIONS

This paper has discussed some of the most common SAS programming bugs and how to exterminate them. For discussions of other SAS bugs see Delwiche and Slaughter (1995) and Carpenter (1996).

You should always check your SAS log even when the output looks fine. Notes are just as important as error messages and warnings in debugging your programs. Once you understand why you got an error, you'll be better able to avoid it in the future.

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

References

Callahan, Philip S. (1971*). Insects and How They Function*. Holiday House, NY.

Carpenter, Arthur L (1996). Programming for Job Security: Tips and Techniques to Maximize Your Indispensability. *Proceedings of the Twenty-First Annual SAS Users Group International Conference*, 19, pp. 1637-1640.

Conklin, Gladys (1996). Insects. Encyclopedia Americana, International Edition (1996). Grolier, Danbury, CT, vol. 15, pp. 197-208.

Delwiche, Lora D. and Susan J. Slaughter (1995). *The Little SAS Book: A Primer*. SAS Institute, Cary, NC.

Hayes, Brian (1995). Debugging Myself. *American Scientist*, 83, pp. 404-408.

LaPlante, Albert A. (1996). Insect Control. Encyclopedia Americana, International Edition (1996). Grolier, Danbury, CT, vol. 15, pp. 197-208.

McGavin, George C. (1993). *Bugs of the World*. Facts on File, Inc., New York.

SAS Software: Changes and Enhancements, Release 6.11 (1995). SAS Institute, Cary, NC.

About the Authors

Lora Delwiche and Susan Slaughter are also the authors of *The Little SAS Book: A Primer* published by SAS Institute, and may be contacted at:

Susan J. Slaughter (916) 756-8434
slslaughter@ucdavis.edu

Lora D. Delwiche (916) 752-6285
lddelwiche@ucdavis.edu