

Debugging 101

Peter Knapp

U.S. Department of Commerce

Overview

The aim of this paper is to show a beginning user of SAS how to debug SAS programs. New users often review their logs only for syntax errors that appear in red. They neglect to look for other types of coding errors. SAS identifies non-syntax errors in notes and warnings. Examples include notes that SAS has found uninitialized variables, generated missing values, or encountered more than one DATA set with repeats of BY values in a MERGE statement. In addition, once all coding errors are cleaned up, new users can find that their programs do not produce the desired results. To produce the desired results they need to uncover logic errors that are often more difficult to find than coding errors. By demonstrating various debugging techniques, I plan to show that with a little practice, one can master the art of debugging SAS programs.

Understanding How SAS Runs a Program

Before discussing programming errors, having a basic understanding of how the SAS System runs a program is important. A SAS program is typically made up of DATA steps and PROC steps. DATA steps create SAS data sets that read and modify data. PROC steps use SAS data sets and perform specific actions with the data.

The component of SAS that runs programs is called the SAS supervisor. It first checks the syntax of a program by reading a step and checking the step for syntax errors. If no errors are encountered, the supervisor compiles the step and runs data through the compiled code before moving on to the next step and repeating the process.

For example to print out a list of home market sales, a DATA step would read in the data and a PROC PRINT would print the list. The log of the program HM SALES looks like this:

```
1  *** CREATE HOME MARKET SALES ***;
2
3  DATA HMSALES;
4  INPUT CONNUMH $ 1-2 GRSUPRH QTYH PACKH
5      @11 SALEDTH DATE9.;
6  FORMAT SALEDTH DATE9.;
7  LIST;
8  CARDS;

RULE:-----1-----2-----3-----4-----5-----6-----7-----8
9  01 23 7 4 07OCT1998
10 02 17 5 2 06OCT1998
11 03 52 2 8 07OCT1998
12 03 62 5 2 05OCT1998
```

NOTE: The data set WORK.HMSALES has 4 observations and 5 variables.
NOTE: The DATA statement used 0.05 seconds.

```
13 RUN;
14
15 /* PRINT HOME MARKET SALES */
16
17 PROC PRINT DATA = HMSALES;
18 TITLE "Home Market Sales Data";
19 RUN;
```

NOTE: The PROCEDURE PRINT used 0.02 seconds.

The output of the run looks like this:

Home Market Sales Data					
OBS	CONNUMH	GRSUPRH	QTYH	PACKH	SALEDTH
1	01	23	7	4	07OCT1998
2	02	17	5	2	06OCT1998
3	03	52	2	8	07OCT1998
4	03	62	5	2	05OCT1998

In this example, the SAS supervisor reads lines of SAS code until it sees the key word RUN and knows it is at the end of a step. Because there are no syntax errors, SAS compiles the DATA step, the code is executed, and the data set HMSALES is created. It has four observations and two variables.

The supervisor then reads additional lines of code until it sees the second RUN statement. The PROC PRINT is written correctly so SAS compiles and executes the procedure. Finally, the job ends as there are no more lines of code to execute.

Different Kind of Errors

As already discussed, SAS can produce syntax errors while it compiles a step. Syntax errors relate directly to the rules that govern how SAS code is written. For example, all variable names in SAS can be no longer than eight characters in length and must start with a letter or underscore. Trying to create a variable HMQUANTITY will cause a syntax error because the variable has a length of ten.

If the submitted step is free of syntax errors, SAS then executes the step and runs data through it. Depending on what data is read, SAS may generate execution-time errors. Execution-time errors occur as SAS is running the step because of the way the code has been written. Generally

speaking the error occurs because of the way variables are defined as opposed to actual data values.

For example, using the variable REBATEH (that has not been previously defined) in an equation will produce an uninitialized variable message. Syntactically, there is nothing wrong with the variable, as it follows the naming convention of variables. But because it has not been previously defined, SAS assigns a value of missing to the variable. If the variable had been explicitly defined, SAS would not define it as an uninitialized variable.

Another class of errors, that are actually a subset of execution-time errors, are invalid data errors. Invalid data errors only occur if the data running through the step causes SAS to produce an execution-time error.

For example, if GRSUPRH is used in the calculation of NETPRIH and the value of GRSUPRH is missing for some observations, the value of NETPRIH for those observations will be set to missing. NETPRIH will be non-missing for the other observations.

All three kinds of errors can prevent your program from working. To make sure your program is working properly, reviewing the log is very important. SAS logs, in addition to printing out the submitted code, contain three kinds of messages: errors, notes, and warnings. Depending on the kind of error SAS finds, it will print out some combination of the three kinds of messages. Reading all of the messages in the log is important, not just the error messages displayed in red.

A fourth class of errors which can be hard to debug are logic errors. Sometimes programs do not produce the desired results after all syntax and execution-time errors have been cleaned up. These errors can occur because the program was not designed properly. It is important to understand the data that will be used by the program and to account for all possible data values when designing and writing the program.

For example, a program may be written to produce mailing labels using a membership database. If the database contains domestic and international addresses, but the program is only written with domestic addresses in mind, there's a good chance that labels for international members will not print properly.

A. Syntax Errors

If SAS detects a syntax error, it usually underlines the error (or where it thinks the error is), prints a number below the underline, and prints that number along with a message at the bottom of the step. The supervisor then enters syntax

check mode. It continues to read statements, check their syntax, and underline any additional errors. SAS will run data through additional steps depending on where the data is being read from and whether the step is a DATA or PROC step.

Missing Semicolons

If in the HM SALES program the semicolon in the DATA statement is left out, SAS produces the following log

```
1 *** CREATE HOME MARKET SALES ***;
2
3 DATA HMSALES
4 INPUT CONNUMH $ 1-2 GRSUPRH QTYH PACKH
      -
      200
      -
      76
5          @11 SALEDTH DATE9.;
6 FORMAT SALEDTH DATE9.;
7 LIST;
9 CARDS;
```

ERROR 200-322: The symbol is not recognized.

ERROR 76-322: Syntax error, statement will be ignored.

NOTE: The SAS System stopped processing this step because of errors.
NOTE: The DATA statement used 0.78 seconds.

```
10 RUN;
```

```
11
12 /* PRINT HOME MARKET SALES */
13
14 PROC PRINT DATA = HMSALES;
ERROR: File WORK.HMSALES.DATA does not exist.
15 TITLE "Home Market Sales Data";
16 RUN;
```

NOTE: The SAS System stopped processing this step because of errors.
NOTE: The PROCEDURE PRINT used 0.08 seconds.

Without the semicolon SAS reads the INPUT statement as part of the data statement and thinks that besides the data set HMSALES the step is trying to create the data sets INPUT and CONNUMH. The \$ is not a valid data set name in SAS so the supervisor generates an error message and stops executing the DATA step. The PROC PRINT executes, but because the data set HMSALES was not created in the DATA step, SAS generates an error message and stops execution of the step.

Forgetting to include a semicolon is a very common programming mistake. Remember that all SAS statements end in semicolons. Be careful that a colon is not used instead of a semicolon.

In the HM SALES program, if the semicolon is left off of the end of the first comment, SAS would read the DATA

statement as part of the comment and complain that the input statement (and every other statement in the DATA step) is not valid or it is used out of order.

```

1 *** CREATE HOME MARKET SALES ***
2
3 DATA HMSALES;
4 INPUT CONNUMH $ 1-2 GRSUPRH QTYH PACKH
   -----
   180
5           @11 SALEDTH DATE9.;

```

ERROR 180-322: Statement is not valid or it is used out of proper order.

(lines 6 through 14 deleted)

```

15
16 /* PRINT HOME MARKET SALES */
17
18 PROC PRINT DATA = HMSALES;
ERROR: File WORK.HMSALES.DATA does not exist.
19 TITLE "Home Market Sales Data";
20 RUN;

```

NOTE: The SAS System stopped processing this step because of errors.
NOTE: The PROCEDURE PRINT used 0.08 seconds.

The Program Will Not Stop

Besides a SAS comment statement that begins with a * and ends with a ; there are comments that begin with a /* and end with a */. If the */ is left off in the HM SALES program, something strange happens

```

1 *** CREATE HOME MARKET SALES ***;
2
3 DATA HMSALES;
4 INPUT CONNUMH $ 1-2 GRSUPRH QTYH PACKH
5           @11 SALEDTH DATE9.;
6 FORMAT SALEDTH DATE9.;
7 LIST;
8 CARDS;
RULE:-----1-----2-----3-----4-----5-----6-----7-----8
9 01 23 7 4 07OCT1998
10 02 17 5 2 06OCT1998
11 03 52 2 8 07OCT1998
12 03 62 5 2 05OCT1998

```

NOTE: The data set WORK.HMSALES has 4 observations and 5 variables.
NOTE: The DATA statement used 1.46 seconds.

```

13 RUN;
14
15 /* PRINT HOME MARKET SALES
16
17 PROC PRINT DATA = HMSALES;
18 TITLE "Home Market Sales Data";
19 RUN;

```

Not only will the PROC PRINT not run (no messages are printed after the PROC PRINT and no output is produced) but SAS continues to run though there is no more code to compile and execute. The last statement in a SAS program

must be a RUN statement. Because the RUN statement was read as part of the comment, SAS is patiently waiting for more code to run. To fix this problem, submit

```
*/ RUN;
```

This will close the comment and give SAS the RUN statement it needs to finish running the program. Similarly, if a semicolon is left off the step proceeding the RUN statement, the RUN statement will be treated as part of the previous statement and the program will not finish. The solution is to submit

```
; RUN;
```

Misspelled or Missing Keywords

To illustrate the next syntax error, the HM SALES program is modified to read an external file.

```

1 FILENAME MYDATA 'D:\SDSIRAWDATA';
2
3 *** CREATE HOME MARKET SALES ***;
4
5 DATA HMSALES;
6 XFILE MYDATA;
   -----
   180
7 INPUT CONNUMH $ 1-2 GRSUPRH QTYH PACKH
8           @11 SALEDTH DATE.;
9 RUN;

```

ERROR 180-322: Statement is not valid or it is used out of proper order.
ERROR: No CARDS or INFILE statement.
NOTE: The SAS System stopped processing this step because of errors.
WARNING: The data set WORK.HMSALES may be incomplete. When this step was stopped there were 0 observations and 5 variables.
NOTE: The DATA statement used 0.05 seconds.

Because the keyword INFILE is misspelled as XFILE SAS does not know how to interpret the statement and an error is generated. The INPUT statement requires a CARDS or INFILE statement. As far as SAS is concerned, it cannot find either statement, so another error is generated. Correcting the keyword will fix the DATA step.

Coding Statements in the Wrong Place

Take a look at the following program:

```

1 *** CALCULATE THE AVERAGE TOTAL PRICE ***;
2
3 PROC MEANS DATA = HMSALES;
4 TOTPRICE = QTYH * GRSUPRH;
   -----
   180
5 VAR TOTPRICE;
ERROR: Variable TOTPRICE not found.
6 RUN;

```

ERROR 180-322: Statement is not valid or it is used out of proper order.

NOTE: The SAS System stopped processing this step because of errors.

NOTE: The PROCEDURE MEANS used 0.19 seconds.

SAS produces an error in the log because it doesn't like the assignment statement that calculates TOTPRICE. While the assignment statement is syntactically correct, it is not allowed in the PROC MEANS. Assignment statements belong in DATA steps. An improved program would look like this:

```
1 *** CALCULATE THE AVERAGE TOTAL PRICE ***;
2
3 DATA HMSALES;
4 SET HMSALES;
5 TOTPRICE = QTYH * GRSUPRH;
6 RUN;
```

NOTE: The data set WORK.HMSALES has 4 observations and 6 variables.

NOTE: The DATA statement used 0.07 seconds.

```
7
8 PROC MEANS DATA = HMSALES;
9 VAR TOTPRICE;
10 RUN;
```

NOTE: The PROCEDURE MEANS used 0.02 seconds.

The output looks like this:

Home Market Sales Data

Analysis Variable : TOTPRICE

N	Mean	Std Dev	Minimum	Maximum
4	165.0000000	101.9182679	85.0000000	310.0000000

Quickly Checking for Syntax Errors

Because the SAS supervisor compiles and runs SAS programs one step at a time, a syntax error may not be found until the last step of the program. If large data sets are used to test the program a system option

```
OPTION OBS = 0;
```

can be coded at the beginning of the program. This allows the program to be run in syntax check mode. Every step will be checked for syntax errors and compiled, but no data will be run. This speeds up the debugging process by eliminating the wait time for data to run through the program.

It is worth mentioning that occasionally, setting the OBS to zero will cause syntax errors to be generated. For example, formats can be created using the PROC FORMAT with a CNTLIN option. This option tells SAS to use the data set specified in the CNTLIN to generate the format. As no data

is running through the program, the input data set has no observations and the PROC FORMAT fails. This error goes away when the input data set has observations.

B. Execution-Time Errors

When SAS runs data through a compiled step, it can encounter execution-time errors. These kinds of errors, depending on their severity, either generate notes in the log and allow the program to continue running, or generate error messages.

Uninitialized Variables

The following program creates a subset of 1998 Home Market (HM) Sales:

```
1 *** KEEP 1998 SALES ***;
2
3 DATA HMSALES;
4 SET HMSALES;
5 IF YEAR(SALEDATE) = 1998;
6 RUN;
```

NOTE: Variable SALEDATE is uninitialized.

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 160:7

NOTE: The data set WORK.HMSALES has 0 observations and 7 variables.

NOTE: The DATA statement used 0.15 seconds.

and generates the note about SALEDATE being uninitialized. The message SAS writes to the log is a note as opposed to an error because the variable SALEDATE is a legitimate name for a SAS variable. It doesn't know that the variable is misspelled. Because SALEDATE is uninitialized, SAS sets its value to missing for all observations. Missing is never equal to 1998 so the data set HMSALES has no observations output to it.

Uninitialized variables occur for many reasons. Things to look for are dropping the variable from the input data set, misspelling the variable name, using the variable before it is created, or using the wrong data set.

Another thing to watch for is accidentally creating an uninitialized variable by assigning a non-existent variable to itself.

For example the following program calculates a net price by adding any packing to gross unit price.

```
1 *** CALCULATE THE NET PRICE ***;
2
3 DATA HMSALES;
4 INFILE MYDATA;
5 INPUT CONNUMH $ 1-2 GRSUPRH;
6 PACKH = PACKH;
```

```
7 RUN;NOTE: The infile MYDATA is:
  FILENAME=D:\SDS\RAWDATA,
  RECFM=V,LRECL=256
NOTE: 4 records were read from the infile MYDATA.
  The minimum record length was 19.
  The maximum record length was 19.
```

```
NOTE: The data set WORK.HMSALES has 4 observations and 3 variables.
NOTE: The DATA statement used 0.08 seconds.
```

```
8
9 DATA HMSALES;
10 SET HMSALES;
11 PACKH = PACKH;
12 NETPRIH = GRSUPRH + PACKH;
13 RUN;
```

```
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 12:22
```

```
NOTE: The data set WORK.HMSALES has 4 observations and 4 variables.
NOTE: The DATA statement used 0.07 seconds.
```

```
14
15 PROC PRINT DATA = HMSALES;
16 TITLE "CALCULATION OF NET PRICE";
17 RUN;
```

```
NOTE: The PROCEDURE PRINT used 0.01 seconds.
```

While PACKH is not in the INPUT statement and therefore does not exist (line 11), no uninitialized note is produced in the first step. It is only when the program tries to use PACKH in a calculation (line 12), that SAS produces the uninitialized note.

Missing Values

In the previous example, missing values are generated in the DATA step that calculates NETPRIH. The note immediately following the note that indicates that "missing values were generated as a result of performing an operation on missing values" explains how many times the missing values were generated, at what line they were generated, and at what column they were generated.

In this example missing values are generated for all observations in the data set. This information, coupled with the uninitialized note, clearly points to the conclusion that a variable used in the equation is somehow invalid.

If PACKH had originally been read in, it would not be uninitialized. If missing values are being generated for only a subset of the DATA set, a likely explanation would be that there are missing values for PACKH in the input data set.

Having missing values for some sales may be okay, but the result of using missing values in an arithmetic expression is that the result will be set to missing. One of the rules of SAS is that missing values propagate themselves. Every NETPRIH that is calculated using a missing PACKH will

be missing. To avoid this, the SUM function can be used. It returns the sum of all if its non-missing arguments. It is not recommended to use the SUM function unless the cause of missing values is known and deemed appropriate. In the calculation of NETPRIH example, if the SUM function is used to eliminate the missing values note, NETPRIH will be equal only to GRSUPRH, which is not the desired result.

Numeric and Character Conversions

In the following example a U.S. sales data set is created and an attempt is made to merge the U.S. and HM sales data sets together.

```
1 *** CREATE U.S. SALES ***;
2
3 DATA USSALES;
4 INPUT CONNUMU GRSUPRU QTYU PACKU SALEDTU;
5 LIST;
6 CARDS;
RULE:-----1-----2-----3-----4-----5-----6-----7-----8
7 1 26 3 2 19981005
8 2 72 2 7 19981006
9 3 13 6 8 19981007
10 4 0 8 4 19981007
```

```
NOTE: The data set WORK.USSALES has 4 observations and 5 variables.
NOTE: The DATA statement used 0.05 seconds.
```

```
11 RUN;
12
13 DATA COMBINE1;
14 MERGE USSALES (IN = INUS)
15       HMSALES (IN = INHM
16             RENAME = (CONNUMH = CONNUMU));
ERROR: Variable CONNUMU has been defined as both character and
      numeric.
17 BY CONNUMU;
18 IF INUS AND INHM;
19 RUN;
```

```
NOTE: The SAS System stopped processing this step because of errors.
WARNING: The data set WORK.COMBINE1 may be incomplete. When this
step was stopped there were 0 observations and 8 variables.
WARNING: Data set WORK.COMBINE1 was not replaced because this step
was stopped.
```

```
NOTE: The DATA statement used 0.05 seconds.
```

SAS generates a syntax error because the BY variable, CONNUMU, has been defined as both character and numeric. The DATA steps that create both data sets are perfectly legitimate. But because the CONNUMs are of different types, the merge will not work. To fix this problem one CONNUM's type needs to be converted. The first example uses a PUT function to convert numeric data to character data. The new value is assigned to a temporary CONNUM and the combination of a DROP = and a RENAME = option in the DATA statement drops the original numeric CONNUMU and renames the character TEMPCON as a character CONNUMU.

```

1 * CONVERT CONNUMU TO A CHARACTER VARIABLE *;
2
3 DATA USSALES2 (DROP = CONNUMU
4     RENAME = (TEMPCON = CONNUMU));
5     SET USSALES;
6     TEMPCON = PUT(CONNUMU, 2.);
7     TEMPCON = '0' || (LEFT(TEMPCON));
8 RUN;

```

NOTE: The data set WORK.USSALES2 has 4 observations and 5 variables.
NOTE: The DATA statement used 0.1 seconds.

```

9
10 DATA COMBINE2;
11 MERGE USSALES2 (IN = INUS)
12     HMSALES (IN = INHM
13     RENAME = (CONNUMH = CONNUMU));
14 BY CONNUMU;
15 IF INUS AND INHM;
16 RUN;

```

NOTE: The data set WORK.COMBINE2 has 4 observations and 8 variables.
NOTE: The DATA statement used 0.1 seconds.

Note that a '0' needs to be added to the value of CONNUMU so that the merge will work properly. Also, because character data is right justified, the value of TEMPCON must be left justified before it is concatenated to the '0'. Otherwise the concatenated value would look like '0 1' and the third digit '1' would be truncated as TEMPCON has a length of two.

To convert character data to numeric data, use a PUT function instead of an INPUT function.

```
TEMPCON = PUT (CONNUMH, 8.);
```

The following example illustrates two other possible causes of values being converted by SAS:

```

1 *** CALCULATE THE NET PRICE ***;
2
3 DATA HMSALES;
4 INFILE MYDATA;
5 INPUT CONNUMH $ 1-2 GRSUPRH;
6 TOTAL = CONNUMH + GRSUPRH;
7 GRSUPRH = TRIM(GRSUPRH);
8 RUN;

```

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

6:12 7:14

NOTE: Numeric values have been converted to character values at the
places given by: (Line):(Column).

7:19

NOTE: The infile MYDATA is:
FILENAME=D:\SDSIRAWDATA,
RECFM=V,LRECL=256

NOTE: 4 records were read from the infile MYDATA.
The minimum record length was 19.
The maximum record length was 19.

NOTE: The data set WORK.HMSALES has 4 observations and 3 variables.
NOTE: The DATA statement used 0.08 seconds.

The variable CONNUMH is a character variable and is being used in an arithmetic expression so the value of CONNUMH is converted to a numeric value. GRSUPRH is numeric and the TRIM function is expecting a character argument so the value of GRSUPRH is converted to a character value, trimmed, and converted back to a numeric value that is assigned to GRSUPRH.

While SAS will convert values automatically, it is recommended that the program explicitly convert values. SAS runs more efficiently if it does not have to figure out how to convert data values. It is also safer. Converting character data to numeric data can truncate leading zeros from the data values. Converting numeric data to character data will not automatically pad the converted value with a zero. Automatic conversion may lead to unexpected results.

C. Invalid Data Errors

Invalid data errors occur when the raw data SAS is trying to read in does not match the way SAS is trying to read the data. In the following example, the INPUT statement is trying to read in four numeric variables and one date variable:

```

1 *** CREATE HOME MARKET SALES ***;
2
3 DATA HMSALES;
4 INPUT CONNUMH 1-6 GRSUPRH QTYH PACKH
5     @14 SALEDTH DATE9.;
6 FORMAT SALEDTH DATE9.;
7 LIST;
8 CARDS;

```

NOTE: Invalid data for CONNUMH in line 9 1-6.

NOTE: Invalid data for PACKH in line 9 11-19.

NOTE: Invalid data for SALEDTH in line 9 14-22.

RULE:-----1-----2-----3-----4-----5-----6-----7-----8

9 01 23 7 4 07OCT1998

CONNUMH=. GRSUPRH=7 QTYH=4 PACKH=. SALEDTH=. _ERROR_=1
N=1

NOTE: The data set WORK.HMSALES has 1 observations and 5 variables.
NOTE: The DATA statement used 0.11 seconds.

```
10 RUN;
```

SAS prints the contents of the input buffer in the log under a line labeled RULE. It also prints out the values assigned to the variables. This information can be used to determine why the data was not read in properly.

In this example, CONNUMH should have been read in using columns 1-2, not columns 1-6. Columns 1-6 contain the string '01 23 ' which is not a valid number. GRSUPRH and QTYH are read in between columns 7-9, though because the real value of GRSUPRH is in columns 4-5, the wrong data values are read in for GRSUPRH and

QTYH. By the time PACKH is read in, SAS is looking at the date value. The date value is not a valid number. Finally, the program tells SAS to read in SALEDTH. The date value really begins at column 11, so the reading of a partial date value is also invalid. This example illustrates that it is important to understand the data used by the program.

Character Field Truncation

Unlike computer languages that require the explicit definition of variables at the top of the program, SAS determines the attributes of a variable by the context in which it is first used. This can cause truncation problems with character variables.

The following program creates a variable COST that is intended to indicate if a GRSUPRH is high.

```
1  *** CREATE WORD DAY ***;
2
3  DATA HMSALES;
4  SET HMSALES;
5  IF GRSUPRH < 50 THEN
6    COST = 'LOW';
7  ELSE
8  IF GRSUPRH >= 50 THEN
9    COST = 'HIGH';
10 RUN;
```

NOTE: The data set WORK.HMSALES has 4 observations and 6 variables.
NOTE: The DATA statement used 0.07 seconds.

```
11
12 PROC PRINT DATA = HMSALES;
13 TITLE "Home Market Sales Cost";
14 VAR GRSUPRH COST;
15 RUN;
```

NOTE: The PROCEDURE PRINT used 0.02 seconds.

The output looks like this:

Home Market Sales Cost		
OBS	GRSUPRH	COST
1	23	LOW
2	17	LOW
3	52	HIG
4	62	HIG

Note that the value of COST in the output is either 'LOW' or 'HIG'. This is because SAS first encounters the variable COST in the first assignment statement. The value that is being assigned to COST is character and has a length of three. Even though the second assignment statement tries to assign a character value with a length of four, the attributes of the variable COST have already been set.

To prevent character values from being truncated, code a LENGTH or ATTRIB statement at the top of the DATA

step to explicitly define the length of the character variable. Here are examples of each:

```
LENGTH COST $ 4;
ATTRIB COST LENGTH = $4;
```

Illegal Mathematical Operations

Occasionally, certain data values will cause a SAS program to fail because it has performed an illegal mathematical operation. The following program tries to determine what percentage of the gross unit price is made up of packing expenses.

```
1 DATA USSALES;
2 SET USSALES;
3 PERCNT = (GRSUPRU - PACKU) / GRSUPRU * 100;
4 RUN;
NOTE: Division by zero detected at line 3 column 31.
CONNUMU=4 GRSUPRU=0 QTYU=8 PACKU=4 SALEDTU=19981007
PERCNT= . _ERROR_=1 _N_=4
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).
1 at 3:41
NOTE: Mathematical operations could not be performed at the following
places. The results of the operations have been set to missing values.
Each place is given by: (Number of times) at (Line):(Column).
1 at 3:31
```

NOTE: The data set WORK.USSALES has 4 observations and 6 variables.
NOTE: The DATA statement used 0.08 seconds.

SAS generates an error because GRSUPRU is equal to zero in the fourth sale. An easy way to revise the program, assuming that zero is a valid value for GRSUPRU, is to check the value of GRSUPRU before calculating PERCENT.

```
1 DATA USSALES;
2 SET USSALES;
3 IF GRSUPRU = 0 THEN
4 PERCNT = 0;
5 ELSE
6 PERCNT = (GRSUPRU - PACKU) / GRSUPRU * 100;
8 RUN;
```

NOTE: The data set WORK.USSALES has 4 observations and 7 variables.
NOTE: The DATA statement used 0.07 seconds.

If SAS generates an error indicating it has performed an illegal mathematical operation, verify that the values causing the illegal mathematical operation are valid. If the data is valid add condition processing to conditionally perform the operation. If the data is invalid, correct the data.

By Group Processing

If data has been sorted by a certain key variable, subsequent processing can take advantage of the fact that

data is organized in unique groupings. For example it allows two data sets to be merged together by the sorted variable. The following program tries to print out HM sales by SALEDTH:

```
1 PROC PRINT DATA = HMSALES;
2 BY SALEDTH;
3 TITLE "HM Sales by Sale Date";
4 RUN;
```

ERROR: Data set WORK.HMSALES is not sorted in ascending sequence.
The current by-group has SALEDTH = 07OCT1998 and the next by-group
Has SALEDTH = 06OCT1998.

NOTE: The SAS System stopped processing this step because of errors.
NOTE: The PROCEDURE PRINT used 0.02 seconds.

SAS cannot print out the data because it has not previously been sorted by sale date. The solution is to sort the data set before printing it.

```
1 PROC SORT DATA = HMSALES OUT = HMSALES;
2 BY SALEDTH;
3 RUN;
```

NOTE: The data set WORK.HMSALES has 4 observations and 6 variables.
NOTE: The PROCEDURE SORT used 0.14 seconds.

```
4
5 PROC PRINT DATA = HMSALES;
6 BY SALEDTH;
7 TITLE "HM Sales by Sale Date";
8 RUN;
```

NOTE: The PROCEDURE PRINT used 0.02 seconds.

The output look like this:

HM Sales by Sale Date				
----- SALEDTH=05OCT1998 -----				
OBS	CONNUMH	GRSUPRH	QTYH	PACKH
1	03	62	5	2
----- SALEDTH=06OCT1998 -----				
OBS	CONNUMH	GRSUPRH	QTYH	PACKH
2	02	17	5	2
----- SALEDTH=07OCT1998 -----				
OBS	CONNUMH	GRSUPRH	QTYH	PACKH
3	01	23	7	4
4	03	52	2	8

Many people assume that the data must be explicitly sorted by SAS to take advantage of BY processing. If the data is read in already sorted, BY processing will work without needing to sort the data. It is safer to sort the data though as it is not always possible to know ahead of time if the data is already sorted as it is being read in.

D. Logic Errors

Sometimes after all syntax errors have been cleaned up and no warnings or notes indicate there is anything wrong with the program, the results of the program are still wrong. For example, the following program is designed to add home market sales information to each observation in the U.S. sales data set:

```
1 DATA MATCH;
2 MERGE USSALES2 (IN = INUS)
3 HMSALES (IN = INHM
4 RENAME = (CONNUMH = CONNUMU));
5 BY CONNUMU;
6 RUN;
```

NOTE: The data set WORK.MATCH has 5 observations and 10 variables.
NOTE: The DATA statement used 0.1 seconds.

Everything looks fine at a quick glance. The program is not producing the desired results though. There are four U.S. sales input into the DATA step and five combined observations being output. This is because there are two HM sales with a CONNUMH of '03'. SAS merges the U.S. sale with a CONNUMU of '03' to both HM sales. To rectify this problem, the duplicate values in the HM data set could be eliminated by sorting the HM data set with a NODUPKEY option before the merge.

It is worth noting the way SAS merges data sets that have repeats of BY variables. If in the above example, USSALES2 had three observations with a value of '03' for CONNUMU the log would look like this:

```
1 DATA MATCH;
2 MERGE USSALES2 (IN = INUS)
3 HMSALES (IN = INHM
4 RENAME = (CONNUMH = CONNUMU));
5 BY CONNUMU;
6 RUN;
```

NOTE: MERGE statement has more than one data set with repeats of BY values.

NOTE: The data set WORK.MATCH has 8 observations and 10 variables.
NOTE: The DATA statement used 0.2 seconds.

When SAS encounters more than one occurrence of a BY group value in both data sets, it performs a Cartesian join. It matches every observation of the BY group in the first data set with every observation of the BY group in the second data set. While this is not an error, as with the previous example, the desired results were not produced.

Diagnosing Logic Errors

There are several techniques for diagnosing logic errors. The best technique is to prevent them from happening. Key to preventing logic errors is understanding the requirements of the program (what it is supposed to do) and knowing

what the data looks like. In the above example, the program is supposed to add HM sales to U.S. sales. Instead it joins both kinds of information together, duplicating one of the U.S. sales in the process.

The incorrect assumption made while writing the program is that there is only one sale for each CONNUM in each market. An examination of the data would reveal that repeats of CONNUMs can occur.

The data can be examined by printing out the data set or printing out a subset of the data. The Libraries Window can be used to view the data in a spreadsheet view. The SAS Data Set Viewer, a utility that can be downloaded from the SAS Institute web page (www.sas.com) also allows for viewing of data in a spreadsheet view.

Once a program is written and checked for errors, if the desired results are not achieved, the tools can help diagnose why the program is not working properly. PROC PRINTS can be added after every step to show how the data is changing step by step. To examine how data changes within a step, PUT statements can be used to print out the value of variables as they are being processed. The DATA step debugger can also be used to execute a DATA step one statement at a time and to print out variable values after each statement is executed.

Resources for Further Study

Many people that ask for debugging help do not take advantage of the online help feature in SAS. Searching the Index Tab in the Help Topics window finds discussions of different types of errors. For example, the Accessing Files page explains that the error

```
Error: File is in use, filename.
```

indicates that “The file you are trying to access is in use by another Windows process, such as another Windows application.”

Another feature in SAS that many beginner users could learn much from is the online tutorial. Module 1 Lesson 3 explains how to debug programs and allows the user to practice debugging skills with simple programs built into the tutorial.

The Technical Support area of the SAS Institute web site, www.sas.com, can provide support in the form of the database of questions and answers to common problems. A search engine is available to help find information on specific topics.

The SAS-L listserve is a forum where people can ask questions or seek guidance on programming problems. The list is very active and helpful.

The papers and books listed in the references also provide guidance in learning how to debug SAS programs.

Conclusion

To debug programs successfully it is important to understand how SAS runs programs. SAS first checks a step for syntax errors. If there are no syntax errors, SAS compiles the step and runs data through it. Depending on the way the step is written and what data is read into the step, execution-time errors or data errors may be generated. Logic errors can also produce undesired results.

It is important to know what the program is trying to achieve and to understand the data the program is using. Reading the log for all three kinds of messages: errors, notes, and warnings is crucial to confirming that a program is working properly. Do not rely on the output alone, and do not assume that the program ran correctly just because there are no error messages in the log.

While the task of debugging a program may be difficult at first, the task gets easier with a little practice. Understanding why an error has occurred will not only help prevent it from happening again in the future, but will also help provide insight as to how SAS works. This can help one write better programs with fewer bugs.

SAS is a registered trademark of SAS Institute, Inc. Cary, NC.

References

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

Howard, Neil and Linda Williams Pickle and James B. Pearson (1996). It's not a Bug, It's A Feature!! *Proceedings of the 21st Annual SAS Users Group International Conference*, pp. 370-378.

SAS Language and Procedures: Usage, Version 6, First Edition (1989). SAS Institute, Cary, NC.

Virgile, Bob (1996). The Dirty Dozen: Twelve Common Programming Mistakes. *Proceedings of the Northeast SAS Users Group '96 Conference*, pp. 205-210.

Walega, Michael A. (1997). Search Your Log Files for Problems the Easy Way: Use LOGCHK.SAS. *Proceedings of the Northeast SAS Users Group '97 Conference*, pp. 343-344.

Contact Information

Peter Knapp
U.S. Department of Commerce
14th & Constitution Avenue, NW
Room 7866
Washington, DC 20230

202/482-1359 (voice)
202/482-1388 (fax)

Peter_Knapp@ita.doc.gov