Programming for Failure:
When Programs Faw Down and Go Boom

Gary E. Schlegelmilch, US Bureau of the Census, Suitland MD

ABSTRACT

Programs fail.

Despite the best efforts of the best programmers, a non-working combination of data, software, and processing is invariably going to happen - and the program is not going to run. Typical result is a sometimes instantaneous, sometimes arduous effort to find where the program has failed - and fixing it.

This paper attempts to cover two major topics: (1) a few ideas for finding the elusive bug, and (2) planning and structuring a SAS program so that perhaps a few less of them happen.

IN SEARCH OF PROGRAM FAILURE

Program failure is addressed in virtually every advanced programming class, and at least mentioned in beginning classes. It's called other things, but the bottom line is this: programs fail, be ready for it.

When learning to code, our first programs are always pretty simple. Read a file, accept data from the screen, or get data from a dataset. Based on that data, perform some kind of test or calculation, and output it. Simple enough. The sad truth is that too many people *stop* with that approach, and go on to write programs for years, sometimes decades.

However, over the years, I've come to the following three simple rules for data processing:

1. Data is not always what you expect.
2. It's tough to program for every contingency - and it can be fun to try.
3. Go back and read Rule 1 and Rule 2 again.

In support of another paper, Ian Whitlock kindly gave me this small gem of software design, innocuously called the 'jiggle' test. Put simply, when he is handed a program requirement or specification, he 'jiggles' it a little to see what the impact would be. If a small potential change down the road would disrupt the whole theory of how the program would function; then, it's time to reexamine how the process is done, to allow it greater flexibility for both functionality and in maintenance.

Another important aspect to problem solving of any kind was given us by Sir Arthur Conan Doyle, via his Sherlock Holmes character. He said, in The Sign of Four, *"When you have eliminated the impossible, whatever remains, however improbable, must be the truth."* It's a good philosophy to keep in mind.

I would imagine that every programmer since Ada Byron Lovelace and Charles Babbage have probably, at one time or

another, uttered the words, "It's not supposed to ***do*** that". It is also true that, no matter how many times we utter those mystical words – the program never seems to realize that it's making the mistake, and correct itself. No, once the elusive program bug manifests, we have to swallow that bitter pill. Somewhere along the way, *we* have told the computer to do something – and it's doing what it's told. Therefore, Watson, it falls to us to find the problem and repair it.

First, we'll look at a few sample problems, which were not obvious at first glance. Then, I offer a few ideas on how to keep them from happening in the first place.

A PROBLEM, OR NOT?

I had a recent instance where a program was to read in three numbers from a source that was ostensibly providing three integers. Yet, printing out the three numbers showed that each of them was 3 - but the result was consistently 10.

Why?

After staring at the simple formula, and trying things like the SUM function, it occurred that perhaps there was ***nothing*** wrong with the code. I looked at the data, and it read as follows: 3, 3, 3. However, PUTing it out with an 8.4 format made it read 3.16, 3.37, 3.468. Sum, 9.998, and displayed as 10.

Solutions: (a) do an INT(n) function to reduce the input fields to their integer values, thus reducing the sum to the integer value; (b) do a ROUND(n) on the input fields, rounding to the nearest integer, which will provide an integer sum; or (c) use the ROUND(n) or INT(n) on the resulting sum. Which is correct?

Answer: unknown. Sounds evasive, but true. This is a case where it is not a case of the data being correct or incorrect - but the concept behind the data. Is the data supposed to be rounded or truncated to an integer when it arrives? Use the appropriate function. If the resulting sum is integer, ensure that it is. That way, later uses of the resulting data will not experience the same problems.

Another thing to remember; the SUM function will save you heartache when adding numbers, when you are not absolutely sure if the numbers are there. Given A=5, B=<missing>, C=10; X=A+B+C gives a value of <missing> for X, but X=sum(A,B,C) yields the correct value of 15. Oddly enough – should you overzealously type X=sum(A+B+C) – it still tries to do the add, and the result is <missing>!

When searching for program error, remember that there are three places to look:
1. Operating system
2. SAS
3. Data

***

Here's an example of a glitch via the operating system. I had a short program that built several directories on a UNIX platform, and then placed files in them. When I went to go to the next module, which read those files, the program failed to find them. After chasing the SAS code fruitlessly for a while, an

experienced UNIX programmer identified the problem.

I needed test files numbered 1 through 10, so I wrote the following simplified code:

```
%macro BUILDFLS;
  %do I=1 %to 10;
    length FN $ 10;
    data _null_;
      FN = 'test'||&I||'.dat';
      file FN;
      put 'This is test file '
"&I";
    run;
  %end;
%mend BUILDFLS;
```

Looked right to me. A directory listing showed me that there were indeed files FILE1.DAT, FILE2.DAT, *et al*, on the appropriate directories. However, SAS couldn't find any except FILE10.DAT; neither could the UNIX editor, when I went to examine them. My UNIX expert showed me that the files could be found with wildcards, proving they did indeed exist. But, I had created them in a $10. formatted field, so they were being written as "FILE1.DAT☆", "FILE2.DAT☆", and so on.

Solution:

```
%macro BUILDFLS;
  %do I=1 %to 10;
    data _null_;
      FN&I =
compress('test'||&I||'.dat');
      file FN&I;
      put 'This is test file '
"&I" '!';
    run;
  %end;
%mend BUILDFLS;
```

Each filename would now be built into an individual dataname, FNn. As it was the initial use of the field, SAS would set it to the exact length needed, since the COMPRESS function would eliminate any blanks. FN1 through FN9 would be 9 characters long; FN10 would be 10. And the files were built without the trailing space, so UNIX could find them just fine.

Lesson Learned: Be *sure* of the data you're outputting - just eyeballing it isn't always enough.

Sometimes, an error can be caused by the most innocent conflicts. Case in point; a recent incident at the office had a program getting erratic results from a process, an external macro which had been used by a number of other programs for quite some time. It turned out that, by coincidence, one of the external macro's being INCLUDEd and called was using the same macro variable name as the calling routine.

(c:\macros\testmacro.sas)

```
%macro GETNEWVAR(NBR);
  ..
  data WORK.FACILITY;
    ..
    %let TEMPVAR=_n_;
    ..
  run;
  ..
%mend GETNEWVAR;
```

(calling program)

```
%let TEMPVAR=7;
%include
'c:\macros\TESTMACRO.SAS';
%GETNEWVAR(TEMPVAR);

data _null_;
  XVAR = &TEMPVAR + 15;
  put XVAR;
run;
```

Both programs were actually accurate in what they were doing; however, since

both were coincidentally using the same macro variable name, a routine that had been running without problems for a year was now providing incorrect data for one survey.

Solution: the %LOCAL statement was added to the %MACRO GETNEWVAR to define the local variable. That way, the values set for the variable remained static within the individual macro routines being run.

CONFLICTING REQUIREMENTS

In a program originally developed on a VMS platform, the requirements stated that the system store a numeric date field, and display it in YYMMDD6. format. Easy enough. Easily a dozen places in different programs, batch and interactive, saw the changes.

And just as surely, once the programs started to run – they started to crash.

I dutifully reported to the user that we were getting bad data in, and how would she like it handled?

She looked at the reports, smiled, and said that the 'D' in the field was perfectly valid, and it represented, "Don't Know".

I pointed out a 'D' was an alpha character. She agreed. So I should make the incoming field alphanumeric? No, she wanted it numeric.

Solution: accept that we needed an alpha field to represent a numeric missing field, like so:

```
data WORK.DATEFILE;
   format DATEREP YYMMDD6.;
   missing D;
   input @1 NAME $40.
         @32 DATEREP YYMMDD6.;
run;
```

Results; the program accepted that a 'D' in this numeric field was a specific representation of a missing value, and there by intent. Any program that read this dataset, however, had to have the missing statement in it, or the program would abort on finding the non-numeric value.

Just putting the format of the DATEREP field isn't enough, either. If a PROC PRINT followed the DATA step, it would have simply reflected the numeric value of the SAS date. By adding the FORMAT statement, the date prints out in exactly the format we'd expect. Now, use of the MISSING statement is fine when the potential non-numeric value is known. But if you want to ensure that you process only numeric values, here's a different approach:

```
data WORK.DATEFILE
     (drop=DATEREPX);
   format DATEREP YYMMDD6.
          DATEREPX $6.;
   input @1 NAME $30.
         @32 DATEREPX $6.;
   DATEREP=
    input(DATEREPX,?? YYMMDD6.);
   if DATEREP eq . then put
'Invalid date in observation '
 _n_;
                   else output;

run;
```

This protects you overall from any bad data. If the date field is in the proper format, it's converted and stored. If not, it leaves a missing value in DATEREP. The '??' in the INPUT function tells SAS to suppress any error messages in the log, and to not set the _ERROR_ variable. That allows the program to handle all the error handling, and to

continue to process data. A single '?' would suppress the error messages to the log, but the _ERROR_ flag would still be set to 1, potentially stopping the process.

SOMETHING YOU CAN COUNT ON

SAS programmers often use the _n_ variable, usually as a counter of how many observations were read. The common misconception is that the _n_ represents the number of observations that have been read. Not so. _n_ ***always represents the number of iterations of the DATA step***. If one SET statement is performed on each iteration of the DATA step, you're safe because the number of iterations and the number of observations should match. However, in the example:

```
/* add up the SUBTOT for each of
   the five regions, and
   output the summary for each
   product where the sales quota
   has been met.              */

data WORK.SUMMARY
    (keep=REGION TOTAL);
   set WORK.SALES; TOTAL+SUBTOT;
   set WORK.SALES; TOTAL+SUBTOT;
   set WORK.SALES; TOTAL+SUBTOT;
   set WORK.SALES; TOTAL+SUBTOT;
   set WORK.SALES; TOTAL+SUBTOT;
   if TOTAL > 400000 then
output;
run;
```

In this case, _n_ would equal neither the number of observations in the input dataset, nor necessarily the output dataset. It should equal the total number of observations in the WORK.SALES dataset, divided by 5, truncated to the next higher integer value. Not a particularly useful number, so a PUT _n_ statement to use in determining the place of an error wouldn't be much help.

Notice too that we use the implicit ADD to good use here. On each iteration of the DATA step, TOTAL will reset to missing. If we had said TOTAL=TOTAL+SUBTOT, the result would always have been missing, since one of the variables in the equation was missing. Since we used the implicit add, the missing was simply ignored. It could also have been coded TOTAL=sum(TOTAL,SUBTOT).

"CHECK THE DOCMENTATION" DOESN'T ALWAYS MEAN READ THE BOOK

Here's one that drove us crazy for a little while:

```
/*  This is a new routine  */

* check on the value of the
   input field  */

%macro TEST1;
  %if "&NEWVAL" eq "FNL" %then
%do;
    %ACCEPT_FINAL;
  %end;
  %else %do;
    %ACCEPT_PRELIM;
  %end;
%mend TEST1;
```

However, a run of %TEST1 resulted in the following error:

```
1    /* This is a new routine */
2
3    * check on the value of the
4       input field */
5
6    %macro TEST1;
7       %if "&NEWVAL" eq "FNL"
%then %do;
```

```
ERROR: The %IF statement is not
valid in open code.
```

Several eyes looked at it before we realized that it had not a thing to do with the perfectly-valid %IF statement. Nor was the problem in the %MACRO statement, which would allow the use of the %IF. No, the guilty culprit here was a comment line prior!

Had it begun with a "/*", the code would have run without incident. Or, had it terminated with the semi-colon, fine. But since the comment line began with an asterisk, it continued to read until it found a semicolon – and the first one it found was at the end of the %MACRO statement. So, to the program's "eyes", the user was still in open code, because the %MACRO statement was part of the comment; and the %IF was indeed illegal.

MAKE THE BUGS EASIER TO FIND

In any program, be it ten lines or a thousand; the easier the program is to read, the easier it is to locate points for update, and to find program failures. Notice, I never said that all of the problems would stick out – just make them a bit easier to see.

Structuring a program is, to say the least, an art form. If you ask 100 programmers the correct way to structure a program, I would estimate at least 150 different philosophies would come to light. As such, I do not advocate any hard-and-fast rules; just a few things that I use (and have shown in all the programming examples here) to make my programs a little easier to read.

First, user-defined terms are capitalized or in all-caps; SAS terms are in lower-case. In e-Speak, the sub-language of e-mail and the Internet, anyone speaking in all-caps is "shouting". Well, when you're searching for potential problems in a program, I always found it easier that the terms we define as programmers "shout" at you as potential problems; lower-case are the SAS language items, and are not typically the source of a problem.

Indentation; again, a lot of opinions here. I indent only to show subordination. The steps enclosed within a %MACRO routine, the steps only executed within an IF… DO, and so on.

Some language constructs lend themselves well to structure. As an example; you have to test a field for a number of potential values, to ensure validity. If there are only a few, a simple IF will suffice;

```
if INDTYPE eq "A" or INDTYPE eq
"D" then do;
```

but if there are a large number of values, the IF… AND… AND… AND… can become cumbersome and difficult to follow. In this case the IN becomes more useful;

```
if INDTYPE in('A', 'C', 'F'-'Q')
then do;
```

Note here that a range can also be used in the IN statement, cutting down on extra typing even more.

If there are many observations to process, and a different action is required for each type, then the IF could be used as follows;

```
if INDTYPE eq 'A' then %TYPEA;
else if INDTYPE eq 'B' then
%TYPEB;
```

and so on. You might even enclose a last ELSE statement, in the event that none of the expected values are in INDTYPE. Or, you could make use of the SELECT statement, which lends itself to structure, readability, and flexibility for multiple values.

```
select (INDTYPE);
  when('A') %TYPEA;
  when('B') %TYPEB;
  ..
  ..
  otherwise put 'PROGRAM NOTE:
ID ' ID ' had no valid INDTYPE';
end;
```

Another relatively painless, but informative, way to code a line; use the '=' sign only for assignments, as in X=20; use the letter equivalents for comparisons, as in if X eq 20. Again, it gives the quick advantage of being able to see at a glance where you're assigning data values, and when you're doing comparisons, because only the assignments will contain symbols.

The simplest overall advice for structuring a program, of course, is one of simplicity. If you can read your program at a glance, you probably are already using your own structure tools and techniques. Ah, but if you can put that code in a drawer, not look at it for a year, and then be able to tell at a glance where to look to update or modify it; the structures are good ones for you.

Invariably, however, there will be programs that will not be yours during the entire software life cycle. Once you inherit someone else's programs, you'll recognize just how valuable simple structuring techniques are.

USING ALL THE TOOLS IN THE TOOLBOX

The /DEBUG statement is an extremely effective tool for debugging DATA steps, and one not to be ignored. Many good papers and workshops have been written on the techniques available, so I will not elaborate here. However, here is a good way to incorporate debugging into a live program so that it need not be modified in order to debug it in Production.

If there are no existing parameters to the program, set the system default of SYSPARM to '/DEBUG'. Then, in the trouble spots of your program;

```
data WORK.NEWFILE &SYSPARM;
   set PROD.DATAFILE;
  ..
  ..
run;
```

You will be able to run the same program in Production, with your live data, and yet only run the debugger when you wish. In a program with numerous potential spots for review, perhaps this would work more effectively, by setting SYSPARM to a number, and coding it like this;

```
%if "&SYSPARM" eq "1" %then %let
DEBUG1="/DEBUG";
%if "&SYSPARM" eq "2" %then %let
DEBUG2="/DEBUG";
%if "&SYSPARM" eq "3" %then %let
DEBUG3="/DEBUG";

data WORK.FILE1 &DEBUG1;
  ..
  ..
run;

data WORK.FILE2 &DEBUG2;
```

```
  ..
  ..
run;

data WORK.FILE3 &DEBUG3;
  ..
  ..
run;
```

Another useful tool is the argument CANCEL to the RUN statement. Yes, that same RUN statement we use to end all the DATA steps has an argument string available. Keying RUN CANCEL at the end of the DATA step allows the DATA step to be checked for syntax, but to take no other action. This becomes valuable in a case where two DATA steps run in sequence; but the second should not run unless the first creates a specific condition. So:

```
/* only run the report if  */
/* over 100 valid records  */
/* are found               */

data WORK.NEWDATA;
  set PROD.OLDDATA;
  where ERR ge 1;
  if _n_ gt 100 then
call symput('CANX','CANCEL');
run;

proc print data=WORK.NEWDATA;
run &CANX;
```

CONCLUSIONS

When you combine the number of requirements that go into the design of a program, factor in the number of data possibilities one might encounter, and the near infinite diversity in infinite combinations of a computer language; it's remarkable that we get as much done as we do. However, on any given day, we get a little more skilled, and a little more knowledgeable; not only on how to find and fix computer errors – but to learn to program so they won't happen at all.

REFERENCES

Aster, Rick, **Professional SAS Programmer's Pocket Reference, 3rd Edition**, 2000, Breakfast Books.

Riba, S. David, Jade Tech, Inc., **SAS Debugging Techniques**, SESUG '98.

Doyle, Sir Arthur Conan, **The Complete Sherlock Holmes, Vol. 1**, Doubleday, 1960.

CONTACT INFORMATION

Gary E. Schlegelmilch
U.S. Dept. of Commerce, Bureau of the Census, ESMPD/MCDIB
Suitland Federal Center, Rm. 1200-4
4700 Silver Hill Road, Suitland MD 20746
Email
Gary.E.Schlegelmilch@census.gov