ERROR: WHAT TO LOOK FOR WHEN DEBUGGING SAS PROGRAMS
Brit Minor, Pharmaceutical Research Associates

ABSTRACT
Have you ever had an experienced SAS programmer glance over your shoulder for ten seconds and spot an error that you’ve spent hours trying to find? That programmer probably saw an old friend, an error she/he’s made and caught many a time. This paper presents a compendium of common SAS errors, their warning signs, and suggestions for how to fix or avoid them.

INTRODUCTION
Debugging is a necessity born of undisciplined but all-too-human practices. If you put enough effort into the design phase of your project [with pen and paper or with computer used solely as a notepad], are familiar with SAS’ eccentricities, and can write code without the designed code, a lot of spare-time experiments, your programs should run correctly the first time or require minimal debugging.

Like most of us, I spend a lot of time debugging. In the following sections, I’ll explain the use of the SAS log as a debugging tool, offer a symptom-by-symptom look at common SAS coding errors, and discuss some general debugging techniques. Finally, I’ll try to convey a few tricks of the “over-the-shoulder” guru, in hopes that you can save yourself time and potential embarrassment when the next all-night debugging session looms.

THE SAS LOG
The SAS log is the first place to look for clues as to how you’ve gone astray. SAS error messages, warnings, and notes help guide you, often indirectly; with practice, these messages become more enlightening.

Notes at the end of each SAS step tell you how many observations, variables, and pages of output resulted from the step. You should compare these to the numbers you had expected. Are the number of observations being consistent with what was input? Other notes may inform you of uninitialized variables, missing values, character/numeric conversions, or ill-formatted values. These notes may provide the first clues that something isn’t as you expect.

Some notes are cause for alarm. The following two, in particular, signal potentially grave problems with your data or program design:

**NOTE:** Merge has more than one data set with repeats of BY values.

When you merge data sets, you tell SAS how to structure the merge by using a BY statement. If you don’t use enough by-variables, you can wind up with groups in each data set that share all by-variables but still have multiple observations. SAS then does something that is nearly always not what you want: it match-merges within the group until it runs out of observations, then it merges the last observation of this group with the remaining observations of the larger group. If you see this message, stop and reassess your data and your design. You probably need to add a by-variable to further distinguish the observations.

**NOTE:** SAS went to a new line when INPUT statement reached past the end of line.

If each record of your data file is one line, you certainly don’t want SAS to try to combine two or more lines when reading a record. This SAS note tells you, implicitly, that SAS is only reading in half your data: every other line is getting included by mistake with the previous line. If you haven’t made an outright error in your INFILE statement, you can often fix this problem by using the missover option on your INFILE statement.

I mention these two notes to emphasize that you really must check the log carefully to be sure your job has run properly. If SAS has printed a message that you don’t understand, don’t just ignore it. Ask someone about it. SAS is probably telling you that it has detected a dangerous programming practice, one that sometimes is okay but usually is inadvisable.

Other notes are cause for concern as well. Causes and cures for the following messages will be discussed in the compendium which composes the middle section of this paper:

**NOTE:** Invalid numeric data, xxx=*y+y* at line...
**NOTE:** Variable xxx is uninitialized.
**NOTE:** Missing values were generated as a result of performing an operation. At line...
**NOTE:** The data set xxx.yyyy had 0 observations and zzz variables.

Version 6.06 of SAS provides more informative messages than past versions. One of these I particularly enjoy:

**NOTE:** Previous statement has been deleted.

It’s nice to be told that your statement was so nonsensical that SAS just ignored it.

If you are faced with checking one or several long SAS logs, you might consider first running a search for the keywords that appear in SAS’ nastiest messages:

**ERROR;** WARNING; uninitialized, invalid, repeats, reached, converted, deleted

If the search finds none of these words in your log, your program has passed one hurdle but is by no means thoroughly debugged.

TYPES OF SAS PROGRAMMING ERRORS
Most programming errors can be classified as resulting from messy data, syntactical or semantic misuse of the language, or logical errors in the program design. You should protect your program from data problems, such as out-of-range or missing values, since it is likely that no one else will. Syntax errors and semantic disagreements are usually pointed out, pretty emphatically, in SAS notes to the log. These errors most often result from typos, omitted symbols, or poor choice of variable names.

Logical errors, unfortunately, are seldom caught by the compiler. Your code should put messages to the log warning you of consequences of any impoverished logic you’ve committed to code. Of course, these messages do no good if they are written to the log but not looked at. One trick is to start them with a “nasty” keyword that you include in the list of search terms given above. At PRA, we signal critical messages with “Problem;”, coded thusly:

```plaintext
put 'Prob' 'lem: ' ... 
```

This coding prevents the search from finding “Problem” in the
put statement. An alternative technique is to write the critical message out to your listing, so that it can't be ignored.

PROBLEMS WITH DATA

SAS programs deal with messy, real-world data. Values may be missing; whole records may be missing; dates entered incorrectly so that records may be out-of-sequence; fields that are guaranteed to only have 1s and 2s in them all too often have a few 3s as well; numbers like 99 may really be codes for "No Data."

Your program should not be blind to errors that live in the data. However, you should not waste time debugging your code when the invalid results you are seeing come from outside your program. Therefore, get to know your data.

A COMPRENDIUM OF COMMON SAS PROGRAMMING ERRORS

The bulk of this paper is a symptom-by-symptom discussion of common mistakes that are made repeatedly by SAS programmers.

Our first class of errors comprises those related to data and missing values. The programmer who assumes that the data will be perfect lives in constant danger of producing misleading results. Missing values, in particular, can wreak havoc on merge and flag-setting operations.

Symptom: Merged records have sequences of missing values.

These probably arose from some non-merging observations among the merged data sets. This condition may be okay, or maybe you really meant to delete certain kinds of non-merging records. You can and should check for records that fail to merge by using the 

This will help you avoid rude surprises.

Symptom: Some wild values among otherwise valid ones.

When doing complex calculations, be sure to put a few results, the values that went into them, and values of any flags that would affect the calculation. You need complete information to find obscure errors.

Symptom: Too many messages being written to obscure logs.

Be careful not to print out so much that the user gets swamped. You want errors to be found, not buried. A useful trick to limit output is the following:

This will help you avoid rude surprises.

Symptom: SAS "ignores" some statements. SAS mixes up two statements.

Occasionally, SAS may be able to pair up quote marks that you intended to be in separate statements. SAS will ignore your intent, assign a nice long string, and, in the process, "not see" some of your intervening code.

Symptom: Your most valuable data set disappears from the system. SAS accepts this code, writes an "uninitialized variable" message to the log (for "parents").
Symptom: SAS fails to translate a macro variable in a string.

Macro variables need to be enclosed in double quotes or special macro quoting functions. Don't embed macro variables in single quotes, e.g. "&macva(. Period.

Symptom: SAS politely refuses to use your data library. When you mis-appropriate a SAS keyword.

Symptom: No-longer-correct values are held over from previous by-group.

Symptom: Values that should be missing get filled in.

Symptom: Sums are too big.

The preceding SAS code segment was incomplete. What if a code was read in from "lookup" that wasn't in "codes"? Forgetting to reinitialize a retained variable at the start of a new by-group can cause problems such as those in the above symptom list. Retain statements should be accompanied by code such as this:

GOOD: data lookedup;
merge lookup (in=inlook)
   codes (in=incodes keep=code word);
by code;
if first.code then do;
   if incodes then codeword = code || word;
   else codeword = ' ';
   retain codeword;
if inlook;
run;

Symptom: Flags set improperly.

Two real sneaky problems can arise when you combine subsetting operations with other conditional processing in one step. If you are using the end=eof option on a set or merge statement and have some code that should execute when eof is true, it will fail if your subsetting operation throws away the last observation too soon. Similarly, if you are doing by-group processing and are expecting SAS to reset some values if first.this or output some records if last.that, it may take you a while to realize that the conditions would have been met had the record not been discarded just before. Here's a simple example:

BAD: data results;
merge acts (keep = id date action)
effs (keep = id date effect)
   end = eof;
by id date;
   if effect = '1';
   n = 1;
if last.id then output;
if eof then put n=; run;

This code is supposed to output one observation for every id that had an effect of 1. However, as written it will output records only when both conditions are true, namely "effect = '1'" and "last.id." The put statement may never be reached. The same problem may crop up when using the delete statement or even when one condition is embedded in an If-then block.

BAD: data results;
merge acts (keep = id date action)
effs (keep = id date effect)
   end = eof;
by id date;
   if effect = '1' then do;
   n = 1;
if last.id then output;
end;
if eof then put n=; run;

Now the eof condition will be reached, but some lines still may not be output.

To solve this problem, you can use the where statement (available in SAS version 6) in place of the subsetting If statement in this section's first BAD: data step. Another safe solution is to use a
Symptom: Unexpected missing values.
Symptom: Too few observations.
Symptom: Incomplete array operations.

GOOD: data results;
merge acts (keep = id data action)
opts (keep = id data effect);
by id data;
if effect = '1';
data results;
set result end = eof;
by id;
n + 1;
if last.id then
output;
if eof then
put n=;
run;

Symptom: Too few observations or too worried, if you trigger a lot of those non-match messages that you've been exhorted to include in your code when merging data sets. 50 observations in a clinical tab data set may apply to only one patient.

Symptom: Unexpected missing values.
Symptom: Incomplete calculations.

When declaring arrays: be sure you've counted light. If you list eight variables as being part of an array but give its dimension, SAS will be happy. This problem can arise when you borrow from an old program, one designed for a different data set. Here's two examples:

BAD: array test [10] atest btest ctest dtest etest ftest gtest htest itest jtest ktest;
miss test = 0;
do i=1 to 10;
if test[i] = . then
miss test = miss test + 1;
end;

BETTER: array test [*] atest btest ctest dtest etest ftest gtest htest itest jtest ktest;
miss test = 0;
do i=1 to dim(test);
if test[i] = . then
miss test = miss test + 1;
end;

In the first example, SAS processes 10 array elements even though there are 11 listed; "ktest" is left out. In the second, SAS counts the number of array elements via the [*] in the array statement and the dim function. If you add or subtract test items, you needn't worry about counting them.

Of course, if you've browsed through the chapter on SAS functions, you'll know that the above eight lines of code could be replaced with one function: nmiss(atest,...,ktest) would count the number of missing items for you. It pays to read the manual!

Symptom: Too few observations.
Symptom: Incomplete array operations.

Nested do loops that use the same index can result from careless additions to a program. This particular error is a personal favorite, I've done it more times than I care to admit. What will the following code do?

do i=1 to 5;
  junk = food[i];
do i=1 to 5;
brew = beer[i];
output;
end;
end;

It will output just 5 observations, not the 25 you expected, because "i" is incremented to 5 in the second loop.

Symptom: Too few observations or too worried, if you trigger a lot of those non-match messages that you've been exhorted to include in your code when merging data sets. 50 observations in a clinical tab data set may apply to only one patient.

Symptom: Incorrect computations.

Do you write complicated expressions without parentheses? Do you also line high-wire walking without a net? It takes a while to realize that the expression:

if a or b and c or d;

is equivalent to

if a or (b and c) or d;

especially if you thought SAS would see it as

if (a or b) and (c or d);

which it won't. Using parentheses liberally will help cut down on computation errors.

Symptom: SAS ignores or misinterprets an else clause.

SAS may be seeing the code differently than you've written it:

BAD: if apples then
    if oranges then
        a = b;
    else
        a = c;
SAS will connect the above else clause with the first If, not the second, your indentation to the contrary. You need to enclose the If-then-else in a do-end block:

GOOD: if apples then
    do
        if oranges then
            a = b;
        else
            a = c;
    end;

If the unmerged data sets share a variable that is not in the by statement, you will write over values in the first data set with values from the second. To avoid this, either rename one of the variables (use the renames option on the data set name), or, if the
data sets are so sorted, add the variable to the by statement.

**Symptom:** "Extra" variable.
**Symptom:** Invalid data message from SAS.
**Symptom:** Input values or data _null_ output lines overwritten.

If you omit a decimal point in a format on input or put statements, you get a column specifier or another variable reference:

```plaintext
put a $6. b 4 c 5. d agefmt;
```

The put statement above will write "a" in $6. format starting at column 1, will write "b" in its default format starting at column 4 (instead of in 4. format starting at column 7, which is no doubt what you intended), and will wind up writing "d" in its default format, not in "agefmt." Finally, SAS will print the message "Variable agefmt is uninitialized" to the log. Had this been an input statement, "b" might have re-read some character data from column 4; an "Invalid data" message might have been the result.

**Symptom:** Critical messages unseen due to being buried in long logs.

Too many errors or put statements: when the log gets really long due to lots of messages, a critical message may go unnoticed. This kind of problem is what makes debugging an iterative process: solve one problem so you can see the next one. Don't spend too much time wading through a SAS log that's more informative than you ever wanted it to be, just fix one of the major problems and return.

**Symptom:** Character-to-numeric, numeric-to-character conversion messages.

Conversion messages may not signal problems. SAS will convert numeric to character values without losing precision in the result. SAS also will reliably convert numbers stored in character variables to the equivalent numeric value. However, conversion messages may be a clue that, while YOU may think that all of the variables involved in an expression are numeric, SAS considers one or more to be character. You may not want SAS to decide for you how these values should be translated.

⇒ Several common problems may arise when manipulating strings. Here are a few choice ones:

**Symptom:** Truncated string.
**Symptom:** Data set size much larger than expected.

The length of a SAS character variable is set the first time SAS learns about it within a data step. That first value should be in a length statement in most cases. Otherwise, SAS may set the length equal to that of the first value encountered (e.g., evar = test; where test is a pre-existing character variable and evar is a new variable) or to $200 (!!) if certain functions are used (e.g., = scan(test,1));

**Symptom:** Concatenations fail, despite best efforts.

When concatenating strings, be wary of unseen "trailing" blanks. You can spend hours worrying over code like the following:

```plaintext
length this that theorem $8;
          this = "pell";
          that = "mell";
          theorem = this || that;
```

What does `theorem` equal? It won't be "pellmell," and here's why. The variable `this` gets set to a 4-character string, but SAS had been told that it was to be length $8$, so SAS silently put 4 trailing blanks on the end. The concatenate operation worked, yielding "pell mhell," then SAS chopped off `theorem` to be length $8$, resulting in `theorem = "pell"`.

A safe way to concatenate is by using the trim function:

```plaintext
theorem = trim(this) || that;
```

**Symptom:** Unexpected missing values.
**Symptom:** If-then block never entered.

Another common problem is comparing two strings where one is uppercase and the other isn't. SAS is case-sensitive. If you aren't sure about the case or don't care, use the upcase function to convert the string to uppercase:

```plaintext
if upcase(theorem) = 'PELL';
```

**FUNCTIONAL GROUPS OF CODING AND LOGICAL ERRORS**

To summarize and augment the compendium of what can go wrong, I've rearranged the types of errors into functional groups rather than symptom groups. This regrouping may help you when doing an over-the-shoulder quick scan of errornous code; it may help you achieve that alternative mindset that successful debugging usually requires.

**Pairing Up**
- Semi-colon
- retain statement

**Missing Tokens**
- Semi-colon
- retain statement
- Omitting decimal point from a format.

**Incomplete Edits**
- Forgetting to keep a newly added variable
- Adding a by-variable without checking first, or last, references
- Forgetting to change set to merge after adding data set to a set statement.
- Adding or removing variables from an array without changing array bounds.

**Improper Counting or Coding**
- Declaring arrays with too few slots.
- Using a format too small for some values.
- Misspelling a variable name so that it matches another var name.

**Just Plain Wrong**
- Nesting do-loops with same index variable.
- Using set when you mean merge, or vice versa.
- Using tape as a libname for non-tape library.
- Letting an %include file call itself.

**Side Effects**
- Overwriting values during a merge.
- Not declaring length of character variables.
- Using length < 8 for floating point numbers.
- Not realizing that SAS evaluates all clauses of a compound boolean.
- Unseen data warnings because SAS stopped printing any after first 20.

**Events Out-of-Sequence**
- New by-values not known yet when printing a header.
- Flags re-initialized too soon or too late.
- Symput-defined macro vars not available yet.
- Referencing a data set that doesn't exist yet.
Boundary Values and Unexpected Conditions
Using < or > when you should use <= or >=.
Using wrong by-variable when setting flags.
Omitting others in format or otherwise in select.

Strings
Length too short to hold all values.
Concatenating string to an existing variable without trim function.
Ignoring upper!lower case when comparing.
Reading a character variable from end of variable-length record without first specifying length of the character variable.

Dangerous Practices
Merge without a by.
Using too few parentheses.
Setting flags after deleting observations in the same step.
Using first, or last, values or eof after a delete or subsetting if or within an if-then block.

SOME GENERAL ADVICE ON DEBUGGING

The big trick to complex debugging tasks is to seek a new perspective.

An insidious problem with trying to debug your own code is just that: it's YOUR code. Your thought patterns produced it, your fingers typed it in, your eyes are bugging out at it. What you probably will wind up doing when trying to fix a nagging error is to keep pounding away with the same mindset that produced the error. What are your chances of finding that error? Low, as I know from many frustrating hours and days of doing just that.

If you suspect you're in that bind, try to break out. A great way is to get another person to look at your code -- choose someone friendly if you fear you've made a REALLY dumb error! If you can, put the problem aside till another day. If you must keep at it, challenge your assumptions:

- Print out some observations from all data sets that are involved, including ones that you are sure are okay.
- Run simple proc freqs to look for unexpected values.
- Remind yourself of odd conditions imposed by the data.
- Try a different SAS method of doing the "same" thing.
- Make a special test data set, not just the first 50 observations of the one you're using. This data set should include observations and values that will test all important combinations and conditions of your code.

PROGRAMMING DEFENSIVELY

You can avoid a lot of debugging simply by following some oft-repeated guidelines. Write down the program's design before you code. Develop a consistent coding style, one that you and others can read easily. Indent. Use white space. Use descriptive variable names. As discussed in an earlier section, assume the data will be messy; design your program so it won't be defeated by the data. Try hard to ignore interruptions if you are in the midst of a series of tedious, interlocking changes. And if you are stuck with a section of awkward, questionable code, alert yourself and other readers to that fact, LOUDLY !

HOW TO BE YOUR OWN GURU

The most important key to being effective at debugging your programs is to seek and gain a new perspective on your problem. Here's a few closing tips on how to gain insight into SAS programming problems and solve them more quickly:

⇒ Learn all you can about the SAS system and how it processes data and code.
⇒ Reassess your programming assumptions. Rethink the problem.
⇒ Check the SAS log carefully. Don't ignore SAS notes.
⇒ Diagram program and data flow, even after the code is written.
⇒ Apply lessons learned from other programming languages.
⇒ Write lots of SAS code.

And, if nothing else seems to work ... call Jim Goodnight!

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