CONQUERING THE DREADED MACRO ERROR
Katie A. Hubbell, Computer Data Systems, Inc.

ABSTRACT
The Macro facility is a very powerful tool in SAS programming. By the same token, it can cause some powerful and, at times, intimidating errors. As macros become more complex with increased programmer skill, it becomes more and more important to form a system for writing macros that avoids as many pitfalls as possible. Debugging must become more systematic and orderly to avoid wasting time and computer costs.

This paper outlines some simple methods for ensuring that macro code is as clean as possible before testing, as well as methods for testing and debugging macros efficiently and quickly.

INTRODUCTION
Macro errors are a different species than SAS errors. There is a higher level of difficulty in resolving macro errors. In base SAS, error resolution is a two-step process: 1) find out why the error occurred, and 2) find out how to fix it. With the addition of macro, an extra step is needed first: find out where the error occurred. Also, when macro is involved there is more than one type of error. There can be:

1. Macro errors (for example, 1301, 1555);
2. SAS errors and program abends that are really caused by a macro (for example, 180 for an unresolved macro call, 405 for an invalid hex literal, SOC4 abend);
3. SAS errors caused by data, options, global macro variables, etc. that the macro inadvertently changed;
4. Not to mention logic errors in output or data caused by something the macro did, which may not even cause error messages.

This difference in the complexity of errors makes sense when you realize that the macro language is in fact a completely different language than base SAS, though the syntax is similar. It operates on a lower level than base SAS and therefore requires more programmer intervention and control.

The following section will help you to avoid each of these types of errors by writing macros defensively, so to speak. The subsequent section will help you debug those problems that slip through anyway.

STARTING OFF RIGHT: WRITING BETTER MACROS
An idea for a useful Macro is born. The company Macro whiz (the one who was sent to the course) sketches it out on paper, then sits down and types in the code for the Macro. The next step should NOT be to test the Macro. There are a few steps which should be taken between writing and testing which can greatly increase testing efficiency and lower cost.

Modularity
First, consider how the Macro is to be stored. Modularity is a popular way to create order in program writing in general, and it can be applied to Macro writing as well.

Complex Macros should be stored as separate entities, since they may (and ideally should) be used by more than one program or even more than one programmer (see Figure 1). A good rule is one module per Macro, and only the Macro definition in the module.

Even if a Macro is not to be used by others, another important reason for modularity is that it is much easier to test and debug a Macro as a separate entity. Rather than culling it out from your lengthy program for testing, or worse yet, running your entire program to test the Macro, it can be unit tested before it is put to use. In addition, code stored in this form is more easily reused and expanded as needs and programs change.
Know your environment

There is a lot of talk about "environments" when the subject of Macro comes up. This is not just another technical term used by Macro experts, but a vital aid to understanding how Macro works. One way to envision the concept is to think of a large program or system as a bunch of envelopes, some inside others, some just separate, in a large envelope which is the entire program (see Figure 2).

It is important to know what the total environment will look like during a given Macro's execution, because everything that is active in the "current processing environment" has the potential to affect the Macro's outcome. A good way to keep track of this is by knowing each Macro's inputs and outputs. Every Macro has obvious inputs, such as Macro parameters, and not-so-obvious ones, such as global Macro variables or Macro variables created in an environment which envelops the current Macro. Because the majority of Macros are created to be used in more than one program, it is important to note the variable names used in a Macro to avoid conflict with surrounding code.

It is also a good idea to use meaningful or unique names for variables which will only be used inside the Macro, or, better still, use the %LOCAL statement whenever possible to avoid ambiguity. %LOCAL is a very powerful means of avoiding logic problems. Using this statement, you can keep the variable names used in your Macro from affecting or being affected by the values of any global variables with the same name.

Macros have hidden outputs as well. A classic example is the %TESTPRINT Macro in Figure 3.

%macro testprint
(dsns=
, nobs=100);
%if &test=Y %then
%do; /* PRINT */
PROC PRINT DATA=&DSN
(OBS=&NOBS);
TITLE "&DSN TEST PRINT";
RUN;
%end; /* PRINT */
%mend testprint;

FIGURE 3: TESTPRINT MACRO

A hidden input to this Macro is the global variable &TEST. The Macro expects this variable to be available in the processing environment. A hidden output is the TITLE statement, which will wipe out any TITLE that may have been previously in effect. If this fact is not documented, it could lead to a nasty surprise for someone using the Macro.

Macros tend to grow and evolve as programmers find better ways to perform the task. For example, the Macro above could be improved by allowing the specification of title line to be used in the parameter list, so that in different programs only the parameter need be changed to avoid trashing TITLEs. Because of changes, the list of inputs and outputs to a Macro may change over time. It is a good idea, whenever possible, to keep track of the inputs and outputs, because they are an invaluable tool in testing. Knowing all of the inputs to a Macro allows you to completely separate the Macro unit and test by setting up all of the inputs and calling the Macro. For example, to test the %TESTPRINT Macro above a data set and a %LET TEST=Y or N are needed.

An organized list of inputs and outputs also makes the Macro more useful to others, because it will be a known quantity and will be "safe" to use. Researching the inputs and outputs should be a part of the analysis of any Macro. One way of
listing the inputs and outputs to a
Macro is shown in Figure 4.

After the %MACRO statement:

/* MACRO TESTPRINT
STORED IN [library]
CREATED BY [name]
CREATED ON [date]
LAST MODIFIED [date]

INPUTS:
&TEST (=Y or N)
&DSN
&NOBS

OUTPUTS:
TITLE is changed */

FIGURE 4:
SAMPLE INPUT/OUTPUT LIST

Checklist

Before doing any testing, you can
cut down on Macro syntax errors by
going through a simple eye-check.
Figure 5 shows a sample checklist.
You may be surprised at how much this
helps. Sometimes the simplest changes
such as changing the name of the Macro
can cause the most daunting errors
(for example, if you forget to change
the %MEND statement name); so don't
forget to use it when modifying as
well.

An additional aid in eye checking
which some find useful is to separate
Macro code from SAS code in some
visual way such as using lower case
for Macro code, as shown in Figure 3.
One problem with eye checking is that
a TO, for example, looks normal at a
glance even when it should be a XTO.
Identifying Macro code with lower case
helps prevent confusion of the two.

Preparation for Testing and
Debugging

When you have checked over the
code and are ready to test, be sure
the right options are in place to aid
you. See Figure 6 for a complete set
of options pertinent to Macro.

To see what your Macro actually
does, you have two choices of Macro
options: those that help you see the
SAS code generated, and those that
help you see the Macro code generated.
If your Macro contains mostly SAS
code, you will want to use the MPRINT
option. If there are complicated

* Is the Macro the only thing in
the module?
* Do the %MACRO and %MEND
statements appear correctly and
contain the same name?
* Do all %DOs and %ENDDO have a
match, and do they all start with %?
* What about %IFs, %THENs and
%ELSEs? %TOs for %DO loops?
* Do you have an appropriate step
break (PROC, DATA, or RUN) between the
assigning of any Macro variable values
using SYMPUT and their use?
* Do the variables used in your
%IFs and Macro functions contain any
"tricky" values and if so, have you
used proper quoting functions?
* If you use any index Macro
variables in loops, are they unique
from any regular variable names you
may be using inside the loop?
* Do all statements end with
semicolons?
* If you use any double or multiple
ampersands, do they plot out
correctly?
* Are all comments ended correctly?
* Do all of the Macro variables
used have meaningful names and %LOCAL
or %GLOBAL statements as appropriate?
(Note: Macro parameter variables do
not need to be specified as %LOCAL;
they automatically are.)
* Do you have any calls to other
Macros inside this one? If so, have
you checked out the inputs and
outputs? What are the inputs to this
Macro?

FIGURE 5:
MACRO SYNTAX CHECKLIST

Macro statements and functions, you
will probably be better off with
NOMPRINT MACROGEN SYMBOLGEN MLOGIC.
(Note MPRINT must be explicitly turned
off if it is the default or has been
previously turned on.) The two types
of options do not work together, so
you will have to make a choice as to
which will give you the most
information. Routinely test with
debugging options on, just in case of
error. You can change from one type
to the other in the middle if the
situation warrants; however, remember
to place a RUN; statement directly
after the options statement to put it
into effect immediately. In version
6, the MTRACE option is added and
further clarifies Macro code. For
the clearest picture, MPRINT should be
paired with MTRACE. Figure 7
demonstrates the type of log output
received using each option set.
CMDMAC allows command style Macros; should be off unless needed
DQUOTE allows Macro resolution inside double quotes; should be on
IMPLMAC allows implied Macros; should be off unless needed
MACRO must be on
MAUTOSOURCE allows compilation of Macros from a specified library without %INCLUDEs; should be off until Macro is completely tested; use %INCLUDEs with SOURCE2 to check for syntax errors
MCOMPILE, when off, introduces Macro compiler each time a Macro is referenced rather than only once for the entire job; should be on
MERROR should be on
MRECALL should be off
MLOGIC should be on
SERROR should be on
SOURCE and SOURCE2 are user's discretion, but for MPRINT to work, SOURCE must be on
OBS=0 should be avoided if using CALL SYMPUT; it does not happen when OBS=0

FIGURE 6:
OPTIONS PERTINENT TO MACRO

An extra help in testing and debugging is the %PUT statement. A system of standard %PUT statements at the beginning and end of each Macro definition can be a great help in pinpointing problem location and determining whether a Macro executed in a given run. A sample is shown in Figure 8.

WHEN YOU DO GET AN ERROR:
TESTING AND DEBUGGING TIPS

It is of course all well and good to plan the heavy analysis and preparation for Macro writing, but it doesn't always happen in practice. Even with the best of intentions, problems can slip through or develop later. In addition, at times programmers are left with predecessors' code which is already written and of course used none of these techniques. What is to be done when Macro code, especially in a large program, suddenly develops a strange bug?

\%let test=y;
\%options source mprint;
\%testprnt (dsn=cars)
\%put NOTE: *** MACRO TESTPRNT HAS BEGUN;
... code ...
\%put NOTE: *** MACRO TESTPRNT HAS ENDED;
\%mend testprnt;

\%macro testprnt;
\%options NOMPRINT MACROGEN SYMBOLGEN MLOGIC;
>Y
<&TEST=y
1-<MLOGIC>
%testprnt (dsn=cars)
+PROC PRINT DATA=CARS
1-%TESTPRNT
+CARS
2-ASYMBO
+(NOBS=NOBS)
1-%TESTPRNT
+100
2-ASYMBO
+1-%TESTPRNT
+TITLE
1-%TESTPRNT
>DSN TEST PRINT
2-<MLOGIC>
>CARS
4-ASYMBO
+"DSN TEST PRINT";
1-%TESTPRNT
+RUN;
1-%TESTPRNT
\%options NOMLOGIC NOMACROGEN
NOSYMBOLGEN MTRACE MPRINT;
\%testprnt (dsn=cars)
Beginning execution.
Parameter DSN has value CARS
\%if condition &TEST=Y is TRUE
PROC PRINT DATA=CARS
(NOBS=100);
TITLE "CARS TEST PRINT";
RUN;

FIGURE 7:
LOG RESULTS OF MACRO OPTIONS

\%macro testprnt;
\%options source mprint;
\%testprnt (dsn=cars)
PROC PRINT DATA=CARS (OBS=100);
TITLE "CARS TEST PRINT";
RUN;

\%options NOMPRINT MACROGEN SYMBOLGEN MLOGIC;
>Y
2-ASYMBO
<&TEST=y
1-<MLOGIC>
%testprnt (dsn=cars)
+PROC PRINT DATA=&DSN
1-%TESTPRNT
+CARS
2-ASYMBO
+(NOBS=&NOBS)
1-%TESTPRNT
+100
2-ASYMBO
+1-%TESTPRNT
+TITLE
1-%TESTPRNT
>DSN TEST PRINT
2-<MLOGIC>
>CARS
4-ASYMBO
+"DSN TEST PRINT";
1-%TESTPRNT
+RUN;
1-%TESTPRNT
\%options NOMLOGIC NOMACROGEN
NOSYMBOLGEN MTRACE MPRINT;
\%testprnt (dsn=cars)
Beginning execution.
Parameter DSN has value CARS
\%if condition &TEST=Y is TRUE
PROC PRINT DATA=CARS
(NOBS=100);
TITLE "CARS TEST PRINT";
RUN;

FIGURE 8:
STANDARD %PUTS
The three main causes of Macro errors are:

1. Simple coding mistakes
2. Nonexistent or unexpected Macro variable values
3. Unforeseen changes to the processing environment caused by a Macro.

Coding mistakes

Always start with the obvious. The Appendix of this paper has hints on where to look for problems when you receive a Macro error message.

Unexpected values

Figure 9 shows two examples of unexpected values. Remember that Macro is a text processor, but unshielded keywords will always be seen as keywords. You might say that everything is out in the open in Macro, whereas you can shield values in SAS with quotes. The Macro quoting functions can solve these problems, as shown in Figure 10.

```macro
%let a=CARY,NC;
%let b=%scan(&a,l);
ERROR 1550: REQUIRED OPERATOR NOT FOUND.

What Macro processor saw:
%let b=%scan(cary,nc,l);
%let instnm=SCHOOL OF SCIENCE AND TECHNOLOGY;
%if &instnm ne %str() %then
    %put &instnm;
    ERROR 1555: CHARACTER OPERAND FOUND WHERE NUMERIC REQUIRED.

What Macro processor saw:
%if SCHOOL OF SCIENCE AND TECHNOLOGY ne %str() %then
    %put &instnm;
```

FIGURE 9: UNEXPECTED VALUES

Unforeseen changes

A good example of this phenomenon is the "ubiquitous &1". Say that one Macro contains a large %DO loop with a lot of processing, inside of which a smaller Macro is executed. This Macro also contains a %DO loop. Both use &1 as the counter. %LOCAL statements have not been used. Figure 11 shows the effect and results.

```macro
%do i=1 %to 3;
    < code >
%end;
```

Consider the opposite effect:

```macro
%do i=1 %to 4;
    %do i=1 %to 2;
    < code >
    %end;
%end;
```

The loop is endless.

FIGURE 11: THE UBIQUITOUS &1

Debugging

If the problem is an obvious coding mistake, it can be fixed where it is. However, if it appears to be of the second or third variety, do not waste time and energy trying to debug the entire program. Instead, follow these steps:

1. Isolate
2. Replicate
3. Apply solution(s)
4. Reinstate

These steps are discussed in detail below.

5
Isolate

The first task is to pinpoint exactly where in the program the error is occurring. If the debugging options are not on, turn them on, or use %PUTs to track the statements. If you use standard %PUTs to signify the beginning and end of each Macro, you may be able to locate the offending Macro immediately. Note that as soon as an error is encountered in a given Macro, processing "skips" out to the next environment without executing any more statements in that Macro. Look for the last statement that was executed before the skip.

The idea is to find the smallest piece of code that will still trigger the error. Then you can remove that piece (a Macro module, or even part of one) and work on it separately. Interactive SAS is good for this.

Replicate

This is probably the most important step, and the one most often forgotten. If you forget this step, you may be running in circles without solving the real problem.

Using your input list and your knowledge of the program and data, manipulate the inputs to the Macro until you have replicated the error. If you can't replicate the error, you know it's either a bigger problem than just the portion you have isolated, and the real bug is probably prior to this part, and you need a larger chunk, or you have input values you don't know about.

Apply solution(s)

Once you've replicated the problem, congratulations -- the hard part is over. Ten to one you'll now know why the problem occurred, or if not why, at least what data value or statement caused it. Now all you have to do is fiddle with the quoting functions, syntax, values, or whatever needs to be changed. If the code worked with one input value and not with another, be sure to test both ways to make sure your fix hasn't merely created a new bug.

Reinstate

When you are satisfied that it works, reinstate the module and test the entire program. Note that sometimes the actual change to be made is elsewhere than the module that was tested.

CONCLUSION

Macro is a rewarding and powerful tool which provides users with a quantum leap in their ability to perform complex tasks using SAS. Macro errors encountered tend to grow in complexity as the user's ability grows, but the same rules apply no matter how complicated the code or obscure the error. Remember that the very characteristics which make Macro seem so daunting and hard to control are those which give it its amazing power. All it takes to tame the wild Macro is a little organization at the start and the right tools along the way.

APPENDIX:
MACRO ERROR REFERENCE

Common Macro Errors

1301: APPARENT SYMBOLIC REFERENCE NOT RESOLVED
Check Macro variables:
* Macro variable name misspelled
* Macro variable created in local environment and referenced outside that environment
* Macro variable created by CALL SYMPUT and referenced in same step
* Variable created by CALL SYMPUT when OBS=0

1353: APPARENT MACRO INVOCATION NOT RESOLVED
Check Macro calls:
* Macro name misspelled
* Macro referenced before it is compiled

1550: REQUIRED OPERATOR NOT FOUND and 1555: CHARACTER OPERAND FOUND WHERE NUMERIC REQUIRED
Check %DO loops and %IF-%THEN statements:
* % signs missing on keywords
* Macro variable used in statement has character (or decimal) value
* Macro variable unresolved or resolved incorrectly inside statement

Macro Error Groupings

%DO loops: 1100, 1124, 1128, 1135, 1550, 1555
%IFS: 1115, 1116, 1120, 1404, 1405, 1550, 1552, 1555
%GOTOs: 1114, 1139, 1402, 1408
Parameters/Macro definitions: 1102, 1104, 1105, 1107, 1125, 1134, 1135, 1353, 1452, 1454, 1455, 1456
%LETs: 1302, 1350, 1500
Open error on Macro: 1101, 1354, 1355, 1400
%GLOBAL: 1501
%EVAL: 1555
MWORK option too small: 1300, 1600, 1603, 1653

SAS Error Messages That Can Be Caused By Macro

ERROR: UNEXPECTED END OF FILE
* %INCLUDED code wrong LRECL
180: STATEMENT IS NOT VALID
* Unrecognized Macro variable or call
405: INVALID NUMERIC HEX LITERAL
* Macro variable whose value contains a valid hex literal is specified where a %EVAL or implied %EVAL is done
107: CHARACTER LITERAL IS TOO LONG
* A single quote appears in a Macro variable value

REFERENCES

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

The author may be contacted at:
Computer Data Systems, Inc.
20010 Century Boulevard
Germantown, MD 20874
Telephone: (301) 363-5729