WARNING: APPARENT MACRO INVOCATION NOT RESOLVED . . . TECHNIQUES FOR DEBUGGING MACRO CODE

Jeff Phillips, ARC Professional Services Group
Veronica Walgamotte, ARC Professional Services Group
Derek Drummond, ARC Professional Services Group

INTRODUCTION

The macro facility is a powerful adjunct to The SAS® System. Macro allows SAS programmers to employ such techniques as symbolic substitution, conditional code generation, hierarchical application structure, and parameter-driven, reusable code. In spite of macro's power and usefulness, it often seems to generate fear and loathing rather than interest and excitement. There are two major barriers to wider use and acceptance of macro as a programming tool in the SAS world. New users are often unsure of exactly what it is that macro does, while beginning and even intermediate users find debugging macro errors to be baffling. Many papers and tutorials have ben written and presented describing just what macro can do. This workshop attempts to address the second problem: debugging. We will consider the most common macro problem types and illustrate their symptoms, causes, and corrections.

WHY MACRO DEBUGGING IS DIFFICULT

Problems occurring when using the macro facility are especially difficult to identify and correct because of the special role macro plays in SAS processing. Macro is a complete programming language, with its own variables, statements, functions, data structures, and syntax rules. However, macro is used primarily to generate and alter code in another programming language, SAS. Macro is in essence a language used to write another language. When a programmer makes a mistake using macro, that mistake can affect the macro code itself, the SAS code that is being generated, or, most often, both. An error that results from a macro mistake appears to be a SAS error. The programmer must distinguish between genuine SAS syntax errors and errors caused by incorrect macro code. The situation is made worse by the fact that macro and SAS have similar syntax. For example, the macro statement:

```
%IF &STATE = VA %THEN %PUT The State is
Virginia.;
```

is not very different from the SAS statement:

```
IF STATE = 'VA' THEN PUT 'The State is
Virginia.';
```

CLASSIFYING THE PROBLEM

The first step in debugging a macro problem is to figure out the type of problem. Typically, macro errors fall into one of the following categories:

- *Macro syntax errors* are caused by incorrect spelling, word order, or special character use in a macro statement.
- *Resolution errors* are caused when macro variables fail to resolve, or resolve to unexpected values.
- *SAS code construction errors* are caused when macro structures, especially macro variables, are improperly placed in SAS code.
- *SYMPUT and SYMGET errors* are caused by improper use of the two DATA step functions CALL SYMPUT and SYMGET to create and retrieve macro variables.
- *Quoting function errors* are caused when a particular special character (such as a semicolon) or a string has an ambiguous meaning to macro.

MACRO SYNTAX ERRORS

As in any programming language, the most basic type of error is incorrect syntax. Syntax errors are often caused by "typos," but the resulting error message may appear far more complex. Consider the following example:

```
        ERROR: Open code statement recursion
detected.
4     %PUT The following statement is a LET
statement:
5     %LET VAR = VALUE;
```

The error message "open code recursion" means that macro has detected a macro statement keyword, %LET, inside another macro statement, %PUT. The problem is caused by the accidental use of a colon instead of a semicolon at the end of the %PUT. The problem is a simple one; the identification is not.

Hints and rules: *In Version 6 of SAS, macro error messages have gotten much better than in "the old days". Since the context of the mistake determines the effect it will have on the code's interpretation, it is difficult to give rules of thumb for debugging. One rule, however, may prove helpful: in cases where the problem is not immediately apparent, the actual error may have occurred in the statement before the one flagged by the error message.*

RESOLUTION ERRORS

The second type of macro problem occurs when dealing with the most basic of macro functions: symbolic resolution. There are two "bad" things that can happen when macro attempts to resolve a macro variable reference: incorrect resolution and no resolution. The symptoms and treatments for each are quite different. In the following SAS log, the ERROR: message says that SAS is " expecting an relational or arithmetic operator."

```
99   %let state = VA;
100  data virginia;
101     length city $ 11;
102     set workshop.states;
103     if state = &stat;
                       -
                       390
WARNING: Apparent symbolic reference STAT not
resolved.
104  run;

ERROR 390-185: Expecting an relational or
arithmetic operator.

NOTE: The SAS System stopped processing this
step because of errors.
WARNING: The data set WORK.VIRGINIA may be
incomplete.  When this step was stopped there
were 0 observations and 4 variables.
NOTE: The DATA statement used 0.17 seconds.
```

The real problem is identified by the WARNING message above it, "apparent symbolic reference STAT not resolved." Many programmers ignore warnings because they do not stop the execution of the code. In this case, however, the warning showed that a macro variable, &STAT, could not be resolved. The unresolved string &STAT is passed to the SAS compiler, which evaluates the & as the symbol for logical "and." The unexpected operator message, caused by the misplaced "and," is only an indirect reference to the real problem, which was the misspelled macro variable name. There are two valuable lessons here: first, one should never ignore warnings in the SAS log, especially if a macro component is involved. Second, when dealing with SAS code containing macro, the programmer should be aware that any SAS error message may be a tipoff that macro has generated an incorrect string.

The following simple macro will illustrate a more subtle problem:

```
%macro subset (varname =,
               value = );
data &value;
   set workshop.states;
   if &varname = "&value";
run;
%mend subset;
```

The purpose of the macro is to allow the user to select both the variable name and value for a subsetting IF condition. A macro call of:

```
%SUBSET(VARNAME=STATE,VALUE=LA)
```

would yield the following DATA step:

```
DATA LA;
SET WORKSHOP.STATES;
IF STATE = "LA";
RUN;
```

However, it takes only a small error to create the following result:

```
132  %let varname = state;
133  %let value = LA;
134  %subset
135  proc print;

NOTE: The data set WORK.DATA1 has 4
observations and 4 variables.
NOTE: The DATA statement used 0.11 seconds.


136   title "Test of SUBSET Macro";
137  run;

NOTE: The PROCEDURE PRINT used 0.05 seconds.

Test of SUBSET Macro

OBS       NAME        STATE     FLOWER
 IF

 1      Louisiana      LA       MAGNOLIA
 2      Maryland       MD       BLACK-EYED SUSA
 3      New York       NY       ROSE
 4      Virginia       VA       DOGWOOD
```

What has happened to eliminate any subsetting criteria? How and why did SAS create a variable called "IF," whose value is missing? Why is the output data set called "DATA1"?

When the macro %SUBSET was called, the user failed to specify any value for either of the two parameters VARNAME and VALUE. Since these macro variables have been declared (they have been established during the macro's definition), but have not been given any explicit values, they will resolve to null. The DATA step then becomes:

```
MPRINT(SUBSET):    DATA ;
MPRINT(SUBSET):    SET WORKSHOP.STATES;
MPRINT(SUBSET):    IF = "";
MPRINT(SUBSET):    RUN;
```

These statements give us a data set called WORK.DATA1, no subsetting, and a new variable called "IF".

Such errors often occur when macro calls are involved, since macro parameters are *local*, or isolated within the macro that defines them. The example illustrates how the interaction between local parameters and variable created by %LET statements can cause a phenomenon known as *blocking*, when an executing macro cannot access variables assigned by %LET statements outside the macro. Blocking occurs because macro parameters are local and "get in the way" of the variables with the same name that are set outside the macro by %LET statements.

Hints and rules: (1) *Identify what happened: incorrect resolution or no resolution.* (2) *The warning message, "apparent symbolic reference not resolved" suggests that a macro variable cannot be resolved at all, as opposed to resolving incorrectly. Look first for misspellings and typographic errors.* (3) *Examine the generated SAS code and output carefully. In the previous examples, resolution errors were indicated by such clues as the presence of a variable called "IF" in a data set and the data set name "WORK.DATA1."* (4) *Use the SYMBOLGEN and MPRINT options when writing and debugging macro code. Otherwise, the log may not show enough information to help adequately in debugging.* (5) *Be especially careful when using macros. They may have local variables that interfere with access to* *variables set by %LET earlier in the program.*

SAS Code Construction Errors

One of the most basic errors in macro programming occurs when the programmer fails to pay proper attention to the way a code string generated by macro will appear to SAS itself. Consider the following SAS log:

```
145  %let flower = MAGNOLIA;
146  data flowers;
147     set workshop.states;
148     if flower = &flower;
149  run;

NOTE: Variable MAGNOLIA is uninitialized.
NOTE: The data set WORK.FLOWERS has 0
observations and 4 variables.
NOTE: The DATA statement used 0.11 seconds.


150  proc print data = flowers;
151     title 'States With &flower as State
Flower';
152  run;

NOTE: No observations in data set WORK.FLOWERS.
NOTE: The PROCEDURE PRINT used 0.0 seconds.
```

There is nothing wrong with the *syntax* in either the macro or the SAS portions of the example (that is, no SAS syntax errors are generated). The incorrect results and the "UNINITIALIZED VARIABLE" message both indicate that there is something logically wrong. Here, the problem is the absence of quotation marks around the reference to &FLOWER in the DATA step. The SAS compiler interprets the resolved value of &FLOWER, which is MAGNOLIA, as if it were a variable name. Such an interpretation is correct in SAS terms because the string "MAGNOLIA" begins with a letter, contains only letters, has no blank spaces, and is less than eight characters long. In the SAS lexicon, that description fits what is called a "name token," and in the particular context of this DATA step, the name token MAGNOLIA looks like a variable name to SAS. Since the goal is to pass in a character constant to SAS, the programmer must add quotation marks:

```
153  %let flower = MAGNOLIA;
154  data flowers;
155     set workshop.states;
156     if flower = "&flower";
157  run;

NOTE: The data set WORK.FLOWERS has 1
observations and 3 variables.
NOTE: The DATA statement used 0.11 seconds.


158  proc print data = flowers;
159     title 'States With &flower as State
Flower';
160  run;

NOTE: The PROCEDURE PRINT used 0.05 seconds.
```

A similar problem may arise if the wrong type of quotation marks are used:

```
States With &flower as State Flower

OBS      NAME        STATE     FLOWER

 1     Louisiana      LA      MAGNOLIA
```

Here, the single quotation marks prevent the resolution of the macro variable &FLOWER. The SAS compiler detects no syntax problems. However, when the PROC PRINT step executes, the title contains &FLOWER instead of MAGNOLIA. A debugging clue to the problem can be gained if SYMBOLGEN is invoked before the DATA step. If SYMBOLGEN is in effect, every appearance of &FLOWER in the DATA step should cause a message. Notice that no message is generated for the &FLOWER reference inside the TITLE statement:

```
162  %let flower = MAGNOLIA;
163  data flowers;
164     set workshop.states;
165     if flower = "&flower";
SYMBOLGEN:  Macro variable FLOWER resolves to
MAGNOLIA
166  run;

NOTE: The data set WORK.FLOWERS has 1
observations and 3 variables.
NOTE: The DATA statement used 0.16 seconds.


167  proc print data = flowers;
168     title 'States With &flower as State
Flower';
169  run;

NOTE: The PROCEDURE PRINT used 0.0 seconds.
```

Changing the single quotes to double quotes will correct the problem, since double quotes are processed so they do not obscure the macro variable reference inside them.

Hints and rules: (1) *This is not a diagnosis rule, but it will eliminate many quote-mark related errors: always use double quotes when writing SAS code. If double quotes are the standard character literal symbols, then the chances of a macro variable being undetected are greatly reduced.* (2) *Pay attention to any "uninitialized variable" or similar messages. Each should be explainable; the cause could be a macro variable reference that should be inside quotes.*

SYMPUT and SYMGET Errors

The SYMPUT routine in the DATA step can lead to errors that are caused by the interaction of the macro facility and SAS. To understand these interaction errors, one must keep in mind that the DATA step has two distinct phases: compilation and execution. During compilation, SAS constructs its object code stream from the programmer's source code plus whatever additional information may come from the macro facility. The following code is an example:

```
%LET FLOWER = ROSE;
DATA &FLOWER;
  SET WORKSHOP.STATES;
  IF FLOWER = "&FLOWER";
RUN;
```

As the step compiles, the source code stream is examined, token by token, by a SAS system component called the wordscanner. Standard SAS code tokens are sent by the wordscanner to the SAS compiler to be checked for syntax and to construct the step's object code and data vectors. During the scanning and compilation process, any tokens containing macro triggers (& and %) are sent to the macro facility for processing. While the SAS compiler waits, macro examines the token(s) passed to it, executes whatever instructions are contained in them, and then sends the result back to the wordscanner. For example, in the previous code the token &FLOWER is sent to the macro facility and macro resolves it to ROSE. ROSE then replaces &FLOWER in the code stream so that the code reads:

```
DATA ROSE;
  SET WORKSHOP.STATES;
  IF FLOWER = "ROSE";
RUN;
```

The original source code (&FLOWER) has been changed or augmented (ROSE) by means of macro activity. A key to this process is that it happens during the scanning and compilation phase, while SAS is building the code that will be executed. The limitation on the process is that once the SAS code begins to execute, there is no way for macro activity to occur. For example, if a programmer wishes to assign or obtain the value of a macro variable based on a SAS data value, %LET and & are of no use. Another example will help illustrate the point:

```
190  data ivy;
191    set workshop.states;
192    if flower =  "IVY"  then
193    do;
194        %let found = YES;
195        output;
196    end;
197   run;

NOTE: The data set WORK.IVY has 0 observations
and 3 variables.
NOTE: The DATA statement used 0.05 seconds.

SYMBOLGEN:  Macro variable FOUND resolves to YES
198
199    %PUT FOUND = &FOUND;
FOUND = YES
```

This DATA step will *always* set the macro variable FOUND to YES, even if there are no observations containing "IVY" in the entire WORKSHOP.STATES data set. The %LET will be sent to macro to execute when the wordscanner detects it during compilation, despite the IF statement.

CALL SYMPUT allows the DATA step to communicate with macro during the execution phase. In effect, CALL SYMPUT is a %LET statement that operates under the control of DATA step logic. Thus, the following code would be appropriate, since CALL SYMPUT is a SAS routine, and executes under the control of DATA step program logic:

```
211  %let found = no;
212  data ivy;
213    set workshop.states;
214    if flower =  "IVY"  then
215    do;
216        call symput("found","YES");
217        stop;
218    end;
219  run;

NOTE: The data set WORK.IVY has 4 observations
and 3 variables.
NOTE: The DATA statement used 0.22 seconds.


SYMBOLGEN:  Macro variable FOUND resolves to
no
220
221   %put FOUND = &found;
FOUND = no
```

Problems occur when the programmer forgets the difference between macro statements and resolutions, which happen during compilation phase, and CALL SYMPUT, which happens *later*, during the execution phase. The following code is similar to the previous examples. Here, however, the programmer attempts to use the value of the macro variable &FOUND to write a message to the SAS log about the existence of roses in the data:

```
251  %let found = NO;
252  data roses;
253    set workshop.states;
254    if flower =  "ROSE"  then
255    do;
256        call symput("found","YES");
257        output;
258    end;
259    %PUT Are there roses? The answer is
&found.;
Are there roses? The answer is NO
260  run;

NOTE: The data set WORK.ROSES has 1
observations and 3 variables.
NOTE: The DATA statement used 0.16 seconds.

261
262  proc print data = roses;
263  title "ROSES Data Set";
264  run;

NOTE: The PROCEDURE PRINT used 0.05 seconds.
```

As the log shows, the answer is always "NO," even when the log notes prove that there is at least one state with the rose as a state flower. The %PUT statement is executed by the macro processor while the DATA step code is still being compiled. The RUN; statement that trigger execution of the DATA step is not encountered until after the %PUT is finished.

The simplest solution to the problem is to make sure that the step containing a CALL SYMPUT has executed before attempting to use the macro variable the CALL SYMPUT assigns:

```
265  %let found = NO;
266  data roses;
267    set workshop.states;
268    if flower =  "ROSE"  then
269    do;
270        call symput("found","YES");
271        output;
272    end;
273  run;

NOTE: The data set WORK.ROSES has 1
observations and 3 variables.
NOTE: The DATA statement used 0.17 seconds.


274
275  %PUT Are there roses? The answer is
&found.;
```

```
Are there roses? The answer is YES
```

Hints and rules: (1) *Look for missing RUN; statements or other step boundaries.* (2) *Be suspicious of %LET, or any other macro statement, inside a DATA step. There are no situations in which %LET can be controlled by the DATA step; use CALL SYMPUT when the intent is to control the assignment of macro variables with DATA step logic.* (3) *Be equally suspicious of macro variables being both assigned and resolved in a single DATA step.*

Quoting Function Errors

Normally, the macro facility interprets everything as a character string. There are no data types in macro; "1.25" is as much a character string as "ABC." There are situations, however, in which macro must attach special meaning to characters. For example, a semicolon ends a macro statement. Therefore, it is not just any character. Sometimes, though, a semicolon is part of SAS text, and is not meant to be of any special significance to macro. Consider the following code:

```
17   %let sasstep = proc print
data=workshop.states;
18   &sasstep
19   title  This PRINT Was Generated By A Macro
Variable ;
19   title  This PRINT Was Generated By A Macro
Variable ;
     -----
     76
ERROR 76-322: Syntax error, statement will be
ignored.

20   run;

NOTE: The SAS System stopped processing this
step because of errors.
NOTE: The PROCEDURE PRINT used 0.11 seconds.
```

As the SAS log shows, there is a semicolon missing, which causes the TITLE statement to be looked at as part of the PROC PRINT statement. The error, however, is a macro error, not a SAS code error. The semicolon after "MYDATA" is meant to be part of the SAS code stream being stored as the value of the macro variable SASSTEP. Macro interprets the semicolon as signaling the end of the %LET statement, and so the semicolon does not get passed on to the compiler as was intended.

Problems such as the one just discussed arise when macro cannot accurately identify a given special character or character string. Such problems are generally called situations of *ambiguous meaning*. Ambiguous meaning often arises because SAS and macro share symbols such as the semicolon, or because a special character or character string within a piece of code could be taken two ways. Macro quoting functions, such as %STR, serve to eliminate the ambiguity by forcing macro to interpret the string being quoted as text and nothing more. Using %STR to the previous example eliminates any ambiguity caused by the presence of the semicolons:

```
13   %let sasstep = %str(proc print
data=workshop.states;);
14   &sasstep
15   title  This PRINT Was Generated By A Macro
Variable ;
16   run;
```

```
NOTE: The PROCEDURE PRINT used 18.55 seconds.
```

Another example of ambiguous meaning will serve to illustrate better the type of error caused by quoting problems:

```
22   %macro subset (state = );
23   data subset;
24      set workshop.states;
25      %if &state = NE or &state = OR %then
26         %do;
27            delete;
28            %put No data for Nebraska or
Oregon.;
29         %end;
30      %else
31         %do;
32      if state = "&state";
33         %end;
34   run;
35   %mend subset;
36   %subset (state = VA);

ERROR: A character operand was found in the
%EVAL function or %IF
      condition where a numeric operand is
required. The condition was:
      &state = NE or &state = OR
ERROR: The macro will stop executing.
```

A problem results when the macro is compiled by the macro facility because there is obvious ambiguity in the phrase "&STATE = NE OR &STATE = OR. "NE" and "OR" can be seen as either state abbreviations or as logical operators. The log shows that macro is trying to evaluate an expression, in this case the expression in the %IF statement. As a rule, such an error message will occur when syntactical mistakes are made in macro %IF statements, %DO loops, or arithmetic expressions. However, there are times when the problem is an ambiguity rather than an outright error. The problem can be solved by using %STR:

```
54   %macro subset (state = );
55   data subset;
56      set workshop.states;
57      %if %str(&state) = %str(NE) or
%str(&state) = %str(OR) %then
58         %do;
59            delete;
60            %put No data for Nebraska or
Oregon.;
61         %end;
62      %else
63         %do;
64      if state = "&state";
65         %end;
66   run;
67   %mend subset;
68   %subset (state = OR);

NOTE: The data set WORK.SUBSET has 4
observations and 3 variables.
NOTE: The DATA statement used 1 minute 18.82
seconds.


No data for Nebraska or Oregon.

NOTE: The data set WORK.SUBSET has 0
observations and 3 variables.
NOTE: The DATA statement used 0.22 seconds.
```

The macro now works properly because macro has been "told" that NE and OR are to be treated as plain text, not as operators. Note that %STR must be used on both the constants NE and OR as well as the references to &STATE. When using quoting functions, the programmer must consider that macro variables may *resolve into* problems that are not immediately apparent. In this example, "&STATE" is not a problem as it stands, but when it resolves into "OR," it can cause an error if %STR is not in place.

Note: in SAS releases prior to 6.08, the %QUOTE function must be used in place of %STR to quote macro variable resolutions. In the example above, the code:

```
        %str(&state)
```

is incorrect for releases 6.07 and prior (including 6.04 for DOS). The correct code would be:

```
        %quote(&state)
```

In pre-6.08 SAS releases, The %STR function does not quote text resulting from either macro executions or macro variable resolutions. In 6.08, %STR has been enhanced.

Hints and rules: (1) *Examine macro syntax containing %IF, %DO, %SCAN (second function argument), %SUBSTR (second and third function arguments), or %EVAL. These macro activities involve logical and/or arithmetic evaluation, and are more "sensitive" to the meaning of symbols and character strings than statements such as %LET.* (2) *Examine areas in which macro statements are directly generating SAS code, such as the %LET SASSTEP ... &SASSTEP example above. SAS and macro share the semicolon, colon, period, comma and parenthesis, and these shared special characters may lead to ambiguous meaning.*

CONCLUSION

There is no question that macro adds a level of debugging complexity to SAS programs. However, macro problems are often caused by the same types of simple mistakes, such as typographical miscues, that cause SAS errors. When the causes of macro problems are not immediately obvious, they can be attacked by carefully observing the symptom and following the diagnosis hints and rules presented in this workshop.

The authors can be contacted at:

ARC Professional Services Group
Information Systems Division
1301 Piccard Drive
Rockville, MD 20850

(301) 258-5300
FAX: (301) 258-6878