Paper 147-2010

SAS® System Options: The True Heroes of Macro Debugging

Kevin Russell and Russ Tyndall, SAS Institute Inc., Cary, NC

ABSTRACT

It is not uncommon for the first draft of any macro application to contain errors. However, without debugging options, it is almost impossible to locate and correct all the errors in an application. Therefore, the first step in writing a macro application is to turn on the debugging options. This paper is intended for programmers at all levels who write macro code, and it concentrates primarily on three SAS system options that you can use to debug macros: MLOGIC, SYMBOLGEN, and MPRINT (our superheroes). In addition, this paper includes some less common options (the superhero sidekicks) that also display valuable information in the SAS log. These options can help you determine the source of many errors that can occur when you program with the macro language.

INTRODUCTION

The macro facility is a great tool for reducing the amount of text that you must enter to complete common tasks. It also enables you to extend and customize SAS. With this type of control, sometimes the macro language can be difficult to understand and use. Macros can be complicated due to the large amounts of code that they generate. In addition, multiple macros can be invoked together, which makes the code even more complicated.

As previously stated, it is not uncommon for the first draft of any macro application to contain errors. When you are trying to debug your macro application, these errors can seem like villains, obstacles that you must overcome in order to achieve peace and harmony in your application. The good news is that where there are villains, there are also superheroes, and these heroes come in the form of macro-debugging SAS system options. Macro-debugging options provide the information that you need in order to determine the cause of a problem with your macro application. Without debugging options, it is almost impossible to locate and correct all the errors. Therefore, the first step in writing a macro application, and thus confounding your villains, is to turn on the debugging options.

All programmers who write macro code, from novice to expert levels, can use the macro-debugging SAS system options described in this paper to combat villains in their own applications. The paper discusses three superhero system options—MLOGIC, SYMBOLGEN, and MPRINT—and their super sidekicks—MFILE, MLOGICNEST, MPRINTNEST, MAUTOLOCDISPLAY, and MCOMPILENOTE. Examples that show how to use these options are also included.

Not every situation will require that you use all the macro-debugging system options that are mentioned in this paper. However, to truly understand what is happening when developing a macro, MLOGIC, SYMBOLGEN, and MPRINT should *always* be used. Because of the importance of these superheroes, the majority of this paper focuses on these SAS system options and the information they provide.

By the end of this discussion, you will have a better understanding of how to use these SAS system options as tools to make macro programming and macro code debugging much easier.

THE SUPERHEROES

MLOGIC

MLOGIC provides the most debugging information of all the superhero system options. With six different elements of debugging, MLOGIC's powers enable it to complete the following super tasks:

- Mark the beginning of macro execution.
- Show the values of macro parameters at invocation.
- Show the execution of each macro program statement.
- Show whether a %IF condition is **TRUE** or **FALSE .**
- Mark the end of macro execution.
- Show the location of an invoked AUTOCALL macro.

Determining when macro execution begins and ends is very helpful when nested macros are involved. Here is an example:

```
   /**Code**/
%macro demo;
   %let val=abc;
   %let len=test;
   %let temp=%substr(&val,1,%length(&len));
%mend;

%macro demo2;
   %let val=abc;
   %let len=tst;
   %let temp=%substr(&val,1,%length(&len));
%demo;
%mend demo2;
%demo2

   /**Log**/
3104  %macro demo;
3105     %let val=abc;
3106     %let len=test;
3107     %let temp=%substr(&val,1,%length(&len));
3108  %mend;
3109
3110  %macro demo2;
3111     %let val=abc;
3112     %let len=tst;
3113     %let temp=%substr(&val,1,%length(&len));
3114  %demo;
3115  %mend demo2;
3116  %demo2
WARNING: Argument 3 to macro function %substr is out of range.
```

There is a warning in the preceding code, but how do you know to which macro the warning relates? It's impossible to know this information without help. Here comes the SAS system option MLOGIC to the rescue! Let's see what happens when the same code is run with MLOGIC turned on:

```
   /**Log**/
3117  options mlogic;
3118  %macro demo;
3119     %let val=abc;
3120     %let len=test;
3121     %let temp=%substr(&val,1,%length(&len));
3122  %mend;
3123
3124  %macro demo2;
3125     %let val=abc;
3126     %let len=tst;
3127     %let temp=%substr(&val,1,%length(&len));
3128  %demo;
3129  %mend demo2;
3130  %demo2
MLOGIC(DEMO2):  Beginning execution.
MLOGIC(DEMO2):  %let (variable name is VAL)
MLOGIC(DEMO2):  %let (variable name is LEN)
MLOGIC(DEMO2):  %let (variable name is TEMP)
MLOGIC(DEMO):  Beginning execution.
MLOGIC(DEMO):  %let (variable name is VAL)
MLOGIC(DEMO):  %let (variable name is LEN)
MLOGIC(DEMO):  %let (variable name is TEMP)
WARNING: Argument 3 to macro function %substr is out of range.
MLOGIC(DEMO):  Ending execution.
MLOGIC(DEMO2):  Ending execution.
```

Because MLOGIC was specified, the log shows that the macro DEMO was executing at the time of the warning, and that shows us which instance of %SUBSTR caused the warning. The code also shows the execution of each %LET statement along with the macro variable name that is created.

%IF EVALUATION

Another benefit of engaging MLOGIC's powers is that you can determine whether each %IF condition is **TRUE** or **FALSE**. Here is an example:

```
   /**Code**/
options nomlogic;
%macro demo(val);
   %if &val eq 'test' %then %put it worked;
   %else %put it did not work;
%mend;

%demo(test)

   /**Log**/
3131  options nomlogic;
3132  %macro demo(val);
3133     %if &val eq 'test' %then %put it worked;
3134     %else %put it did not work;
3135  %mend;
3136
3137  %demo(test)
it did not work
```

Because &VAL resolves to **`test`**, you might have thought the %IF statement should be **TRUE**; instead, the %ELSE portion is executed. Consider the same example, but this time with MLOGIC set.

```
   /**Log**/
3138  options mlogic;
3139  %macro demo(val);
3140     %if &val eq 'test' %then %put it worked;
3141     %else %put it did not work;
3142  %mend;
3143
3144  %demo(test)
MLOGIC(DEMO):  Beginning execution.
MLOGIC(DEMO):  Parameter VAL has value test
MLOGIC(DEMO):  %if condition &val eq 'test' is FALSE
MLOGIC(DEMO):  %put it did not work
it did not work
MLOGIC(DEMO):  Ending execution.
```

With MLOGIC set, you see that the %IF condition is **FALSE**; therefore, the %ELSE executes. Remember, in macro language programming, you are comparing text to text. Therefore, this example compares the literal word "test" to the string **`'test'`**. In this case, the %IF statement should be coded as follows:

```
%if &val eq test %then %put it worked;
```

This way compares the literal word "test" to the literal word "test". Also notice in this sample code that MLOGIC provides another feature by giving you the value of the parameter that is passed to the macro DEMO.

AUTOCALL LOCATION

A lesser-known power of MLOGIC is its ability to reveal the location of an AUTOCALL macro. Here is an example:

```
   /**Code**/
options mlogic;
%macro demo(str);
   %let val=%str(&str    );
   %put **%trim(&val)**;
%mend;

%demo(123)

   /**Log**/
3200  options mlogic;
```

```
3201  %macro demo(str);
3202     %let val=%str(&str    );
3203     %put **%trim(&val)**;
3204  %mend;
3205
3206  %demo(123)
MLOGIC(DEMO):  Beginning execution.
MLOGIC(DEMO):  Parameter STR has value 123
MLOGIC(DEMO):  %let (variable name is VAL)
MLOGIC(DEMO):  %put **%trim(&val)**
MLOGIC(TRIM):  Beginning execution.
MLOGIC(TRIM):  This macro was compiled from the autocall file
                    C:\Program Files\SAS\SAS 9.1\core\sasmacro\trim.sas
MLOGIC(TRIM):  Parameter VALUE has value _123    _
MLOGIC(TRIM):  %local  I
MLOGIC(TRIM):  %do loop beginning; index variable I; start value is 7; stop value is
                   1; by value is -1.
MLOGIC(TRIM):  %if condition %qsubstr(&value,&i,1)^=_ _ is FALSE
MLOGIC(TRIM):  %do loop index variable I is now 6; loop will  iterate again.
MLOGIC(TRIM):  %if condition %qsubstr(&value,&i,1)^=_ _ is FALSE
MLOGIC(TRIM):  %do loop index variable I is now 5; loop will  iterate again.
MLOGIC(TRIM):  %if condition %qsubstr(&value,&i,1)^=_ _ is FALSE
MLOGIC(TRIM):  %do loop index variable I is now 4; loop will  iterate again.
MLOGIC(TRIM):  %if condition %qsubstr(&value,&i,1)^=_ _ is FALSE
MLOGIC(TRIM):  %do loop index variable I is now 3; loop will  iterate again.
MLOGIC(TRIM):  %if condition %qsubstr(&value,&i,1)^=_ _ is TRUE
MLOGIC(TRIM):  %goto trimmed (label resolves to TRIMMED).
MLOGIC(TRIM):  %if condition &i>0 is TRUE
MLOGIC(TRIM):  Ending execution.
**123**
```

Even though this sample demonstrates all six MLOGIC elements of debugging in action, this particular sample focuses on the ability of MLOGIC to provide the location of the AUTOCALL macro %TRIM, which is **`C:\Program Files\SAS\SAS 9.1\core\sasmacro\trim.sas`**. A subsequent section in this paper presents another option that you can use to determine the location of the AUTOCALL macro %TRIM.

SYMBOLGEN

There will be times when you reference a macro variable, but you are unsure of the resolved value. In such cases, you can rely on the superhero SYMBOLGEN.

SYMBOLGEN displays in the SAS log the value that results when you reference a macro variable. Then you can check the value that appears in the SAS log to ensure that the variable was resolved as expected. This might seem like a trivial contribution from this superhero, but using SYMBOLGEN can answer the following pertinent questions:

- Am I getting the correct value?
- Is the case of the value correct?
- Does the value contain special characters that need attention?
- Is the macro variable created with a macro-quoting function?
- What is the resolution of an indirect macro variable reference?

For example, suppose a %IF condition is not evaluated as expected. SYMBOLGEN can show the value that was used for the macro variable in that %IF statement. You could have a typographical error or an incorrect value. Because the macro language is case-sensitive, you can use %IF with SYMBOLGEN to determine if the case of a macro variable is what is expected. For example, in a macro you might have a parameter named DEPARTMENT. If the user who is invoking the macro enters the character string **`accounting`**, SYMBOLGEN will show the value of the macro variable that is being evaluated in the following %IF statement:

```
%if &department=ACCOUNTING %then %do;
```

This condition will be evaluated as **`False`** because the case does not match. SYMBOLGEN produces the following log information:

```
symbolgen:  macro variable department resolves to accounting
```

By looking at the %IF condition and the SYMBOLGEN output, you can see that the case does not match, therefore, you can determine that the %UPCASE function is required. %UPCASE converts the value of the DEPARTMENT variable to uppercase so that the %IF condition is evaluated as expected:

```
%if %upcase(&department)=ACCOUNTING %then %do;
```

Macro variables are often evaluated in a macro function. In this example, a comma-separated list is stored in a macro variable that is created by using a CALL SYMPUTX statement. This macro variable is used in the subsequent %SCAN function. Without knowing that the argument to the function contains commas, an error is generated by the %SCAN function. Here is a very simple macro that returns the third value of the VALUE variable:

```
   /**Code**/
options symbolgen;
data _null_;
   call symputx('value','A,B,C,D,E');
run;

%macro test;
   %let val_3=%scan(&value,3,%str(,));
   %put val_3=&val_3;
%mend;
%test
```

However, this code returns the following error:

```
SYMBOLGEN:  Macro variable VALUE resolves to A,B,C,D,E
ERROR: Macro function %SCAN has too many arguments.  The excess arguments will be
       ignored.
ERROR: A character operand was found in the %EVAL function or %IF condition where a
       numeric operand is required. The condition was: B
ERROR: Argument 2 to macro function %SCAN is not a number.
ERROR: The macro TEST will stop executing.
```

This error is generated because of the commas that are contained in the value of the VALUE variable. The %SCAN function interprets the commas to mean that there are too many arguments to the function. The SYMBOLGEN output shows the commas in VALUE, so you can see that you need to add a macro-quoting function. Use the %BQUOTE function to mask the commas and enable the %SCAN function to evaluate as expected. Here is the new log:

```
   /**Log**/
28   options symbolgen;
29
30   data _null_;
31      call symputx('value','A,B,C,D,E');
32   run;

NOTE: DATA statement used (Total process time):
      real time           0.00 seconds
      CPU time            0.00 seconds


33
34   %macro test;
35      %let val_3=%scan(%bquote(&value),3,%str(,));
36      %put val_3=&val_3;
37   %mend;
38   %test
SYMBOLGEN:  Macro variable VALUE resolves to A,B,C,D,E
SYMBOLGEN:  Macro variable VAL_3 resolves to C
val_3=C
```

QUOTING ISSUES

One of the most complicated aspects of the macro language is macro quoting. A *macro-quoting function* uses hexadecimal delta characters to mask special characters in the value of a macro variable. These delta characters can often cause syntax errors in code that appears to be completely acceptable in the log. In the following example,

our superhero SYMBOLGEN teams up with another superhero, MPRINT, to show the code that is generated by the macro. The abilities of MPRINT are discussed in greater detail later in this paper. Consider the following example:

```
   /**Code**/
options symbolgen mprint;
%macro test;
%let val = aaa;
%let testval = %str(%'&val%');

data _null_;
   file print;
   val = &testval;
   put 'VAL =' val;
run;
%mend;
%test
```

When you look at the log that results from the preceding example, you see that the MPRINT output shows correct syntax, but an error is generated:

```
   /**Log**/
41   %let val = aaa;
42   %let testval = %str(%'&val%');
43
44   data _null_;
45      file print;
46      val = &testval;
47      put 'VAL =' val;
48   run;
49   %mend;
50   %test
SYMBOLGEN:  Macro variable VAL resolves to aaa
MPRINT(TEST):   data _null_;
MPRINT(TEST):   file print;
SYMBOLGEN:  Macro variable TESTVAL resolves to 'aaa'
SYMBOLGEN:  Some characters in the above value which were subject to macro quoting
            have been unquoted for printing.
NOTE: Line generated by the macro variable "TESTVAL".
50    'aaa'
      -
      386
        ---
        202
MPRINT(TEST):   val = 'aaa';   ←—  This line of code appears correct
MPRINT(TEST):   put 'VAL =' val;
MPRINT(TEST):   run;

ERROR 386-185: Expecting an arithmetic expression.

ERROR 202-322: The option or parameter is not recognized and will be ignored.

NOTE: The SAS System stopped processing this step because of errors.
NOTE: DATA statement used (Total process time):
      real time           0.09 seconds
      CPU time            0.03 seconds
```

The following SYMBOLGEN output indicates that there might be a macro-quoting issue:

```
SYMBOLGEN:  Macro variable TESTVAL resolves to 'aaa'
SYMBOLGEN:  Some characters in the above value which were subject to macro quoting
            have been unquoted for printing.
```

This message indicates that the macro variable listed in the first SYMBOLGEN output was created with a macro-quoting function. If the syntax looks correct in the MPRINT output and the previous SYMBOLGEN output is generated, that is a good indication that the %UNQUOTE function is required. Adding the %UNQUOTE function will resolve the error. Here are the modified statements and log:

```
   /**Code**/
options symbolgen mprint;
%macro test;
%let val = aaa;
%let testval = %str(%'&val%');

data _null_;
   file print;
   val = %unquote(&testval);   ←—%UNQUOTE was added here
   put 'VAL =' val;
run;
%mend;
%test


   /**Log**/
62   %test
SYMBOLGEN:  Macro variable VAL resolves to aaa
MPRINT(TEST):   data _null_;
MPRINT(TEST):   file print;
SYMBOLGEN:  Macro variable TESTVAL resolves to 'aaa'
SYMBOLGEN:  Some characters in the above value which were subject to macro quoting
            have been unquoted for printing.
MPRINT(TEST):   val = 'aaa';
MPRINT(TEST):   put 'VAL =' val;
MPRINT(TEST):   run;

NOTE: 1 lines were written to file PRINT.
NOTE: DATA statement used (Total process time):
      real time           0.20 seconds
      CPU time            0.00 seconds
```

If a macro-quoting function is used when a macro variable is created, SAS always returns a SYMBOLGEN message, similar to the following: **`Some characters in the above value which were subject to macro quoting...`** However, because the %UNQUOTE function removes the delta characters that are used to mask the single quotation marks, an error message is not printed to the log.

Finally, SYMBOLGEN can be used to track the resolution of an indirect macro-variable reference. An *indirect reference* is a macro variable that contains multiple ampersands—for example, &&VAR&i. This type of syntax is often used in a macro %DO loop, where &I is the index for the %DO loop. However, this example uses a simpler approach just to illustrate what the SYMBOLGEN output will display. The SYMBOLGEN output shows you exactly how this indirect macro variable is referenced as follows:

```
   /**Code**/
options symbolgen;

%let var1=test;
%let n=1;

%put &&var&n;
```

In the log, SYMBOLGEN shows exactly how the macro reference is resolved.

```
   /**Log**/
2896  %let var1=test;
2897  %let n=1;
2898
2899  %put &&var&n;
SYMBOLGEN:  && resolves to &.
SYMBOLGEN:  Macro variable N resolves to 1
SYMBOLGEN:  Macro variable VAR1 resolves to test
test
```

The preceding code is processed in the following way: the first two ampersands in &&VAR&N resolve to a single ampersand, and &N resolves to `1`. That leaves &VAR1, which resolves to **`test`**. The previous SYMBOLGEN output nicely shows the resolution of the macro-variable reference.

MPRINT

Macros are most commonly used to generate code, but depending on the complexity of the macro, it can be hard to determine what the final code will look like. When this problem strikes, superhero MPRINT is on the scene to lead the way. You can use MPRINT to echo the generated code to the log. Here is an example:

```
   /**Code**/
%macro mktitle(proc,data);
   title "%upcase(&proc) of %upcase(&data)";
%mend mktitle;

%macro runplot(ds);
      /** Determine if GRAPH is licensed **/
   %if %sysprod(graph)=1 %then %do;
       %mktitle (gplot,&ds)
       proc gplot data=&ds;
          plot style*price
              / haxis=0 to 150000 by 50000;
       run;
       quit;
   %end;
   %else %do;
         %mktitle (plot,&ds)
         proc plot data=&ds;
            plot style*price;
         run;
         quit;
   %end;
%mend runplot;

options mprint;
%runplot(sasuser.houses)

/**Log**/
MPRINT(MKTITLE):   TITLE "GPLOT of SASUSER.HOUSES";
MPRINT(RUNPLOT):   PROC GPLOT DATA=SASUSER.HOUSES;
MPRINT(RUNPLOT):   PLOT STYLE*PRICE / HAXIS=0 TO 150000 BY 50000;
MPRINT(RUNPLOT):   RUN;
MPRINT(RUNPLOT):   QUIT;
```

As you can see from the log, MPRINT gives us the exact code that is generated from the macro. This will be the actual code that gets executed. With MPRINT you are also able to determine which macro produced which statement. Notice in the example that the TITLE statement came from the MKTITLE macro, and the other statements came from the RUNPLOT macro.

SUPER SIDEKICKS

Most superheroes have super sidekicks. Occasionally superheroes call on their sidekicks to help them battle the villains. In addition to the three debugging options that were already discussed, there are five more that can be used. These super sidekicks are MFILE, MLOGICNEST, MPRINTNEST, MAUTOLOCDISPLAY, and MCOMPILENOTE. Although these sidekicks are not called on as often as the superheroes, they can also help resolve macro issues.

MFILE

A vexing problem that is difficult to resolve is when the log contains an error that flags the macro invocation as the culprit. You know the error resides *somewhere* in the generated code, but the log does not provide the correct line number. In this case, two heroes, MPRINT and MFILE, team up to help you find the true location of the error. MFILE, used in conjunction with MPRINT, enables you to run the code that is generated by the macro. In this example, the line that causes the error to occur is identified:

```
   /**Code**/
%macro test;
data one;
   temp='abc';
   new=substr(temp,1,5);
 run;
%mend test;
%test
```

```
   /**Log**/
3328  options nomlogic;
3329  %macro test;
3330   data one;
3331      temp='abc';
3332      new=substr(temp,1,5);
3333   run;
3334  %mend test;
3335  %test

NOTE: Invalid third argument to function SUBSTR at line 3335 column 34.
      temp=abc new=abc _ERROR_=1 _N_=1
NOTE: The data set WORK.ONE has 1 observations and 2 variables.
NOTE: DATA statement used (Total process time):
      real time           0.00 seconds
      CPU time            0.00 seconds
```

The line that is flagged in the log for setting _ERROR_ to **1** is 3335, which is the macro invocation. When you examine the log, there is no way to tell what is causing the problem except that it came from using a SUBSTR function. This is a small sample, but most of the time, this error will come from a large macro that is too big to determine where the error occurs. This is where MFILE and MPRINT team up to help. Let's look at the same sample but this time turn on the MFILE and MPRINT options:

```
   /**Code**/
filename mprint 'c:\nomac.sas';
options mprint mfile;
%macro test;
   data one;
      temp='abc';
      new=substr(temp,1,5);
run;
%mend test;
%test
```

When you use the MFILE option you must also have a FILENAME statement with the fileref MPRINT. The FILENAME statement points to the file that you are creating, which will contain the code that is generated from the macro:

```
   /**Code from the nomac.sas file**/
data one;
temp='abc';
new=substr(temp,1,5);
run;
```

After running the preceding code, you get the following log:

```
   /**Log**/
3345  data one;
3346  temp='abc';
3347  new=substr(temp,1,5);
3348  run;

NOTE: Invalid third argument to function SUBSTR at line 3347 column 5.
      temp=abc new=abc _ERROR_=1 _N_=1
NOTE: The data set WORK.ONE has 1 observations and 2 variables.
NOTE: DATA statement used (Total process time):
      real time           0.00 seconds
      CPU time            0.00 seconds
```

Looking at the new log, you can now see that line number 3347 is identified, and this is the correct line that is causing _ERROR_ to be set to **1**. Using this option can save a lot of time in debugging because now the correct locations of the errors are revealed.

MLOGICNEST

With nested macros, it's often hard to determine which macro is invoked from within which macro. Super sidekick MLOGICNEST enables the macro-nesting information to be written to the SAS log in the MLOGIC output.

MLOGICNEST shows the nesting sequence of the macros in the log, listing the outer-most macro to the inner-most macro. In order to use MLOGICNEST, the MLOGIC system option must be turned on. The best way to explain MLOGICNEST is to show an example. Here is the SAS code:

```
   /**Code**/
%macro outer;
   %put THIS IS OUTER;
   %inner;
%mend outer;
%macro inner;
   %put THIS IS INNER;
   %inrmost;
%mend inner;
%macro inrmost;
   %put THIS IS INRMOST;
%mend;

options mlogic mlogicnest;
%outer
```

Here is the MLOGIC output in the SAS log using the MLOGICNEST option:

```
   /**Log**/
292  %outer
MLOGIC(OUTER):  Beginning execution.
MLOGIC(OUTER):  %PUT THIS IS OUTER
                THIS IS OUTER
MLOGIC(OUTER.INNER):  Beginning execution.
MLOGIC(OUTER.INNER):  %PUT THIS IS INNER
                      THIS IS INNER
MLOGIC(OUTER.INNER.INRMOST):  Beginning execution.
MLOGIC(OUTER.INNER.INRMOST):  %PUT THIS IS INRMOST
                              THIS IS INRMOST
MLOGIC(OUTER.INNER.INRMOST):  Ending execution.
MPRINT(INNER):  ;
MLOGIC(OUTER.INNER):  Ending execution.
MPRINT(OUTER):  ;
MLOGIC(OUTER):  Ending execution.
```

MPRINTNEST

Super sidekick MPRINTNEST enables the macro-nesting information to be written to the SAS log in the MPRINT output. In order to use MPRINTNEST, the MPRINT system option must be turned on.

The following example uses the MPRINT and MPRINTNEST options:

```
   /**Code**/
%macro outer;

data _null_;
   %inner
run;

%mend outer;
%macro inner;
   put %inrmost;
%mend inner;
%macro inrmost;
   'This is the text of the PUT statement'
%mend inrmost;

options mprint mprintnest;
%outer
```

Here is the output that is written to the SAS log using both the MPRINT and the MPRINTNEST options:

```
   /**Log**/
MLOGIC(OUTER):  Beginning execution.
MPRINT(OUTER):   data _null_;
MLOGIC(OUTER.INNER):  Beginning execution.
MPRINT(OUTER.INNER):   put
MLOGIC(OUTER.INNER.INRMOST):  Beginning execution.
MPRINT(OUTER.INNER.INRMOST):  'This is the text of the PUT statement'
MLOGIC(OUTER.INNER.INRMOST):  Ending execution.
MPRINT(OUTER.INNER):  ;
MLOGIC(OUTER.INNER):  Ending execution.
MPRINT(OUTER):   run;

This is the text of the PUT statement
NOTE: DATA statement used (Total process time):
      real time           0.00 seconds
      cpu time            0.00 seconds


MLOGIC(OUTER):  Ending execution.
```

MAUTOLOCDISPLAY

MAUTOLOCDISPLAY enables a macro to display the source location of the AUTOCALL macro in the log when an AUTOCALL macro is invoked. This enables you to determine the source location of the macro without having to turn on the MLOGIC system option. The location of the AUTOCALL macro is important when code changes are required. Here is the output of the option.

```
308  options mautolocdisplay nomlogic;
309  %let x=%trim(ABC);
mautolocdisplay(trim):  This macro was compiled from the autocall file
C:\Program Files\SAS\SAS 9.1\core\sasmacro\trim.sas
```

MCOMPILENOTE

MCOMPILENOTE confirms that the compilation of a macro was completed. This super sidekick also issues a note that contains the size and number of instructions of the macro. There are times when a macro is defined, but it never executes. It's possible that the macro compiled, but with errors. The syntax for MCOMPILENOTE is different from the other options because this option must contain an equal sign with one of three possible values. The syntax is as follows:

OPTIONS MCOMPILENOTE=<NONE | NOAUTOCALL | ALL>;

The value NONE prevents any NOTE from being written to the log.

The value NOAUTOCALL prevents any NOTE from being written to the log for AUTOCALL macros, but a NOTE is issued to the log upon the completion of the compilation of any other macro.

The value ALL issues a NOTE to the log upon the completion of the compilation of any macro.

Here is an example in which a macro compiles successfully without errors:

```
   /**Code**/
option mcompilenote=noautocall;
%macro test;
   %put this is a test;
%mend test;

   /**Log**/
3601  option mcompilenote=noautocall;
3602  %macro test;
3603     %put this is a test;
3604  %mend test;
NOTE: The macro TEST completed compilation without errors.
      3 instructions 36 bytes.
```

In this case, the macro compiled successfully without complications. There were three instructions: the %MACRO statement; the %PUT statement; and the %MEND statement. There are 36 bytes in the total length of all instructions in the macro. Now let's look at an example in which compilation is successful, but with errors:

```
/**Code**/
option mcompilenote=noautocall;
%macro test;
   %end;
%mend test;

   /**Log**/
3617  option mcompilenote=noautocall;
3618  %macro test;
3619  %end;
ERROR: There is no matching %DO statement for the %END. This statement will be
       ignored.
3620  %mend test;
NOTE: The macro TEST completed compilation with errors.
      2 instructions 4 bytes.
```

Notice that this time the NOTE reflects that the macro is compiled, but with errors. There is no matching %DO for this %END statement, therefore, an error occurs, and no instruction is generated for the %END. The two instructions that the note refers to are generated for the %MACRO statement and the %MEND statement. The result is a macro that does nothing, which takes only 4 bytes of code. MCOMPILENOTE is beneficial for large macros because a note is generated that indicates whether an error occurred in the macro.

CONCLUSION

The macro language is a very powerful and, sometimes complicated, tool that can accomplish many tasks. As this paper emphasizes, using the macro debugging options when you develop macro code provides many benefits. During the development stage of a macro, the SAS system options SYMBOLGEN, MLOGIC, and MPRINT, along with others, will be your superheroes as you fight your way through confusing logs. Without these options, the SAS log gives you little information about the code that you just spent precious time developing.

CONTACT INFORMATION

Your comments and questions are valued and encouraged. Contact the authors at:

Kevin Russell
SAS Institute Inc.
SAS Campus Drive
Cary, NC 27513
E-mail: support@sas.com
Web: support.sas.com

Russ Tyndall
SAS Institute Inc.
SAS Campus Drive
Cary, NC 27513
E-mail: support@sas.com
Web: support.sas.com