PharmaSUG 2016 – Paper BB11

Surviving the SAS® Macro Jungle by Using Your Own Programming Toolkit

Kevin Russell, SAS Institute Inc., Cary, North Carolina

ABSTRACT

Almost every night there is a reality show on TV that shows someone trying to survive in a harsh environment by using just their wits. Although your environment is not as harsh, maybe you feel you do not have all the tools you need to survive in the macro programming jungle. This paper's purpose is to provide you with more programming tools, to make life in the jungle a whole lot easier. For example, the new DOSUBL function is the flint that will ignite your code and enable you to dynamically invoke your macro using DATA step logic. Another tool is the stored compiled macro facility, which can provide protection from the elements by enabling you to keep your code safe from prying eyes. Also included is a discussion of the macro debugging system options, MLOGIC, SYMBOLGEN, and MPRINT. These options act as a compass to point you in the right direction. They can help guide you past a log riddled with errors to an error-free log that puts you back on the right path. CALL SYMPUTX is the new and improved version of CALL SYMPUT. CALL SYMPUTX automatically converts numeric values to character strings and even automatically removes trailing blanks, saving you steps and getting you out of the jungle faster. These tools, along with macro comments, enhancements to %PUT, the MFILE system option, read-only macro variables, the IN operator, and SASHELP.VMACRO, help equip all macro programmers with the tools that they need to survive the macro jungle.

INTRODUCTION

The macro language is an extremely powerful tool within SAS® software that enables you to add functionality to your code as well as make it much more dynamic. However, this power comes with a certain level of complexity. It is this complexity that can sometimes frustrate programmers and make them feel like the macro language is a jungle—difficult to navigate and full of unseen dangers.

This paper describes ten macro programming tools that programmers of all levels can add to their toolboxes. Although a few of these tools have been available for a while, some macro programmers might not yet be aware of them or familiar with their uses. The tools discussed in this paper are intended to help programmers better understand the macro language and program more effectively. For example, the DOSUBL function provides programmers with new functionality to invoke macros from within a DATA step based on DATA step logic, which makes it easier to accomplish complex coding tasks. Another useful tool, the stored compiled macro facility, enables programmers to protect their code from end users. This paper also covers system options that write essential information to the SAS® log to make debugging macro applications much simpler.

This paper discusses the following macro programming tools:

- The DOSUBL function
- The stored compiled macro facility
- Macro debugging system options
- The CALL SYMPUTX routine
- Using comments within a macro definition
- The %PUT statement
- The MFILE system option
- Read-only macro variables
- The macro IN operator
- SASHELP.VMACRO

THE DOSUBL FUNCTION

Quick Facts

- DOSUBL is a new and improved version of the CALL EXECUTE routine.
- DOSUBL was introduced in SAS® 9.4.

SAS Technical Support often receives questions such as the following:

- "How can I send the data for every hospital listed in my data set to a separate CSV file?"
- "How can I run a PROC REPORT for each region listed in my data set?"
- "How can I send an email to each manager listed in my data set?"

Each of these questions has something in common. Each question is asking how to execute a task based on the value of a data-set variable. In SAS, the best way to execute a repetitive task is to use the SAS® Macro language. The best way to invoke a macro based on a data-set variable is to use the new DOSUBL function. You can use DOSUBL the same way you use the CALL EXECUTE routine, but with some added functionality.

Like CALL EXECUTE, DOSUBL gives you the ability to perform two tasks:

- Conditionally execute a macro based on DATA step logic.
- Pass in a data-set variable's value as the value of a macro parameter.

The following example shows how to use these abilities with DOSUBL. This example executes a REPORT procedure for each hospital listed in the HOSPITALS data set. Below is the sample data set that has data for two hospitals, including the hospital's ID number, the quarter, and the number of patients in the hospital during that quarter.

```
data hospitals;
   input hospital_id $ quarter number_of_patients;
   datalines;
A100 1 125
A100 2 115
A100 3 130
A100 4 110
A200 1 200
A200 2 195
A200 3 180
A200 4 190
;
```

In the next step, the MYMAC macro uses the HOSP_ID parameter in a WHERE clause to subset the data in the HOSPITALS data set. Each time the macro is executed, the value of HOSP_ID changes so that a separate PROC REPORT is run for each hospital. The reports show the data for each hospital individually.

```
/* this macro runs a PROC REPORT for each hospital */
%macro mymac(hosp_id);
title "Data for hospital:  &hosp_id";
proc report data=hospitals(where=(hospital_id="&hosp_id"));
  column hospital_id quarter number_of_patients;
  define hospital_id / order;
run;
%mend;
```

The last step is the DATA null step that contains the DOSUBL function. This DATA step uses the BY statement to enable BY-group processing. This type of processing causes the DOSUBL function to be executed only when the BY group changes, which then invokes the MYMAC macro. In order to invoke the macro, the argument to the DOSUBL function must be a text string containing the macro invocation. In other words, the first time the DOSUBL function executes, the argument to the function is **`%mymac(A100)`**. Because there are two unique values for the BY variable in this example, this code invokes MYMAC twice, producing reports for both hospitals listed.

```
data _null_;
   set hospitals;
   by hospital_id;
/* this logic allows us to call the MYMACRO macro
each time the value of HOSPITAL_ID changes */
   if first.hospital_id then do;
    rc=dosubl(cats('%mymac(',hospital_id,');'));
   end;
run;
```

In addition to the abilities illustrated in the above example, the DOSUBL function also provides you with the ability to move macro variables to and from the calling environment.

The following example demonstrates this ability. This example shows the DOSUBL function being used to invoke the TRYIT macro, which then uses CALL SYMPUTX to create the macro variable MACVAR. Then, this macro variable is resolved in the DATA step that contains the DOSUBL function.

Note: This code contains numbers that are bolded and enclosed in parentheses, which correspond to a numbered list below that explains elements of the code.

```
%macro tryit(x);
data _null_;
   if &x=1 then macvar=2;
   else macvar=3;
   call symputx('macvar', macvar, 'g');     (2)
run;
%mend;

data _null_;
   rc=dosubl('%tryit(1)');                 (1)
   x=symget('macvar');                     (3)
   put x= '(should be 2)';
run;
```

1. The DOSUBL function executes the TRYIT macro, passing in a value of **1** for the X parameter. All of the code within the TRYIT macro executes immediately.
2. Then, the CALL SYMPUTX statement creates the global macro variable MACVAR. The value of this macro variable is returned to the calling DATA step.
3. This assignment statement uses the SYMGET function to resolve the macro variable MACVAR. SYMGET is needed because the resolution of the macro variable MACVAR must be delayed until execution time.

This example illustrates the key difference between DOSUBL and CALL EXECUTE. When CALL EXECUTE is used to invoke a macro, the macro executes immediately. Execution of SAS statements generated by the execution of the macro are delayed until after a step boundary is reached. SAS macro statements, including macro variable references, execute immediately.

When DOSUBL is used to invoke a macro, all code is executed immediately, including all SAS statements. This is important in this example because of the timing of the CALL SYMPUTX call routine, which is a DATA step call routine. So, in the example above, if the TRYIT macro was invoked using CALL EXECUTE, CALL SYMPUTX would not execute until a step boundary was reached. This means that the macro variable MACVAR would not be available in the calling DATA null step.

As you can see, the DOSUBL function is a powerful tool. DOSUBL can be the flint that ignites your code and enables you to make your macro invocations data driven.

THE STORED COMPILED MACRO FACILITY

Quick Facts

- Always save your macro source code, because once the macro is compiled, you cannot re-create the source statements.
- The stored compiled macro facility compiles the code only once.

The stored compiled macro facility enables you to store large production jobs that are updated infrequently in a library that you specify within the SASMACR catalog. The macro code is compiled only the first time you submit the code. In all subsequent calls to the macro, the macro processor executes the compiled code. Skipping the compilation phase enables the code to run faster.

However, there is another reason that the stored compiled macro facility is useful. SAS Technical support often receives requests from customers about how they can share code they have written with another person without having that person be able to see the actual code. The answer is to use the stored compiled macro facility.

The following sections discuss three primary reasons why the stored compiled facility is successful at protecting your source code.

STORES MACRO CODE IN AN UNREADABLE MACHINE LANGUAGE

When a macro is compiled, the code is stored in an unreadable machine language as an entry in the SASMACR catalog. There is no way to see the actual macro code that created the entry.

PREVENTS USERS FROM USING MACRO DEBUGGING SYSTEM OPTIONS ON YOUR MACRO

If you specify the SECURE option in the %MACRO statement when a macro is compiled, the macro debugging system options have no effect when the user executes the macro. This means that if users have the MPRINT, SYMBOLGEN, and MLOGIC system options enabled in their SAS session, these options do not write any output to the log when the stored compiled macro is executed. The two logs below show the differences in the log output when you run a macro that was compiled without the SECURE option specified and when the macro was compiled with the SECURE option specified.

Output 1 shows the resulting log from executing the TEST macro without specifying the SECURE option.

```
503  %test
MLOGIC(TEST):  Beginning execution.
SYMBOLGEN:  Macro variable SYSDATE resolves to 07MAR16
MLOGIC(TEST):  %IF condition &sysdate ne 31MAR16 is TRUE
MPRINT(TEST):   data new;
MPRINT(TEST):   set sashelp.class;
MPRINT(TEST):   run;

NOTE: There were 19 observations read from the data set SASHELP.CLASS.
NOTE: The data set WORK.NEW has 19 observations and 5 variables.
NOTE: DATA statement used (Total process time):
      real time           0.01 seconds
      cpu time            0.01 seconds


MPRINT(TEST):   proc freq data=new;
MPRINT(TEST):   table age;
MPRINT(TEST):   run;

NOTE: There were 19 observations read from the data set WORK.NEW.
NOTE: PROCEDURE FREQ used (Total process time):
      real time           0.01 seconds
      cpu time            0.01 seconds


MLOGIC(TEST):  Ending execution.
```

Output 1. Log Output from Executing a Macro Not Compiled Using the SECURE Option

Although you cannot see the definition for the TEST macro in Output 1, you can see that a DATA step and a FREQ procedure step were conditionally executed based on the value of the automatic macro variable SYSDATE. The complete DATA step code and PROC FREQ code are visible in the log.

When you compile this same macro using the SECURE option, the following log is produced when the macro is executed:

```
NOTE: There were 19 observations read from the data set SASHELP.CLASS.
NOTE: The data set WORK.NEW has 19 observations and 5 variables.
NOTE: DATA statement used (Total process time):
      real time           0.00 seconds
      cpu time            0.00 seconds



NOTE: There were 19 observations read from the data set WORK.NEW.
NOTE: PROCEDURE FREQ used (Total process time):
      real time           0.01 seconds
      cpu time            0.01 seconds
```

Output 2. Log Output from Executing a Macro Compiled Using the SECURE Option

As Output 2 shows, even if users specify the MPRINT, SYMBOLGEN, and MLOGIC system options, only NOTES, WARNINGS, and ERRORS are written to the log when the macro executes if the SECURE option is specified. The users cannot see any of the actual code in the log.

PREVENTS USERS FROM RETRIEVING YOUR MACRO SOURCE CODE

Finally, storing your macro as a stored compiled macro prevents users from being able to access the code. Once a macro has been compiled, there is no way to decompile it. So, if users have access only to the macro catalog, they are not able to retrieve the code that created the entries in the catalog.

For a complete discussion of the stored compiled macro facility, see the "Saving Macros Using the Stored Compiled Macro Facility" section of *SAS® 9.4 Macro Language: Reference, Fourth Edition* (**support.sas.com/documentation/cdl/en/mcrolref/67912/HTML/default/viewer.htm#n0sjezyl65z1cpn1b6mqfo8115h2.htm**).

Using the stored compiled macro facility is one of the best ways to protect your code from the elements and from prying eyes.

MACRO DEBUGGING SYSTEM OPTIONS

Quick Facts

- The SYMBOLGEN system option can also be specified as SGEN.
- SYMBOLGEN, MPRINT, and MLOGIC must be enabled.

A best practice for macro programmers is to ***always*** have the SYMBOLGEN, MPRINT, and MLOGIC system options enabled when developing a macro application. These system options write out information about the execution of your macro to help you with debugging any undesired results. If these options are not enabled, it can be nearly impossible to debug macro code.

When you are debugging your code, you need to know whether the values of your macro variables are correct. The SYMBOLGEN system option writes out the resolved value of a macro variable each time it is referenced, and it is one of the best ways to find this information.

The MPRINT system option writes out all the non-macro SAS statements that are generated by the macro's execution. This means that MPRINT writes out code like an OPTIONS statement, LIBNAME statement, DATA step code, and PROC code that is contained within the macro definition. If the non-macro code in your macro definition is causing an error, the MPRINT output in the log is the best source of information to help you debug the error.

MLOGIC is the most informative of the three system options. MLOGIC writes the following out to the log:

- The beginning and ending of macro execution
- The location of an autocall macro's definition
- The values of the macro parameters when the macro is invoked
- The execution of macro statements
- The true or false status for each %IF condition

You can use the output of the MLOGIC system option to help track where you are in the code when tracing its execution in the log. For example, if the outermost macro contains a %IF condition that determines which macro is going to be invoked next, the output of MLOGIC is the best tool to show how the condition was evaluated and which macro is going to be invoked next.

Output 3 is an example of a log that shows an error that was generated during macro execution when these system options were not enabled. The ERROR shows that the variable **GENDER** was referenced when reading the SASHELP.CLASS data set. The log does not show where in the code this error occurred or what syntax caused it.

```
18   %b
ERROR: Variable gender is not on file SASHELP.CLASS.

NOTE: The SAS System stopped processing this step because of errors.
WARNING: The data set WORK.NEW may be incomplete.  When this step was stopped there
were 0 observations and 0 variables.
WARNING: Data set WORK.NEW was not replaced because this step was stopped.
NOTE: DATA statement used (Total process time):
      real time           0.01 seconds
      cpu time            0.01 seconds


NOTE: No variables in data set WORK.NEW.
NOTE: PROCEDURE PRINT used (Total process time):
      real time           0.01 seconds
      cpu time            0.01 seconds
```

Output 3. Output Showing ERROR from Macro Execution

Output 4 shows the log after running the exact same code, but with the SYMBOLGEN, MPRINT, and MLOGIC system options enabled before executing the code.

As you can see, much more information is written to the log:

```
5    %b
MLOGIC:  Beginning compilation of B using the autocall file c:\dstep\b.sas.
MLOGIC:  Ending compilation of B.
MLOGIC(B):  Beginning execution.
MLOGIC(B):  This macro was compiled from the autocall file c:\dstep\b.sas   (6)
MLOGIC(B):  %LET (variable name is V)                (5)
MLOGIC(B):  Beginning compilation of A using the autocall file c:\dstep\a.sas.
MLOGIC(B):  Ending compilation of A.
MLOGIC(A):  Beginning execution.
MLOGIC(A):  This macro was compiled from the autocall file c:\dstep\a.sas
SYMBOLGEN:  Macro variable V resolves to gender       (4)
MLOGIC(A):  Parameter VAR has value gender            (3)
MPRINT(A):   data new;
SYMBOLGEN:  Macro variable VAR resolves to gender     (2)
MPRINT(A):   set sashelp.class(where=(gender='M'));
ERROR: Variable gender is not on file SASHELP.CLASS.  (1)
MPRINT(A):   run;

NOTE: The SAS System stopped processing this step because of errors.
WARNING: The data set WORK.NEW may be incomplete.  When this step was stopped there
were 0 observations and 0 variables.
NOTE: DATA statement used (Total process time):
      real time           0.01 seconds
      cpu time            0.01 seconds

MPRINT(A):   proc print;
MPRINT(A):  run;
NOTE: No variables in data set WORK.NEW.
NOTE: PROCEDURE PRINT used (Total process time):
      real time           0.03 seconds
      cpu time            0.03 seconds

MLOGIC(A):  Ending execution.
MLOGIC(B):  Ending execution.
```

Output 4. Log Output from Macro Execution with Debugging System Options Enabled

The numbers in this list correspond to numbers bolded and enclosed in parentheses in Output 4 and contain an explanation of how you can use this output to debug a problem with macro code.

1. In the MPRINT output in the line above the ERROR, you see that **GENDER** is being referenced in a WHERE clause in a SET statement. You know that **GENDER** is not the correct variable name. So, you need to determine where the **GENDER** value is being introduced.
2. The SYMBOLGEN output in this line shows that the macro variable VAR resolves to **GENDER**. This lets you know that the value of VAR is what you need to investigate.
3. The MLOGIC output in this line shows that the parameter VAR has the value of **GENDER**.
4. The SYMBOLGEN output in this line provides a great clue about where the value of **GENDER** originates from. The SYMBOLGEN output shows that the macro variable V resolves to **GENDER**. Now you must investigate the value of the macro variable V.
5. The MLOGIC output in this line shows that the macro variable V is created with a %LET statement. So, the next step is to go to the code and change the %LET so that it lists the correct value. Note that 'MLOGIC' is followed by a parenthetical reference to the macro name that contains the code MLOGIC is commenting on. Now you know that you must go to the definition for macro B to fix the code.
6. The MLOGIC output in this line shows that macro B was compiled from the following location: **c:\dstep\b.sas**, which is where you must go to correct the error.

If you ever find yourself lost in the jungle, you could use the sun to determine which way is north, but using a compass is much easier. The next time you are lost in the macro jungle, let the macro debugging system options SYMBOLGEN, MPRINT, and MLOGIC be your compass to guide you out of the jungle.

THE CALL SYMPUTX ROUTINE

Quick Facts

- CALL SYMPUTX includes all the functionality of the CALL SYMPUT routine.
- CALL SYMPUTX has two required arguments (*macro-variable* and *value*) and one optional argument (*symbol-table*).

The CALL SYMPUTX routine is a new and improved version of the CALL SYMPUT routine. For a complete list of improvements, see "CALL SYMPUTX Routine" in *SAS® Functions and CALL Routines: Reference, Fourth Edition* (**support.sas.com/documentation/cdl/en/lefunctionsref/67960/HTML/default/viewer.htm#n1nexcs36ctqk5n11uao7k9myz7y.htm**). Here are two of the most beneficial improvements that CALL SYMPUTX offers:

- Left-justifies arguments and trims all trailing blanks.
- Enables you to specify the symbol table in which to store the macro variable.

LEFT-JUSTIFIES ARGUMENTS AND TRIMS ALL TRAILING BLANKS

The following example using CALL SYMPUT shows why this improvement is important. If the second argument to CALL SYMPUT contains a numeric value, that value will be formatted using the BEST12 format and will be right-justified when it is stored as the value of a macro variable. This behavior can cause unexpected problems when you resolve that macro variable in your SAS syntax if the variable is going to be used as a numeric suffix. Consider the following example.

This code uses CALL SYMPUT to create a macro variable whose value is going to be used as a numeric suffix:

```
/* CALL SYMPUT is used to create the macro variable N */
  Call symput('n',total);
```

Then, it uses the macro variable N as a suffix in the DATA statement as shown below:

```
data new&n;
   set mylib.salesdata(where=(quarter='1'));
run;
```

This code produces the following error:

```
      cpu time            0.00 seconds

SYMBOLGEN:  Macro variable N resolves to           8
20621
20622
20623  data new&n;
NOTE: Line generated by the macro variable "N".
20623   new           8
                      -
                      22
                      200
ERROR 22-322: Syntax error, expecting one of the following: a name, a quoted
string, (, /, ;, _DATA_, _LAST_, _NULL_.

ERROR 200-322: The symbol is not recognized and will be ignored.

20624    set mylib.salesdata(where=(quarter='1'));
20625  run;

NOTE: The SAS System stopped processing this step because of errors.
WARNING: The data set WORK.NEW may be incomplete.  When this step was stopped there
were 0 observations and 0 variables.
WARNING: Data set WORK.NEW was not replaced because this step was stopped.
NOTE: DATA statement used (Total process time):
      real time           0.00 seconds
      cpu time            0.00 seconds
```

Output 5. Log Output from Macro Variable N Referenced in a DATA Statement

This error occurs because the actual value of N is eleven spaces and then the number **8**. These leading spaces in the macro variable value are causing the error. The spaces cause an unwanted gap between the data-set name **NEW** and the numeric suffix **8**.

Before CALL SYMPUTX, you had to use the LEFT and TRIM functions to left-justify the value and remove the unwanted spaces:

```
Call symput('n',trim(left(total)));
```

When you use CALL SYMPUTX, the value of the numeric variable TOTAL is automatically left-justified and any trailing blanks that remain are removed:

```
Call symputx('n',total);
```

Now you can safely use the macro variable N as a suffix without causing any unexpected errors.

SPECIFIES THE SCOPE OF THE MACRO VARIABLE

The second big advantage CALL SYMPUTX has over CALL SYMPUT is the ability to specify the scope of the macro variable being created, which can be useful if you need to create a large, but unknown number of macro variables. *Scope* refers to the symbol table that the macro variable is stored in. CALL SYMPUTX has a third argument that enables you to specify the scope of the macro being created:

```
Call symputx(macro-variable, value <, symbol-table>);
```

The two most common values for this argument are **G**, which specifies the global symbol table, and **L**, which specifies the most local symbol table that exists.

The following syntax enables you to create the global macro variables VAR1-VARn without having to specify a %GLOBAL statement for each variable. This example shows how to create a macro variable from each observation in the data set. This syntax is using the CATS function to concatenate the value of the automatic DATA step variable **_N_** to the prefix '**var**:'

```
Call symputx(cats('var',_n_),total,'g');
```

Creating a macro variable based on the value of a data-set variable is one of the most common tasks programmers perform using the macro language. Using CALL SYMPUTX instead of CALL SYMPUT makes this task easier, which gets you out of the jungle faster.

USING COMMENTS WITHIN A MACRO DEFINITION

Quick Facts

- Asterisk-style comments and macro-style comments might cause unexpected results in macro definitions.
- PL/1-style comments are read character-by-character, which allows them to contain special characters like macro triggers, semicolons, and unmatched quotation marks.

Using comments in your code is usually a good idea. Comments can make code easier to follow and understand for anyone who uses it. Comments can be especially helpful in code that is used and updated by multiple programmers. Comments at the beginning of the code can include a description of the entire program as well as information about who last updated the code and what changes were made.

When a macro definition is submitted, everything between the %MACRO statement and the %MEND statement is compiled. This can result in unexpected behavior when your macro definition contains comments. The following sections provide a description of what happens when the three different types of comments available in SAS are used within a macro definition.

ASTERISK-STYLE COMMENTS

Here is the syntax for asterisk-style comments:

```
* This is my comment;
```

Asterisk-style comments are simply stored as text when a macro is compiled. However, the contents of the comment are processed in the tokenizer, which means that special characters within the comment are seen by the word scanner. This can cause undesirable results if your comment contains a macro statement trigger, unmatched quotation marks, or a semicolon. For example, consider the following comment:

```
*The %let x=value in the following section is used to set the lower limit;
```

The **%** is recognized by the word scanner as a macro trigger that causes the %LET statement to execute. The result is the creation of the macro variable X with a value of **'value in the following section is used to set the lower limit**'. The word scanner uses the semicolon that was intended to end the comment to actually end the %LET statement. Then, the semicolon at the end of the next statement in the code is used to end the comment.

MACRO-STYLE COMMENTS

Here is the syntax for macro-style comments:

```
%* This is my comment;
```

A macro-style comment is a macro statement, so it is processed by the macro processor when the **%** that begins the statement is encountered. Unlike an asterisk-style comment, a macro-style comment can contain a % without causing unexpected results. A % within a macro-style comment is not seen as a macro trigger, so no attempt is made to execute macro code within the comment. However, the contents of the macro-style comment are still tokenized. This means that unmatched quotation marks and semicolons can cause unexpected problems. For example, if you have a comment like the one below in your macro definition, you will encounter undesirable results:

```
%* The rest of the developer's comments in this section discuss the metadata;
```

Because the contents of this comment are tokenized, the tokenizer sees the apostrophe as the beginning of a *literal*, which is a string of characters enclosed in quotation marks. To end the literal token, the tokenizer will continue to gather tokens until the closing quotation mark to end the literal is found. Because the close to the literal is never found, the word scanner interprets that all of the subsequent code is part of the literal. When the macro that contains the unmatched quotation mark is submitted, no code executes.

PL/1-STYLE COMMENTS

Here is the syntax for PL/1-style comments:

```
/* This is my comment */
```

As you can see, there are certain situations when using asterisk-style comments and macro-style comments within a macro definition can be very problematic. Because of these problems, it is a best practice to use only PL/1-style comments when defining a macro. PL/1-style comments can include the following special characters:

```
/* The following special characters can all be safely used within this type
comment:  ' " % ;  */
```

PL/1-style comments are safer to use because of how they are processed. Instead of looking at each token within the comment, the word scanner looks at each character, character-by-character. The result is that special characters like **%** and **;** have no special meaning between the opening **/*** and the closing ***/**, making PL/1-style comments the safest method of including comments within your macro definition.

Using comments in your code can help the user better navigate the jungle. However, using the wrong style of comment in macro code can make it harder to navigate due to unexpected problems. So, every time you use comments in your macro code, you should use PL/1-style comments to make sure there are no issues with the contents of the comments.

THE %PUT STATEMENT

Quick Facts

- The %PUT statement has similar functionality to the PUT statement in the DATA step.
- The %PUT statement supports named output starting in SAS® 9.3.

One of the best ways to check your coding work is to use the %PUT statement. %PUT simply writes text or macro-variable information to the SAS log. The majority of the time, %PUT is used to write out the value of a macro variable. Here is an example in which the variable PROCEDURE contains an unknown number of comma-separated values. The objective is to capture the largest value of N into a macro variable. After the DATA step is complete, you must make sure that the value of the macro variable N is correct. You can use %PUT to do this.

```
571  data _null_;
572    set datain;
573    retain tmp;
574    n=countw(procedure,',');
575    if tmp>n then n=tmp;
576    else tmp=n;
577    call symputx('n',n);
578  run;

NOTE: There were 4 observations read from the data set WORK.DATAIN.
NOTE: DATA statement used (Total process time):
      real time           0.00 seconds
      cpu time            0.00 seconds


579  %put &=n;
N=6
```

Output 6. Log Output from %PUT

Because the data set used in this example contains only 4 observations, it was easy to make sure that **6** is indeed the correct value. Note the syntax for this %PUT statement, which is new for SAS 9.3:

```
%PUT &=macro-variable-name;
```

This syntax writes out the macro variable name, followed by an '**=**', followed by the macro variable's value. The variable name being followed by the '**=**' makes the macro variable's value easier to see when you look at the log.

The %PUT statement has multiple arguments that can be used to write out a specific group of macro variables. For a complete list of arguments, see the "%PUT Statement" section in *SAS® Macro Language: Reference, Fourth Edition* (**support.sas.com/documentation/cdl/en/mcrolref/67912/HTML/default/viewer.htm#n189qvy83pmkt6n1bq2mmwtyb4oe.htm**). _USER_ is the only argument detailed in this paper.

USER shows all of the user-generated local and global macro variables. Here is the syntax:

```
%put _user_;
```

When you specify _USER_, the output shows every macro variable that you have created, including the macro variable's name, scope, and value. This list can be extremely useful when you have created multiple macro variables and need to check your work.

Other than the macro debugging system options, the %PUT statement is one of the most useful tools that you can use to see whether your code is executing as expected and to inventory your macro variables.

THE MFILE SYSTEM OPTION

Quick Facts

- MFILE requires you to set the MPRINT option.
- To use MFILE, you must designate a fileref of MPRINT an external file. The non-macro code generated by MFILE is written to this external file.

An earlier section of the paper covered the "big three" macro debugging system options: SYMBOLGEN, MPRINT, and MLOGIC. This section discusses the macro system option MFILE that can be useful in debugging your code. MFILE enables you to route the non-macro code generated by your macro, which includes all the code that appears in the MPRINT output in the log, to an external file.

Here is a situation in which MFILE can be really useful. How many times have you seen notes similar to the following written to the SAS log?

```
NOTE: Invalid argument to function INPUT at line 839 column 10.
```

This note is returned because most SAS functions expect each of the function's arguments to be a certain variable type. The INPUT function expects the first argument to be a character value that can be converted into a numeric value. If the first argument lists a character value that cannot be converted into a numeric value, the NOTE shown above is written to the log.

Notice that the NOTE includes a specific line number. When this NOTE is generated by code that is not inside a macro definition, this line number corresponds to the line number in the log that shows the syntax that produced this NOTE. However, if the code that generates this NOTE is within a macro definition, the line number included in the NOTE is not as helpful. Here is an example in which the DATA step within the TEST macro generates the same NOTE as above, but with one key difference:

```
149
150  %macro test;
151    data a;
152      set sashelp.class;
153      new=input(sex,1.);
154    run;
155  %mend;
156  %test
MLOGIC(TEST):  Beginning execution.
MPRINT(TEST):   data a;
MPRINT(TEST):   set sashelp.class;
MPRINT(TEST):   new=input(sex,1.);
MPRINT(TEST):   run;

NOTE: Invalid argument to function INPUT at line 156 column 41.
Name=Alfred Sex=M Age=14 Height=69 Weight=112.5 new=. _ERROR_=1 _N_=1
```

Output 7. Log Output Generated by the Execution of the TEST Macro

Notice that the line number in the NOTE points to the line where the macro is invoked instead of pointing to the line that contains the INPUT function. The NOTE points to this line because it is the invocation of the macro that generates the INPUT function code that then generates the NOTE. Because there is currently no option other than having the NOTE point to the line where the macro is invoked, you can use the MFILE system option to debug.

Once MFILE has routed all the non-macro code generated by the macro to the specified file, it is much easier to debug the problem. Here is the syntax for the MFILE system option:

```
Filename mprint 'c:\mymacros\mycode.sas';
Options mprint mfile;
```

In the syntax above, notice the following points:

- The fileref is "**mprint**," which is required for the use of the MFILE option.
- The file listed within the quotation marks can have any name and be in any location.
- The MPRINT option is included in the OPTIONS statement as required for the use of the MFILE option.

The MYCODE.SAS file that is created by MFILE can now be executed outside of the macro definition. The same NOTE is still generated, but now the line number listed in the NOTE points to the exact line of code that generated the NOTE, which greatly helps the debugging process.

Although MFILE is not used as often as MLOGIC, MPRINT, and SYMBOLGEN, it can be just as useful at helping you navigate the jungle when debugging code if there are problems with the non-macro code contained in your macro definition.

READ-ONLY MACRO VARIABLES

Quick Facts

- Both global and local macro variables can be defined as read-only.
- If you omit a value when creating the read-only variable, the macro variable has a null value.

Another way to help protect your code is to use read-only macro variables. Once a macro variable has been defined as read-only, that macro variable's value cannot be overwritten, nor can the macro variable be deleted. If a macro variable is defined as read-only, it also cannot be redefined within another scope. All read-only macro variables are available until the scope in which they were defined no longer exists.

You can use a read-only macro variable to prevent users from altering certain portions of the code. Consider this example in which there are multiple macros that all reference a single macro variable that resolves to someone's user ID. To keep users from being able to change this variable's value to another user ID, you can create the macro variable as read-only. You can use the %GLOBAL statement to create the macro variable as a global macro variable so that it is available for use in all macros. Starting in SAS 9.4, you can add the READONLY option to the %GLOBAL and %LOCAL statements to define the variable(s) listed as read-only. Here is the syntax:

```
%global / readonly user_id=sasid_123;
```

Now, if any attempt is made to re-declare the macro variable USER_ID, the following error is written to the log:

```
ERROR: The variable USER_ID was declared READONLY and cannot be modified or re-
declared.
```

If an attempt is made to delete the macro variable USER_ID using %SYMDEL, this error message is written to the log:

```
ERROR: The variable USER_ID was declared READONLY and cannot be deleted.
```

Being able to protect your code is extremely important. Read-only macro variables are yet another tool that you can use to help you protect your code and keep users from altering it.

THE MACRO IN OPERATOR

Quick Facts

- The IN operator must be enabled by specifying the MINOPERATOR system option.
- The IN operator might require you to use the MINDELIMETER option.

A common programming need is the ability to determine whether a single value is contained in a specified list of values. For many releases of SAS, it has been possible to do this easily in the DATA step by using the IN operator. In the following example, if the value of MAKE is equal to any of the listed values, then the CARTYPE equals '**Luxury**':

```
data new;
   set sashelp.cars;
   if make in('Acura' 'Audi' 'Lexus' 'Infiniti' 'Mercedes-Benz') then
cartype='Luxury';
```

Prior to SAS 9.3, the following syntax was needed to make the same type of evaluation:

```
%if %bquote(&make)=Acura or  %bquote(&make)=Audi or %bquote(&make)=Lexus
or %bquote(&make)=Infiniti or %bquote(&make)=%str(Mercedes-Benz) %then %let
cartype=Luxury;
```

In the code above, note that MAKE must be compared to each possible value using the OR operator. Starting in SAS 9.3 however, the IN operator was introduced to the macro language. The following code performs the same evaluation as above, but now uses the IN operator:

```
%if %bquote(&make) in(Acura Audi Lexus Infiniti %str(Mercedes-Benz)) %then %let
cartype=Luxury;
```

As this example shows, the IN operator makes this %IF condition much easier to code.

By default, the IN operator is not available. You have to enable the IN operator by specifying the MINOPERATOR system option:

```
options minoperator;
```

Another option that you might need to specify is the MINDELIMTER= system option, which lets you specify the delimiter used between listed values. The default value for this option is a space. If you want to use any other character as a delimiter, you must set this option. Note that the MINOPERATOR and MINDELIMTER options can also be %MACRO statement options. If you want to use the IN operator only within a single macro, you can use the following syntax:

```
%macro test/ minoperator mindelimiter=',';
```

The IN operator in the macro language definitely makes coding certain %IF conditions much easier, which helps you get out of the jungle faster.

SASHELP.VMACRO

Quick Facts

- SASHELP.VMACRO is useful at keeping track of all the macro variables you create.
- SASHELP.VMACRO enables you to subset the view to see only the macro variables you are interested in.

SASHELP.VMACRO is an automatically generated view that contains a list of all the macro variables that currently exist in your SAS session. This list includes all the macro variables you have defined as well as all the macro variables that have been defined by the SAS system.

The contents of this view show you the scope, name, and value of the macro variable. The view also has a value called **OFFSET**. Because this is a view, the value of the macro variable can be stored only in 200-byte chunks. An OFFSET of **>0** indicates that this observation is a continuation of the variable listed in the previous observation. If you access the SASHELP.VMACRO view from within a macro definition, you are able to view the local macro variables as well.

You can use SASHELP.VMACRO to help keep track of macro variables when you have created hundreds of them. If you have created only a few macro variables, then you can use %PUT _USER_ for this purpose. However, if you have many variables, it is much easier to read the information about the macro variables in the output window versus in the log window.

One benefit from having the macro-variable information stored in a view is that you can use logic to create a subset of the file. For example, you can use a WHERE clause similar to the following to subset the view so that you see only macro variables with a prefix of **MAKE**:

```
proc print data=sashelp.vmacro(where=(name=:'MAKE'));
run;
```

obs	scope	name	offset	value
1	GLOBAL	MAKE1	0	Acura
2	GLOBAL	MAKE10	0	GMC
3	GLOBAL	MAKE11	0	Honda
4	GLOBAL	MAKE12	0	Hummer
5	GLOBAL	MAKE13	0	Hyundai
6	GLOBAL	MAKE14	0	Infiniti
7	GLOBAL	MAKE15	0	Isuzu
8	GLOBAL	MAKE16	0	Jaguar
9	GLOBAL	MAKE17	0	Jeep
10	GLOBAL	MAKE18	0	Kia

Output 8. Partial Output from the PRINT Procedure Statement

A helpful task that uses SASHELP.VMACRO is deleting all the existing user-defined global macro variables. The following example uses SASHELP.VMACRO and the %SYMDEL statement to delete all the user-defined global macro variables:

```
%macro delvars;
  data vars;
    set sashelp.vmacro;
  run;
  data _null_;
    set vars;
    temp=lag(name);
    if scope='GLOBAL' and substr(name,1,3) ne 'SYS' and temp ne name then
      rc=dosubl('%symdel '||trim(left(name))||';');
  run;
%mend;
%delvars
```

The first step of this code block creates a temporary data set that contains the information from SASHELP.VMACRO. This is important because VMACRO is a SASHELP view that contains information about currently defined macro variables. Creating a data set from SASHELP.VMACRO avoids a potential macro-symbol table lock.

The next step checks the variable's scope and ensures that the variable name does not begin with **SYS**. This is important because you do not want to mistakenly try to delete a macro variable defined by the SAS system. Finally, the DOSUBL function passes the macro variable name to the %SYMDEL statement, which is a macro statement that deletes global macro variables. After you run this code, all global macro variables that you created in this session will have been deleted.

The larger your application, the more likely that it is going to be complicated. SASHELP.VMACRO is an excellent tool to help you keep track of a large number of macro variables that are likely being created with a large application.

CONCLUSION

This paper has discussed ten tools that can be added to your macro programming toolbox. The tools discussed in this paper can make certain tasks in macro programming possible and can make other tasks easier. Hopefully, you can incorporate these tools into your own macro toolbox so that they can better equip you to survive the macro jungle.

REFERENCES

- SAS Institute Inc. 2015. "Saving Macros Using the Stored Compiled Macro Facility." *SAS® 9.4 Macro Language: Reference, Fourth Edition*. Cary, NC: SAS Institute Inc. Available at **support.sas.com/documentation/cdl/en/mcrolref/67912/HTML/default/viewer.htm#n0sjezyl65z1cpn1b6mqfo8115h2.htm**
- SAS Institute Inc. 2015. "CALL SYMPUTX Routine." *SAS® Functions and CALL Routines: Reference, Fourth Edition*. Cary, NC: SAS Institute Inc. Available at **support.sas.com/documentation/cdl/en/lefunctionsref/67960/HTML/default/viewer.htm#n1nexcs36ctqk5n11uao7k9myz7y.htm**
- SAS Institute Inc. 2015. "%PUT Statement." *SAS® Macro Language: Reference, Fourth Edition*. Cary, NC: SAS Institute Inc. Available at **support.sas.com/documentation/cdl/en/mcrolref/67912/HTML/default/viewer.htm#n189qvy83pmkt6n1bq2mmwtyb4oe.htm**

ACKNOWLEDGMENTS

I would like to thank my colleague, Russ Tyndall, for his assistance when I was writing this paper.

CONTACT INFORMATION

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

Kevin Russell
SAS Institute Inc.
SAS Campus Drive
Cary, NC 27513
Email: **support@sas.com**
Web: **support.sas.com**