SESUG Paper 111-2019

Tales from the Help Desk: Solutions to Common Macro and Macro Variable Issues

Bruce Gilsen, Federal Reserve Board, Washington, DC

ABSTRACT

In 35 years as a SAS ® consultant at the Federal Reserve Board, I have seen some issues related to common SAS tasks surface again and again. This paper collects the most common simple issues related to macros and macro variables from my previous "Tales from the Help Desk" papers, and provides code to explain and resolve them. The following issues are reviewed:

1. Using an array definition in multiple DATA steps.
2. Using a DATALINES statement in a macro.
3. Surrounding a macro variable with single quotes.
4. Using comments in a macro.
5. Creating a macro variable and using it in the same DATA step.
6. Having an overlapping macro variable in a main macro and a called macro.
7. Using %SYSFUNC to execute DATA step functions in a macro.
8. Using the macro IN operator to check in a macro if a value equals one of the values in a list.

In the context of discussing these issues, the paper provides details about SAS processing that can help users employ SAS more effectively. See the references for seven previous papers that contain additional common issues.

1. USING AN ARRAY DEFINITION IN MULTIPLE DATA STEPS

An array definition is only in effect in the DATA step in which it is defined and cannot be stored in a data set.

If you use the same array in multiple steps, as users often do, it can be helpful to define it in only one place. Then, if the array definition changes, you only need to change the code once.

To define the same array in more than one DATA step, you can do one of the following.

- Create a macro variable containing the array definition with a %LET statement, and use the macro variable in each DATA step.
- Create a macro containing the array definition, and execute the macro in each DATA step.

Here is an example with a macro variable.

```
    /* Create a macro variable called ARRAYDEF.  It contains an
       array definition that is used in subsequent steps. */
  %let arraydef= array gnp (*) consume invest gov tax;

    /* Use the array definition in a DATA step */
  data two;
    set one;
    &arraydef; /* define array GNP */
```

```
   /* more SAS statements */

run;
     /* Use the array definition in another DATA step */
data three;
  set one;
  &arraydef; /* define array GNP */

   /* more SAS statements */

run;
```

Here is an example with a macro.

```
  /* Create a macro called ARRAYDEF.  It contains an
     array definition that is used in subsequent steps. */
%macro arraydef;
  array gnp (*) consume invest gov tax;
%mend arraydef;

  /* Use the array definition in a DATA step */
data two;
  set one;
  %arraydef /* define array GNP */

   /* more SAS statements */

run;
        /* Use the array definition in another DATA step */
   data three;
     set one;
     %arraydef /* define array GNP */

      /* more SAS statements */

   run;
```

2. USING A DATALINES STATEMENT IN A MACRO

The DATALINES statement, which is used to read data entered directly in a SAS program, cannot be used in a macro. CARDS is an alias for DATALINES and has the same restriction.

This restriction is not noted in the DATALINES statement documentation in the *SAS® 9.4 DATA Step Statements: Reference*, but is noted in the %MACRO statement documentation in the *SAS 9.4 Macro Language: Reference*.

An example of invalid code is as follows. This macro could include code above or below the DATA step.

```
%macro blah;
  /* other statements here */
data one;
  input a b;
  datalines;
1 2
3 4
5 6
;
run;
```

```
    /* other statements here */
  %mend blah;
  %blah
```

When this code is submitted, the following error message is written to the SAS log.

```
  ERROR: The macro BLAH generated CARDS (data lines) for the DATA step, which
         could cause incorrect results.
         The DATA step and the macro will stop executing.
```

Four solutions are presented in this paper; others are available.

Solution 1. Move the DATA step to an external file and bring the DATA step into the macro with a %INCLUDE statement.

For example, put the DATA step in the UNIX file /my/include/file.

```
  data one;
    input a b;
    datalines;
  1 2
  3 4
  5 6
  ;
  run;
```

Use a %INCLUDE statement in the macro. Any code above or below the DATA step in the original macro remains in the macro, above or below the %INCLUDE statement.

```
  %macro blah;
    /* other statements here */
  %include '/my/include/file';
    /* other statements here */
  %mend blah;
  %blah
```

Solution 2. Move the data to an external file and use an INFILE statement to read the data. For example, if the data are in the external file /my/flat/file, code the macro as follows.

```
  %macro blah;
    /* other statements here */
  data one;
    infile '/my/flat/file' ;
    input a b;
  run;
    /* other statements here */
  %mend blah;
  %blah
```

Solution 3. Change the DATALINES statement to assignment statements. This is practical for small amounts of data.

```
  %macro blah;
    /* other statements here */
  data one;
    a=1;
    b=2;
    output;
    a=3;
    b=4;
    output;
    a=5;
    b=6;
    output;
  run;
    /* other statements here */
  %mend blah;
  %blah
```

Solution 4. Use PROC SQL.

```
  %macro blah;
    /* other statements here */
  proc sql;
    create table one
      (a num,
       b num);

    insert into one
      (a, b)
       values(1, 2)
       values(3, 4)
       values(5, 6)
       ;
  quit;
    /* other statements here */
  %mend blah;
  %blah
```

3. SURROUNDING A MACRO VARIABLE WITH QUOTES IN SAS CODE

Beginning users usually do not use macros, but might use macro variables. A common mistake is surrounding a macro variable with single quotes, so that it does not resolve, as in the following DATA step statement.

```
  newvar = '&macvar1';
```

The simple solution is to use double quotes, as in the following DATA step statement.

```
  newvar = "&macvar1";
```

One reason that this error occurs is that single and double quotes are equivalent in some instances, such as the following DATA step statements, so users assume that single and double quotes are equivalent in all instances.

```
  newvar = 'mynamehere';
  newvar = "mynamehere";
```

4. USING COMMENTS IN A MACRO

Note. In this section, the following terms are used.

- A *block comment*, also known as a PL/1 style comment, is a comment that begins with /* and ends with */.

```
     /* this example is on one line */

     /* this example starts on this line
```

```
1 or more lines in between
ends on this line */
```

- A *single-statement comment*, also known as an asterisk style comment, is a comment that begins with an asterisk (*) and ends with a semicolon (;).

```
* this example is on one line ;

* this example starts on this line
1 or more lines in between
ends on this line;
```

- A *macro comment* is a comment that begins with %* and ends with a semicolon (;). Though generally used in a macro, it can be used outside a macro as well.

```
%* this example is on one line ;

%* this example starts on this line
1 or more lines in between
ends on this line;
```

Data set USA has the following values.

```
Obs   year   consume   invest   gov
 1    2008      1         2       3
 2    2009      4         5       6
```

Initially, we calculate GNP in a DATA step and print the results with PROC PRINT, and the code works correctly.

```
data countryprint;
    * Calculate each year's GNP;
  set usa;
  gnp = consume + invest + gov;
run;
proc print data=countryprint;
run;
```

Then, we decide to generalize the code and put it in a macro, as follows.

```
%macro gnpprint(country=);
  data countryprint;
      * Calculate each year's GNP;
    set &country;
    gnp = consume + invest + gov;
  run;
  proc print data=countryprint;
  run;
%mend gnpprint;

%gnpprint(country=usa)
```

The macro does not invoke, and the following cryptic message is written to the SAS log.

```
WARNING: Missing %MEND statement.
```

If we re-submit this code with some additional statements, the problem becomes easier to identify.

```
%macro gnpprint(country=);
  data countryprint;
```

```
      * Calculate each year's GNP;
    set &country;
    gnp = consume + invest + gov;
    govplusconsume = gov + consume;
    govplusgovplusinvest = gov + gov + invest;
    consumeplusconsume = consume + consume;
  run;
  proc print data=countryprint;
  run;
%mend gnpprint;

%gnpprint(country=usa)
```

Now, the following is written to the SAS log.

```
WARNING: The quoted string currently being processed has become more than
         262 characters long.  You may have unbalanced quotation marks.
WARNING: Missing %MEND statement.
```

The problem is that the single quote in the word *year's* is treated as the beginning of a string literal, which is comprised of all characters until the next single quote. While character values can have a maximum length of 32,767 bytes, quoted strings longer than 262 characters generate the warning shown above. Single-statement comments (as in the code above) and macro style comments cause this problem.

To prevent this problem, use block comments in a macro. In this example, change the comment to the following.

```
/* Calculate each year's GNP */
```

For more details about the use of comments in a macro, see SAS Note 32684 (2010).

5. CREATING A MACRO VARIABLE AND USING IT IN THE SAME DATA STEP

Before executing a DATA step, the macro variable MACVAR is assigned the value "oldvalue" in either a prior DATA or PROC step or (as in the code below) in open code (not in a DATA or PROC step).

In the DATA step, the macro variable MACVAR is set to "newvalue" with CALL SYMPUT. Then, using a macro variable reference (preceding a macro variable's name with an ampersand), the DATA step variable DATASTEP_VAR1 is set to the value of MACVAR. The value of DATASTEP_VAR1, displayed with a PUT statement, is expected to be "newvalue", but it is "oldvalue".

```
  /* Create or update a macro variable before executing a DATA step */
%let macvar=oldvalue ;

data _null_ ;
    /* copy a DATA step variable to a macro variable with CALL SYMPUT */
  call symput("macvar","newvalue");
    /* use macro variable reference, macro variable value is
       still "oldvar" */
  datastep_var1 = "&macvar";
  put 'from macro variable reference: ' datastep_var1= ;
run;
```

The PUT statement writes the following text to the SAS log.

```
from macro variable reference: datastep_var1=oldvalue
```

This problem occurs because you cannot assign a value to a macro variable with CALL SYMPUT and use a macro variable reference (preceding a macro variable's name with an ampersand) to retrieve the value of the macro variable in the same step.

As explained in the *SAS 9.4 Macro Language: Reference*, CALL SYMPUT assigns the value of a macro variable during program execution, but macro variable references resolve at one of the following times.

- During step compilation
- In a global statement used outside a step
- In a SAS Component Language (SCL) program

Thus, if you assign a value to a macro with CALL SYMPUT, you cannot retrieve the value with a macro variable reference until after the DATA step finishes.

To use CALL SYMPUT and then retrieve the value of the macro variable in the same step, use the RESOLVE or SYMGET function instead of a macro variable reference. RESOLVE and SYMGET are similar, but RESOLVE accepts a wider variety of arguments, such as macro expressions.

In the following DATA step, RESOLVE and CALL SYMGET both correctly retrieve the value of the macro variable in the same step as the CALL SYMPUT. DATASTEP_VAR2 and DATASTEP_VAR3 have the value "newvalue", as expected.

```
  /* Create or update a macro variable before executing a DATA step */
%let macvar=oldvalue ;

data _null_ ;
    /* Copy a DATA step variable to a macro variable with CALL SYMPUT */
  call symput("macvar","newvalue");
  datastep_var2 = resolve('&macvar') ;   /* resolve function */
  put 'from resolve function: ' datastep_var2= ;
  datastep_var3 = symget('macvar') ;     /* call symget */
  put 'from symget function: ' datastep_var3= ;
run;
```

The PUT statements write the following text to the SAS log.

```
from resolve function: datastep_var2=newvalue
from symget function: datastep_var3=newvalue
```

6. HAVING AN OVERLAPPING MACRO VARIABLE IN A MAIN MACRO AND A CALLED MACRO

In the following code, the %DO loop in macro ONE is expected to execute (and call macro TWO) three times, but only executes once.

```
%macro one;
  %do j = 1 %to 3;
    %two;
    /* other macro code here */
  %end;
%mend one;

%macro two;
  %do j = 1 %to 2;
    /* macro TWO loop code here */
  %end;
%mend two;

%one;  /* call macro ONE */
```

To illustrate what happened, add some %PUT statements to the macros and execute macro ONE again.

```
%macro one;
  %do j = 1 %to 3;
```

```
    %put PUT 1.1: j= &j;
    %two;
    %put PUT 1.2: j= &j;
    /* other macro code here */
  %end;
     %put PUT 1.3: j= &j;
%mend one;

%macro two;
  %do j = 1 %to 2;
    %put PUT 2.1: j= &j;
    /* macro TWO loop code here */
  %end;
  %put PUT 2.2: j= &j;
%mend two;

%one;  /* call macro ONE */
```

The PUT statements write the following text to the SAS log.

```
PUT 1.1: j= 1
PUT 2.1: j= 1
PUT 2.1: j= 2
PUT 2.2: j= 3
PUT 1.2: j= 3
PUT 1.3: j= 4
```

This problem is an example of a "macro variable scope" error. It occurs because changes to the value of macro variable J in macro TWO affect the value of J in macro ONE. Note that in macro ONE, the value of macro variable J is

- 1 before macro TWO is called
- 3 after macro TWO ends

When the %DO loop in macro ONE finishes for the first time, the index variable, J, is increased from 3 to 4 instead of from 1 to 2. Since 4 is outside the loop bounds, the loop does not execute again.

The *SAS 9.4 Macro Language: Reference* devotes an entire chapter to the topic of macro variable scope. A detailed discussion is beyond the scope of this paper, but the following programming practice prevents many scope-related problems, including the one in this example.

> If a macro variable is created in a macro, and is not needed after the macro finishes executing (i.e., the macro variable is needed only in the macro or any macros called by the macro), explicitly make it a local macro variable with the %LOCAL statement.

To fix the problem in this example, make J a local macro variable in macro TWO with a %LOCAL statement, as in the following code. Macro ONE is unchanged.

```
%macro two;
  %local j;
  %do j = 1 %to 2;
    %put PUT 2.1: j= &j;
    /* macro TWO loop code here */
  %end;
  %put PUT 2.2: j= &j;
%mend two;
```

Now, the %DO loop in macro ONE executes (and calls macro TWO) three times. The PUT statements write the following text to the SAS log.

```
PUT 1.1: j= 1
PUT 2.1: j= 1
PUT 2.1: j= 2
PUT 2.2: j= 3
PUT 1.2: j= 1
PUT 1.1: j= 2
PUT 2.1: j= 1
PUT 2.1: j= 2
PUT 2.2: j= 3
PUT 1.2: j= 2
PUT 1.1: j= 3
PUT 2.1: j= 1
PUT 2.1: j= 2
PUT 2.2: j= 3
PUT 1.2: j= 3
PUT 1.3: j= 4
```

Macro variable J in macro TWO is local to macro TWO, and does not affect the value of J in macro ONE. In macro ONE, the value of macro variable J is

- 1 before macro TWO is called the first time
- 1 after macro TWO ends the first time

When the %DO loop in macro ONE finishes for the first time, the index variable, J, is increased from 1 to 2. Since 2 is inside the loop bounds, the loop continues to execute (a total of 3 times).

Notes.

1. This problem can occur with any macro variable, but is especially common with index (looping) variables with names like I, II, and J, which users seem to regard as somehow different than other variables.
2. A somewhat analogous DATA step problem is to code a loop with the index variable I, II, or J, without determining if the data set being processed already has a variable with any of those names.
3. This problem manifests itself in different ways based on the values of the index variables. For example, without the %LOCAL statement, if the loop in macro ONE was %do j = 1 %to 4;, an infinite loop occurs, because the value of macro variable J in macro ONE is
   - 1 before macro TWO is called the first time
   - 3 after macro TWO ends the first time
   - 4 after the %DO loops ends the first time
   - 4 before macro TWO is called the second time and beyond
   - 3 after macro TWO ends the second time and beyond
   - 4 after the %DO loops ends the second time and beyond
4. If macro variable J in macro ONE is local to macro ONE, it (macro variable J in macro ONE) should also be specified as local, with a %LOCAL statement in macro ONE.
5. Defining local macro variables is especially important if you code a utility macro used by multiple users or applications. Users calling your macro might not (and should not have to) know the names of macro variables that are local to your utility macro. Ensuring that your macro leaves the computing environment unchanged is good programming practice.

7. USING %SYSFUNC TO EXECUTE DATA STEP FUNCTIONS IN A MACRO

The SAS macro facility provides a modest number of functions. The %SYSFUNC function greatly increases the set of macro tools by allowing you to use all DATA step functions except the following (as of SAS 9.4): ALLCOMB, ALLPERM, DIF, DIM, HBOUND, IORCMSG, INPUT, LAG, LBOUND, LEXCOMB, LEXCOMBI, LEXPERK, LEXPERM, MISSING, PUT, RESOLVE, SYMGET, and all variable information functions such as VNAME and VLABEL.

Instead of INPUT and PUT, you can use INPUTN, INPUTC, PUTN, and PUTC, where N or C are for numeric or character.

This section shows two situations where function calls in a DATA step must be modified for use with %SYSFUNC in a macro.

QUOTING CHARACTER VALUES

Example 1 illustrates a simple use of %SYSFUNC. Example 2 illustrates a problem with quoting character values when using %SYSFUNC.

Example 1. Calculate the mean of some macro variables

First, let's illustrate the MEAN function in a DATA step. Data set ONE has the following values.

```
Obs   aa     bb
 1    10     20
 2    30     40
 3    50     60
```

In a DATA step, use the MEAN function.

```
data two;
  set one;
  meanab = mean (aa,bb);
run;
```

Data set TWO has the following values.

```
Obs   aa     bb     meanab
 1    10     20       15
 2    30     40       35
 3    50     60       55
```

Alternately, call the MEAN function in a macro with %SYSFUNC, passing in either macro variables or numbers.

```
%macro sysfunc1;
  %let aa1=10;
  %let bb1=20;
  %let aa2=30;
  %let bb2=40;

  %let meanab1 = %sysfunc( mean(&aa1,&bb1) );
  %let meanab2 = %sysfunc( mean(&aa2,&bb2) );
  %let meanab3 = %sysfunc( mean(50,60) );
%mend sysfunc1;
%sysfunc1
```

The macro variables created by this macro are as follows. These results are consistent with the DATA step calculations above.

```
Macro variable    Value
   MEANAB1         15
   MEANAB2         35
   MEANAB3         55
```

Example 2. Use the COUNTC function to count the number of times the character "z" appears in a character string

First, in a DATA step, the number of occurrences, N_COUNT, is 5.

```
data two;
  n_count =  countc("zbcdddABczzzzbc","z");
run;
```

Alternately, call COUNTC in a macro with %SYSFUNC.

```
%macro sysfunc2;
  %let nn_count = %sysfunc( countc("zbcdddABczzzzbc","z") );
%mend sysfunc2;
%sysfunc2
```

We do not get the correct answer. NN_COUNT is 7, not 5. Because %SYSFUNC is a macro function, all values are treated as character text, and character values should not be quoted, unlike in the DATA step. The quotes are treated as ordinary text characters, so

- The first argument to COUNTC, the string to check, includes the quotes, and has the following value: "zbcdddABczzzzbc"
- The second argument to COUNTC includes the quotes as values to check for, so COUNTC finds all occurrences of double quote, z, or double quote and finds them 7 times.

The %SYSFUNC documentation is misleading about this issue, suggesting that quoting character values is optional rather than incorrect. For example, according to the *SAS 9.4 Macro Reference*,

> Because %SYSFUNC is a macro function, you do not need to enclose character values in quotation marks as you do in DATA step functions. For example, the arguments to the OPEN function are enclosed in quotation marks when the function is used alone, but do not require quotation marks when used within %SYSFUNC.

To get the correct answer, remove the quotes, as in the following code.

```
%macro sysfunc3;
  %let nn_count = %sysfunc( countc(zbcdddABczzzzbc,z) );
%mend sysfunc3;
%sysfunc3
```

NESTING FUNCTIONS

Another quirk of %SYSFUNC, as discussed in Gilsen (2009), is that functions cannot be nested.

The TODAY function returns the current date as a SAS date value, and the WEEKDAY function returns the day of the week for a SAS date value (1=Sunday, 2=Monday, ... 7=Saturday). In a DATA step, the following statement assigns the day of the week for the current date to the variable WEEK_DAY.

```
week_day = weekday(today());
```

Since functions cannot be nested within %SYSFUNC, the following statement generates an error.

```
%let week_day = %sysfunc(weekday(today()));
```

The following is written to the SAS log.

```
ERROR: Required operator not found in expression: today()
ERROR: Argument 1 to function WEEKDAY referenced by the %SYSFUNC or
       %QSYSFUNC macro function is not a number.
ERROR: Invalid arguments detected in %SYSCALL, %SYSFUNC, or %QSYSFUNC
       argument list.  Execution of %SYSCALL statement or %SYSFUNC or
       %QSYSFUNC function reference is terminated.
```

To resolve the error, use %SYSFUNC once for the WEEKDAY function and once for the TODAY function.

```
%let week_day = %sysfunc(weekday(%sysfunc(today())));
```

8. USING THE MACRO IN OPERATOR TO CHECK IN A MACRO IF A VALUE EQUALS ONE OF THE VALUES IN A LIST

In the DATA step, the IN operator allows you to check if a value equals one of the values in a list, e.g.,

```
if x in (1 2 3);
```

The macro IN operator, introduced in SAS 9.2, is specified with IN or #. It allows you to check in a macro if a value equals one of the values in a list. This operator is not available by default and is enabled with the system option MINOPERATOR.

Macro IN operator syntax is similar to the DATA step IN operator, except that the list of values to check is just a list of values not enclosed in parentheses. The default separator between each value is a space. Specify a different separator with the MINDELIMITER= system option. For example, to separate with a comma, use: options mindelimiter=',';

Here is a macro that includes some examples of the macro IN operator.

```
options MINOPERATOR;     /* enable macro IN operator */
%macro intest1(macvar1=);

    /* Does macro variable MACVAR1 have the value aa, bb, or cc */
  %if &macvar1 in aa bb cc
    %then %put test 1 is in;              /* this text is printed */
    %else %put test 1 is not in;

    /* Same as the previous comparison: # is the same as IN */
  %if &macvar1 # aa bb cc
    %then %put test 2 is in;              /* this text is printed */
    %else %put test 2 is not in;

    /* Value to check is literal text instead of a macro variable */
  %if cc in aa bb cc
    %then %put test 3 is in;              /* this text is printed */
    %else %put test 3 is not in;

  %let macvar1=qq;
  %if &macvar1 in aa bb cc
    %then %put test 4 is in;
    %else %put test 4 is not in;          /* this text is printed */

%mend intest1;

%let mymacvar=bb;
%intest1 (macvar1=&mymacvar)   /* Call the macro */
```

This macro generates the following output.

```
test 1 is in
test 2 is in
test 3 is in
test 4 is not in
```

Here is a macro that does not use the macro IN operator, and has run successfully in the past.

```
%macro intest2;
  %let blah=in;
```

```
    %if &blah = ABC
      %then %put Blah matches value;
      %else %put Blah does not match value;
  %mend intest2;
  %intest2
```

This macro generates the following output.

```
  Blah does not match value
```

However, if you enable the macro IN operator with OPTIONS MINOPERATOR;, macro INTEST2 stops working and generates the following error.

```
  ERROR: Operand missing for IN operator in argument to %EVAL function.
  ERROR: The macro INTEST2 will stop executing.
```

The problem is that enabling the macro IN operator introduces an important compatibility issue: in a logical or arithmetic macro expression, a character string that begins with "#" or equals "in" (or "IN", "iN", or "In") must be masked with a macro quoting function. Otherwise, an error occurs because the macro processor treats the “#” or “in” as the macro IN operator, not ordinary text.

To fix the problem, mask BLAH with the macro quoting function %BQUOTE in the %IF statement so that the value will be seen as ordinary text. As modified, this macro generates the expected output.

```
  %macro intest2;
    %let blah=in;
    %if %bquote(&blah) = ABC
      %then %put Blah matches value;
      %else %put Blah does not match value;
  %mend intest2;
  %intest2
```

Note that the error also occurs in the original version of INTEST2 if the value of BLAH begins with "#", but does not occur if the “#” is after the first character or if “IN” is followed by other characters.

```
  %let blah=#abc;       /* the error occurs */
  %let blah=a#b#c;      /* the error does not occur */
  %let blah=INDIANA;    /* the error does not occur */
```

CONCLUSION

This paper collected the most common simple questions related to macros and macro variables from my previous "Tales from the Help Desk" papers, and provided code to explain and resolve them. It is hoped that this paper enables users to better understand SAS system processing and thus employ SAS more effectively in the future.

REFERENCES

Gilsen, Bruce (2003), "Deja-vu All Over Again: Common Mistakes by New SAS Users," *Proceedings of the Sixteenth Annual NorthEast SAS Users Group Conference.* <http://www.lexjansen.com/nesug/nesug03/bt/bt010.pdf>

Gilsen, Bruce (2007), "More Tales from the Help Desk: Solutions for Common SAS Mistakes," *Proceedings of the SAS Global Forum 2007 Conference*. <http://www2.sas.com/proceedings/forum2007/211-2007.pdf>

Gilsen, Bruce (2009), "Tales from the Help Desk 3: More Solutions for Common SAS Mistakes," *Proceedings of the SAS Global Forum 2009 Conference*. <http://support.sas.com/resources/papers/proceedings09/137-2009.pdf>

Gilsen, Bruce (2010), "Tales from the Help Desk 4: Still More Solutions for Common SAS Mistakes," *Proceedings of the SAS Global Forum 2010 Conference*. <http://support.sas.com/resources/papers/proceedings10/146-2010.pdf>

Gilsen, Bruce (2012), "Tales from the Help Desk 5: Yet More Solutions for Common SAS Mistakes," *Proceedings of the SAS Global Forum 2012 Conference*. <http://support.sas.com/resources/papers/proceedings12/190-2012.pdf>

Gilsen, Bruce (2015), "Tales from the Help Desk 6: Solutions to Common SAS Tasks," *SESUG 2015: The Proceedings of the SouthEast SAS Users Group, Savannah, GA, 2015*. <http://www.lexjansen.com/sesug/2015/72_Final_PDF.pdf>

Gilsen, Bruce (2016), "Tales from the Help Desk 7: Solutions to Common SAS Tasks," *SESUG 2016: The Proceedings of the SouthEast SAS Users Group, Bethesda, MD, 2016*. <http://www.lexjansen.com/sesug/2016/PA-105_Final_PDF.pdf>

SAS Institute Inc. (2010), SAS Note 32684, "Using comments within a macro." < http://support.sas.com/kb/32/684.html>

SAS Institute Inc. (2015), "Base SAS 9.4 Procedures Guide, Fifth Edition," Cary, NC: SAS Institute Inc.

SAS Institute Inc. (2018), SAS® 9.4 DATA Step Statements: Reference," Cary, NC: SAS Institute Inc.

SAS Institute Inc. (2016), "SAS 9.4 Statements: Reference, Fifth Edition," Cary, NC: SAS Institute Inc.

SAS Institute Inc. (2016), "SAS 9.4 Macro Language: Reference, Fifth Edition," Cary, NC: SAS Institute Inc.

ACKNOWLEDGMENTS

The following people contributed to the development of the previous "Tales from the Help Desk" papers, and thus to this paper: Mike Doughty, Donna Hill, Bill Jackson, Heidi Markovitz, and Steve Taubman at the Federal Reserve Board; Harry Droogendyk at Stratia Consulting Inc.; Sue Douglass; Lisa Eckler at Lisa Eckler Consulting Inc.; Avi Peled at the Federal Reserve Bank of Philadelphia; Bryan Beverly, Ronald Johnson, and David Swanson at the Bureau of Labor Statistics; Mike Rhoads; and Jason Secosky at SAS Institute. Their support is greatly appreciated.

CONTACT INFORMATION

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

Bruce Gilsen
Federal Reserve Board, Mail Stop N-122, Washington, DC 20551
202-452-2494
bruce.gilsen@frb.gov