SESUG 2020 Paper 107

Tales from the Help Desk: How the Problems Get Solved

Bruce Gilsen, Federal Reserve Board, Washington, DC

ABSTRACT

In 35 years as a SAS ® consultant at the Federal Reserve Board, I've answered thousands of SAS-related questions. Some of the most common problems I've encountered were discussed in seven previous "Tales from the Help Desk" papers. In this paper, I go "under the hood" to discuss the techniques I use to help solve SAS problems. Topics include the following.

1. Look at the SAS log.
2. Make it a SAS question.
3. Program control: execute the minimum amount of code.
4. Minimize data required to solve problems.
5. Use temporary files.
6. Write DATA step values.
7. Write macro variables and debug macros.
8. My program worked on Friday and I swear, I haven't changed a thing!

This paper has two purposes.

1. My techniques might be useful to other people who help SAS users or even users of other languages.
2. It might encourage people to think about how they solve problems and share their techniques.

INTRODUCTION

As I answered thousands of SAS-related questions over the past 35 years, techniques I found useful were added to my problem-solving toolkit and used for years without me ever thinking about it. To help prepare this paper, I spent some time "observing myself" solving problems so I could identify and document these techniques.

An underlying theme in the techniques described in this paper is simplification. In some cases, I run the user's code, or a subset of the user's code, using either their data or a small data set I create. In other cases, I send the user a small block of diagnostic code to insert in their program. But simplification is always key.

My approach to solving problems is admittedly subjective. What works for me might not work for other people. But I hope this paper is useful to other people who help SAS users and even users of other languages, and encourages them to share their problem-solving techniques.

LOOK AT THE SAS LOG

This is largely self-evident and the details are covered quite effectively in other conference papers, so this section is limited to a brief comment and some information on system options.

WHERE TO LOOK

Users often focus on the missing or incorrect data set at the end of the process or at the error messages at the bottom of the log. Whereas I start at the top of the log and search for "ERROR" and "WARNING" to find the first problem. The first problem often causes subsequent problems so it's important to start from the top, not the bottom.

SYSTEM OPTIONS

The following system options control the amount of text written to the SAS log.

- MSGLEVEL=I (versus MSGLEVEL=N, the default).
  - Additional notes about index usage, merging, sorting, data type conversion, and more are written to the SAS log.
- NOTES (versus NONOTES).
  - SAS notes are written to the SAS log.
  - This is the default and seems to be in effect in most cases.
- SOURCE (versus NOSOURCE)
  - Source statements are written to the SAS log.
  - This is the default and seems to be in effect in most cases.
- SOURCE2 (versus NOSOURCE2, the default).
  - Source statements from files included by %INCLUDE statements and autocall macros are written to the SAS log.
  - Production jobs commonly utilize %INCLUDE statements so this option can be useful.

MAKE IT A SAS QUESTION

REMOVE THE CONTEXT

Users aren't writing SAS code in a vacuum; they're trying to solve a problem in statistics or economics or something else. The application is the user's focus and what they find interesting, so they typically want to start by explaining their application, which usually doesn't help me solve the problem. In the interest of simplification, I generally try to re-direct them to think about and express their problem as a SAS problem. This often requires a bit of adjustment on the part of the user, but I've noticed some repeat customers adopting this approach by default for subsequent questions.

MINIMIZE NON-SAS COMPONENTS

If a SAS system command (X, CALL SYSTEM, %SYSEXEC, and so on) calls other software, I remove the call or replace it with something trivial.

If other software (for example, a driver script or another language's system command) invokes SAS, I try to determine the minimum data needed by SAS and invoke SAS standalone, with data entry at the top of the SAS code as follows.

- If the number of inputs to SAS is manageable, I replace them with %LET statements.
- Otherwise, I typically add a DATA step that contains data values and reads the data with a DATALINES (or CARDS) statement.

Besides allowing debugging to take place in a more purely SAS context, this also eliminates the possibility that the interaction with other software is causing the problem, as sometimes happens.

PROGRAM CONTROL: EXECUTE THE MINIMUM AMOUNT OF CODE

COMMENTING BLOCKS OF CODE: THE SKIP MACRO

I try to execute as little code as possible. Using SAS comments to comment out blocks of code can sometimes cause problems, as in the following.

- Asterisk style comments (begin with an asterisk (*) and end with a semicolon (;)) can cause errors in macros, as discussed in Gilsen (2012). And, adding asterisks before all statements that end in a semicolon is unwieldy for more than a few lines of code.
- Block comments, also known as PL/1 style comments (begin with /* and end with */) cannot be nested so errors occur if the code to be excluded already contains any block comments, as discussed in Gilsen (2003) and earlier in Grant (1994).

To comment out a block of code, I turn it into a macro that is never executed. By convention, the macro is called SKIP, as discussed in Gilsen (2003) and earlier in Grant (1994).

In the following code, a DATA step is commented out by turning it into a macro called SKIP. The code between the %MACRO and %MEND statements is treated as part of macro SKIP and not executed unless you invoke SKIP by coding the statement %SKIP.

```
data two;
  set one;
  var2 = var1 * 100;
run;
%macro skip;
  data three;
    set two;
      /* multiply var2 by 1000 */
    var2 = var2 * 1000;
  run;
%mend skip;
```

To comment out multiple blocks of code, the same macro name can be used multiple times. Any valid macro name can be used, as long as your application does not call a macro by that name.

ENDSAS

To execute a non-interactive program up to a certain point and then stop, I add an ENDSAS statement where execution should stop.

```
(Code to run)
endsas;
(Code to not run)
```

This approach does not work for interactive sessions because ENDSAS causes all windows to close before ending the SAS session.

MINIMIZE DATA REQUIRED TO SOLVE PROBLEMS

The data required to solve problems falls into four categories, in increasing levels of complexity.

- No data are required.
- Data are required but the problem is not data-specific.
- The structure of the data but not the values matter.
- Specific data values matter.

I always try to minimize the amount of data used in the interest of simplicity.

NO DATA REQUIRED

This is the least common situation, and no further discussion is required.

DATA REQUIRED BUT THE PROBLEM IS NOT DATA-SPECIFIC

Data Values

My approach to data values in this case is as follows.

- Use a small number of values.
- When possible, use integers, which I find easier to think about and display.
- If the values don't matter at all, use ascending numbers (1, 2, 3, ….).
- Make the data almost but not exactly square (for example, 3 rows and 2 columns). If the data is subsequently transposed, not being square makes it easier programmatically and visually to follow the flow of data.
- Use simple variable names. Common choices include the following.
  - var1 var2 or v1 v2
  - aa bb
  - char1 char2
  - num1 num2
- The most common data set I use is the following.

```
var1 var2
 1    2
 3    4
 5    6
```

- If I need more columns, I find the data easy to follow if values in successive columns either repeat the digit an additional time or add an additional zero, as in the following.

```
var1 var2 var3            var1 var2 var3
  1   11   111              1   10   100
  2   22   222              2   20   200
  3   33   333              3   30   300
  4   44   444              4   40   400
```

The data set on the right works well for arithmetic operations. For example, dividing one column by another yields integer results.

Data Creation

My approach to data creation in this case is as follows.

- It's self-contained at the top of the program. This helps create modular code; the user can insert their own data creation code at the top and otherwise run the program as-is.
- It's not operating system or execution mode (interactive or batch) specific.
- If the code is not in a macro and the data consists of numeric values and simple character values (no embedded spaces, no special characters) I use a DATALINES statement, as in the following examples.

```
data one;
  input var1 var2;
  datalines;
1 2
3 4
5 6
;run;

data one;
  length char1 $10;
  input num1 num2 char1 $;
  datalines;
1 2 ibm
3 4 att
5 6 microsoft
;run;
```

- For more complex character data or data in a macro (where the DATALINES statement does not work), I use a LENGTH statement to prevent truncation and OUTPUT statements to create observations.

```
data one;
  length char1 $10 char2 $20;
  char1 = "ibm";
  char2 = "chicago";
  output;
  char1 = "microsoft";
  char2 = "new york city";
  output;
;run;
```

THE STRUCTURE OF THE DATA BUT NOT THE VALUES MATTER

In some cases, users cannot provide relevant SAS data sets that contain confidential information. Sometimes, the problem can be identified or resolved with innocuous data values that preserve the structure of the confidential data, or the confidential values are in variables unrelated to the problem. For such cases, I developed a simple macro, REVALUE (Gilsen, 2016a), to mask data values in a way that preserves much of the structure of the data. Users can execute the REVALUE macro and email me the resulting data set.

The REVALUE macro is provided in the 2016 paper, or if preferred, other masking algorithms can be found on the internet.

SPECIFIC DATA VALUES MATTER

In these cases, some or all of the user's data is required to replicate or resolve the problem. If only some data is needed, I try to simplify as follows.

- Use a DROP or KEEP statement to process only variables needed for the problem.
- Run the code with just the first few observations and see if the problem persists.
  - Limit data to the first ten observations as follows.
    ```
    options obs=10;run;
    ```
  - Subsequently resume processing all observations as follows.
    ```
    options obs=max;run;
    ```
- Use interval halving to iteratively remove half the data, as follows.
  - Remove the first half of the data and run.
    - If the problem persists, remove half the remaining data.
    - If the problem disappears, restore the just removed data and run again.
  - Repeat until the data is sufficiently small or no further data can be successfully removed.

USE TEMPORARY FILES

Using temporary files has the following advantages.

- File-related operations are not operating system specific.
- Permission issues can be avoided.
- Overwriting something useful is less likely.
- Files are not left behind after the code executes.

I use temporary files as follows.

- write SAS data sets to the WORK library instead of a permanent location

- use FILENAME with the TEMP argument to create a temporary file that exists only as long as the filename is assigned. Here is an example of a typical user FILENAME statement and a corresponding FILENAME statement with the TEMP argument.

```
filename mydata1 'some-external-file';
filename mydata1 temp;
```

- write files requiring a physical name to the WORK library, which is a directory in Linux and a folder in Windows. In the following code, DATA step function PATHNAME returns the WORK library path, which is used to specify the location of an Excel file. PATHAME is executed via %SYSFUNC, which allows most DATA step functions to be executed in a macro or in open code (outside of any step).

```
%let work_library=%sysfunc(pathname(work,L));
filename __excel1 "&work_library/__excel1.xlsx";
proc export data= one outfile= "__excel1" dbms=xlsx replace;
  sheet="one";
run;
```

WRITE DATA STEP VALUES

An important part of resolving problems is writing data values to the SAS log or a file as a diagnostic tool. The challenge can be to keep the results simple and readable.

WRITE DATA STEP VALUES TO THE SAS LOG: NAMED OUTPUT

My favorite tool for writing values to the SAS log in a DATA step is Named Output with the PUT statement, which easily writes a few variables in a simple format.

The simplest use of named output is as follows.

```
put var1= var2= …;
```

The resulting output is written to the SAS log.

```
var1=value var2=value …;
```

While named output can utilize more advanced PUT statement tools like column pointer controls and formats, I usually find a simple variable list to be quick, easy, and sufficient. It could include the following.

- _N_, the number of times the DATA step has iterated, which for straightforward DATA steps is the observation number
- array references, where the actual variable name associated with the array element is printed

Here is a small example of named output. There's an indeterminate number of AGE-related variables, and a few non-positive values cause errors. The code loops through an array containing all AGE-related variables and writes diagnostic information to the SAS log when a non-positive age is found.

```
data _null_;
  set agedata;
    /* Array has all numeric variables whose names start with AGE */
  array all_agevars (*) age:;
  do i=1 to dim(all_agevars);
    if all_agevars(i) <= 0 then put _n_= all_agevars(i)=;
  end;
run;
```

Suppose data set AGEDATA has the following values.

```
ID  AGE1  AGE2  AGE3  INCOME  TAX
1    10    20    30    100    20
2    11   -99   -33    200    40
3    12    22     0    300    50
4    13    23    35    400    60
```

The following is written to the SAS log. Note that ALL_AGEVARS(I) in the PUT statement resolves to the actual variable name, AGE2 or AGE3.

```
_N_=2 AGE2=-99
_N_=2 AGE3=-33
_N_=3 AGE3=0
```

WRITE DATA STEP VALUES TO A SAS DATA SET: CALL VNAME

To write the previous section's results to a SAS data set, modify the code to do the following whenever a non-positive AGE is found.

- Use the CALL VNAME routine to assign the variable name (in this case the actual variable name associated with the array element) as the value of character variable BAD_VARIABLE_NAME.
- Assign the value of _N_ to the variable OBS_NUMBER.
- Assign the bad variable value to the variable VALUE.
- Use an OUTPUT statement to write an observation to the data set.

```
data badvalues;
  set agedata;
  length bad_variable_name $32;
  keep obs_number bad_variable_name value;
  array all_agevars (*) age:;
  do i=1 to dim(all_agevars);
    if all_agevars(i) <= 0 then do;
        /* Negative value found, CALL VNAME sets variable
           BAD_VARIABLE_NAME to name of current array element. */
      call vname(all_agevars (i),bad_variable_name);
      obs_number=_n_;
      value= all_agevars(i);
      put _n_= all_agevars(i)= value=;
      output;
    end;
  end;
run;
```

Data set BADVALUES has the following values.

```
BAD_VARIABLE_NAME   OBS_NUMBER   VALUE
      AGE2              2         -99
      AGE3              2         -33
      AGE3              3           0
```

A LENGTH statement for BAD_VARIABLE_NAME is required. If omitted, BAD_VARIABLE_NAME is defined as a numeric variable, every call to CALL VNAME sets VARNAME to a numeric missing value (.), and the following warning is printed in the SAS log. The length, 32, is the maximum length of a variable name.

```
WARNING 716-185: Argument #2 is a numeric variable, while a character
variable must be passed to the VNAME subroutine call in order for the
variable to be updated.
```

WRITE DATA STEP VALUES TO THE SAS LOG: PUT _ALL_

I infrequently write all current variables to the SAS log in a DATA step with PUT _ALL_;. When there's more than a few variables, I find it verbose and hard to read compared to targeting specific variables with named output. I find PUT _ALL_; most useful in two cases.

- To debug INPUT statements, identifying the first of multiple missing values can help pinpoint which field is being read improperly.

- Errors such as missing semicolons or improperly specified comments can lead to unexpected variables such as SAS keywords or comment text. PUT _ALL_; displays the unexpected variables, which can help identify the problem.

WRITE DATA STEP VALUES: OTHER APPROACHES

I sometimes use conditional logic to limit the amount of output for readability, as in the following examples.

- when a bad value is encountered
  ```
  if age <= 0 then put _n_= age= var1= var2=;
  ```
- every so many observations (every 100th observation in this case)
  ```
  if mod(_n_,100) = 0 then put _n_= age= var1= var2=;
  ```

I tried the little-used DATA step debugger many years ago, but it seemed too focused on interactive debugging for my needs.

WRITE MACRO VARIABLES AND DEBUG MACROS

WRITE MACRO VARIABLES

The macro variable equivalent of named output with the %PUT statement (though not known by that name) has the following syntax.

```
%put &=macrovar1 &=macrovar2 ….;
```

The following code creates and then displays two macro variables.

```
%let part1=the meaning;
%let part2=of life;
%put &=part1 &=part2;
```

The following is written to the SAS log.

```
PART1=the meaning PART2=of life
```

MACRO DEBUGGING OPTIONS

The following system options all cause text that can help debug macros to be written to the SAS log.

- MLOGIC traces the flow of execution of a macro.
- MLOGICNEST adds nesting information (if the current macro was called by other macros) to output generated by option MLOGIC.
- MPRINT writes each SAS statement generated by a macro.
- MPRINTNEST adds nesting information (if the current macro was called by other macros) to output generated by option MPRINT.
- SYMBOLGEN shows the resolution of macro variables.

To use these options, enter the following statement before running a macro.

```
options mlogic mlogicnest mprint mprintnest symbolgen;
```

These options can generate a lot of text in the SAS log, and can be turned off after debugging is finished or after the section of a program being debugged with the following statement.

```
options nomlogic nomlogicnest nomprint nomprintnest nosymbolgen;
```

An additional macro debugging option, MFILE, is discussed in the next section.

DEBUG SAS CODE IN MACROS

The Problem

Debugging SAS code contained in a macro can be frustrating because the SAS error messages do not refer to the line where the error occurred.

- In Linux and in Windows without the Enhanced Editor, error messages refer to the line in the SAS log where the macro was invoked.
- In Windows with the Enhanced Editor, there is no set rule on how line numbers are generated when errors occur inside a macro. In SAS 9.4TS1M4, they appear to refer to raw source code lines in the step containing the error but that might not always be true.

As discussed in Gilsen (2014), a convenient way to "de-macroify" SAS code in a macro is to specify the MPRINT option (discussed above) and the MFILE option along with the fileref MPRINT and write just the SAS code generated by the macro to a file, then submit the generated code.

Consider the following small example. Data set ONE includes the following values.

```
X1  X2  X3
1   2   3
4   5   6
```

The DATA step below contains an error. Array XALL has 3 elements, but the second DO loop iterates from 1 to 4, so the array subscript is out of range when I = 4.

Macro variable MYVARS contains the list of variables used in the DATA step calculations. In this example it is coded manually, but could be created dynamically earlier in the program, so the number of variable names and their values isn't assumed to be known.

```
%let myvars=x1 x2 x3;
%macro mac1 (vars=);
  data two;
    set one;
    array xall (*) &vars;
    do i=1 to 3;
      xall(i) = xall(i) + 100;
    end;
    do i=1 to 4;
      xall(i) = xall(i) + 1000;
    end;
  run;
%mend mac1;
%mac1 (vars=&myvars);
```

In Linux SAS 9.4TS1M4, the SAS log shows that the error occurred at line 20, where the macro was invoked, rather than at the line of code that caused the error. Note that lines 1-7 of the SAS log are for the DATA step that created data set ONE and that line numbers differ in Windows with the Enhanced Editor.

```
8    %macro mac1 (vars=);
9      data two;
10       set one;
11       array xall (*) &vars;
12       do i=1 to 3;
13         xall(i) = xall(i) + 100;
14       end;
```

```
15       do i=1 to 4;
16         xall(i) = xall(i) + 1000;
17       end;
18     run;
19   %mend mac1;
20   %mac1 (vars=&myvars);

ERROR: Array subscript out of range at line 20 column 141.
X1=1101 X2=1102 X3=1103 i=4 _ERROR_=1 _N_=1
NOTE: The SAS System stopped processing this step because of errors.
```

The macro had only one small DATA step and two statements with array references, so debugging the program might not be difficult. In real life, a macro could contain a very large DATA step or multiple steps or could be building the SAS code (and thus include macro variables or be generated in macro loops), making it difficult to identify the problem.

Solution: Write the code to a file with MPRINT and MFILE

The system option MPRINT writes the text (SAS statements) generated by a macro to the SAS log. If you specify options MPRINT and MFILE and the fileref MPRINT, the SAS statements displayed by the MPRINT system option are also written to a file for easy use.

To use options MPRINT and MFILE, enter the following statements before running a macro. The fileref MPRINT is required.

```
filename mprint 'external-file-for-SAS-statements';
options mprint mfile;
run;
```

The MPRINT option can generate a lot of text in the SAS log, so after writing the SAS code to a file you can turn off MPRINT and MFILE as follows.

```
filename mprint clear;
options nomprint nomfile;
run;
```

Now, submit the same macro as before, preceded by options MPRINT and MFILE and fileref MPRINT.

```
%let myvars=x1 x2 x3;
options mprint mfile;
filename mprint '/home/m1xxx00/sasfile1.sas';
run;
%macro mac1 (vars=);
  data two;
    set one;
    array xall (*) &vars;
    do i=1 to 3;
      xall(i) = xall(i) + 100;
    end;
    do i=1 to 4;
      xall(i) = xall(i) + 1000;
    end;
  run;
%mend mac1;
%mac1 (vars=&myvars);
```

The file '/home/m1xxx00/sasfile1.sas' contains SAS code identical to the DATA step executed in macro MAC1 except for the indentation. Note that macro variable VARS has been resolved (replaced by its value).

```
data two;
set one;
array xall (*) x1 x2 x3;
do i=1 to 3;
xall(i) = xall(i) + 100;
end;
do i=1 to 4;
xall(i) = xall(i) + 1000;
end;
run;
```

If we submit the DATA step that generates data set ONE and the code in '/home/m1xxx00/sasfile1.sas', the SAS log now helpfully shows that the error is due to an array reference at line 15. We can review that line of code and easily fix the program.

```
8 data two;
9 set one;
10 array xall (*) x1 x2 x3;
11 do i=1 to 3;
12 xall(i) = xall(i) + 100;
13 end;
14 do i=1 to 4;
15 xall(i) = xall(i) + 1000;
16 end;
17 run;

ERROR: Array subscript out of range at line 15 column 11.
X1=1101 X2=1102 X3=1103 i=4 _ERROR_=1 _N_=1
NOTE: The SAS System stopped processing this step because of errors.
```

MY PROGRAM WORKED ON FRIDAY AND I SWEAR, I HAVEN'T CHANGED A THING!

This common user proclamation is sort of the "Check is in the mail" of help desk support. Upon interrogation, it often turns out that the user made some small change that they're sure could not have caused the error, but it did.

In cases when the user genuinely did not change anything, here are some possible causes of the problem that I consider.

PERMISSIONS

Network administrators sometimes change file permissions without notifying users, causing programs to suddenly stop working. Sometimes the cause is more nuanced, as in the following examples.

- The user changes jobs and a Linux file they write to is owned by a group to which they no longer belong.
- An output file is shared by multiple users. Another user wrote to the file and now the permissions prevent this user from writing to the file.

DATA CHANGES

If data used in the program is maintained by other people, the format of the data could have changed without notification. If this seems like a possible cause, I encourage the user to contact the data provider.

RESOURCE LIMITS IN CODE

Programs that run periodically can stop working when they exceed resource limits in the code. For example, the following code in a monthly application worked until the 100th month, when the 2. format was too small for the value 100 and the macro variable NUM_MONTHS was not created correctly. Changing 2. to 3. fixed the problem (until the 1000th month).

```
call symput('num_months',compress(put(num_months,2.)));
```

SYSTEM RESOURCE LIMITS

Some programs run periodically (every week, month, or quarter, for example) and each time they run, adding a new period's data slightly increases the total amount of data and the total amount of memory, time, and disk space. At some point, the small data increase can cause the program to run out of memory, time, or disk space, but this is not self-evident because the resource increase each period is so small.

SAS SOFTWARE UPGRADE

Programs run infrequently might not run until quite a while after a SAS upgrade, so I work with network administrators to ensure that the prior SAS release is available for an extended time after upgrades.

Users sometimes focus on the upgrade as the likely culprit, and the first thing I have them do is run their program with the prior SAS release to determine if the new release is really the issue.

It's more difficult to debug programs after an upgrade when the prior release is not available, and I strongly encourage other sites to retain the prior release as we do. If necessary, usage can be logged or locked down except on an as-needed basis to prevent stragglers from not migrating to the current release.

OPERATING SYSTEM CHANGE

Users are not always notified about operating system changes, and sometimes ignore notifications that they do receive, perhaps because they don't understand them or don't think they apply to them.

As with SAS software upgrades, I lobby for availability of the old operating system after the cutover, though I have less leverage than with SAS releases. If the old operating system is available on a limited basis after an upgrade, for example on a few backup Linux servers or a Windows virtual machine, I have the user run their program on the old operating system to verify that the operating system upgrade really caused the problem.

CONCLUSION

This paper described the techniques I use to help solve user SAS problems. I hope this paper is useful to other people who help SAS users and even users of other languages, and I hope it encourages people to think about how they solve problems and share their techniques as well.

REFERENCES

Gilsen, Bruce (2014), "Debugging SAS Code in a Macro," Proceedings of the SAS Global Forum 2014 Conference.
<http://support.sas.com/resources/papers/proceedings14/1302-2014.pdf>

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 (2016a), "Masking Data to Obscure Confidential Values: a Simple Approach," Proceedings of the SAS Global Forum 2016 Conference.
<http://support.sas.com/resources/papers/proceedings16/3301-2016.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 (1996), "SAS Arrays: A Basic Tutorial," *Proceedings of the Ninth Annual NorthEast SAS Users Group Conference*. <https://www.lexjansen.com/nesug/nesug96/NESUG96025.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 (2016b), "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>

Grant, Paul (1994), "The 'SKIP' Statement," *Proceedings of the SAS Users Group International 19 (SUGI 19) Conference*.
<https://support.sas.com/resources/papers/proceedings-archive/SUGI94/Sugi-94-23%20Grant.pdf>

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

SAS Institute Inc. (2017), "Base SAS 9.4 Procedures Guide, Seventh 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 this paper: Donna Hill, Heidi Markovitz, and Peter Sorock at the Federal Reserve Board. 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