Paper BB-13-2015

Tips and Tricks for Getting the Most Out of the SAS® Macro Facility

Scott A. Miller, Wells Fargo, Des Moines, IA

ABSTRACT

The Base SAS macro facility is a powerful tool to enable your SAS programs to work dynamically with data. However the richness of this tool can be very confusing, and even experienced SAS coders can be mystified by macros. Learn a variety of ways to improve your coding technique with SAS macro programs and macro variables.

INTRODUCTION

Using macros can improve programmer efficiency by eliminating manual tasks, improving a program's flexibility, improved organization by eliminating code duplication, and facilitate code re-use. Furthermore, there are some problems that are impossible or impractical to solve without using macros and macro variables. Ultimately, the best use of the macro facility is about doing more work with less effort. This paper describes many methods the author has developed for getting the most out of SAS macros and macro variables. By adding these tools to your toolbox, you can boost your efficiency while improving quality and eliminating defects.

The ideas presented in this paper are intended to be used as suggestions. They may not work for every application, or may need to be adapted to fit your situation or circumstances. Use these ideas as you see fit to get the most out of your SAS programs.

1. PARAMETERIZE YOUR PROGRAM WITH MACRO VARIABLES

A typical SAS program will have one or more SAS constants representing some parameters of the question you are trying to answer. For example, you might need to focus on data from a certain range of dates or values above or below a certain threshold. I think of these types of values as the parameters for your program, and they are the kind of things that you may be required to change once you produce your finished program. One way to organize your program is to store these parameters in macro variables at the top of your program. This has a few advantages: 1) with the parameter values at the top of the program, it is easy to review the current settings and make any changes necessary, and 2) the same parameter may be used several times throughout the program – by having the value stored only once in your program, you know that the same value will be consistently used throughout the program.

A good practice is to use comments to indicate where the parameter section begins and ends so that parameter changes are limited to that section, while leaving the rest of the program unchanged. Here's an example of what this might look like:

```
*****  BEGIN -- program parameters  *****;
%let  Monthly_begin_date = '01JAN2015'd;
%let  Monthly_end_date = '31JAN2015'd;

%let  Systolic_BP = 150;
%let  Diastolic_BP = 90;

%let  Busiest_US_Airports = "ATL" "LAX" "ORD" "DFW" "JFK";
*****  END -- program parameters  *****;
```

2. NAMING VARIABLES AND MACROS

Avoid using commonly-used SAS keywords for your macro and macro variable names. If you do, you will find it difficult to trace the macro or macro variable through your program due to all the false hits from the keyword, and it will be similarly difficult to search for that keyword. It's OK to use a keyword as part of your name as you can use the "Match whole word only" check box when searching to avoid false hits.

Here's a partial list of some of the most commonly used SAS keywords, functions, options, operators, and auto variables that you might be tempted to use, but should avoid.

> DATA, RUN, FILE, END, FIRST, LAST, VAR, IN, OUT, OBS, DAY, MONTH, YEAR, MIN, MAX, AVG, SUM, LT, GT, LE, GE, EQ, NE, AND, OR

Macro and macro variable names can be up to 32 characters long. Good names will make your code more readable and easier to understand. If you use an abbreviation as part of the name, consider explaining the abbreviation in comments at the first use.

3. CREATE DATE MACRO VARIABLES AUTOMATICALLY

A common task in reporting is to select data based on a date or range of dates. The easiest way to do this is to embed the date or dates as hard coded values within the program. While this requires the least amount of work up front, over time, the manual task of modifying the dates each time the report needs to be re-run will add up. While it's nice to be needed at your job, you might want to take some time off and this requires training someone else how to update these dates. Ultimately, you may be required to document all the steps to build the report, and this is an additional burden. Additionally, it's easy to make a mistake, such as when updating a monthly report from June 1 through June 30, you might end up with July 1 through July 30, accidentally dropping a day's worth of data.

Fortunately it's easy to automate the creation of one or more macro variables containing these dates.

TODAY FUNCTION

The Today function provides you with the current date. When creating the date variables for your program, I would recommend calling Today only once and storing that value in a variable named something like report_date, and then use that value to create all of your other date variables. This way, if you need to recreate a report for a prior date, such as for auditing or to replace a defective report from a few days ago, you only need to manually override one value.

INTNX FUNCTION

For a daily report, to compute yesterday's date, just subtract 1 from today's date. For just about everything else, you'll need something more complicated.

I've seen many creative ways to calculate report dates, and many of them were buggy and/or incomprehensibly complicated. The Intnx function provides an easy way to compute date intervals, and is easier-to-read and less likely to contain bugs than a home-brewed solution.

As an example, here's code to create macro variables for the prior full month.

```
data _null_;
   report_date = today ();

   Monthly_begin_date = intnx ('month', report_date, -1, 'begin');
   Monthly_end_date = intnx ('month', report_date, -1, 'end');

   call symput ('Monthly_begin_date', Monthly_begin_date);
   call symput ('Monthly_end_date', Monthly_end_date);
run;
```

CREATE HUMAN-READABLE DATE VALUES

The code in the previous section will create macro values containing a number corresponding to the day. These values will appear in many places throughout the logs, which might be important while chasing down bugs. Here's a sample log for the month of August 2015:

```
NOTE: There were 0 observations read from the data set SASHELP.CITIDAY.
      WHERE (date>=20301 and date<=20331);
```

Looking at this log, I might wonder "am I getting 0 observations because there's no data, or because I'm using the wrong date range?" With this in mind, I don't find "20301" and "20331" to be all that helpful.

With a small change, we can create values that look like standard SAS date constants.

```
   call symput ('Monthly_begin_date',
                "'" || put (Monthly_begin_date, date9.) || "'d");
   call symput ('Monthly_end_date',
                "'" || put (Monthly_end_date, date9.) || "'d");
```

Now the log will look like this:

```
NOTE: There were 0 observations read from the data set SASHELP.CITIDAY.
      WHERE (date>='01AUG2015'D and date<='31AUG2015'D);
```

This provides a lot better starting point for debugging.

EXAMPLES

Here's a full range of examples ready for use.

```
data _null_;
   report_date = today ();

   Daily_date = report_date - 1;

   *  For a week starting on Sunday and ending on Saturday. *;
   Weekly_begin_date = intnx ('week', report_date, -1, 'begin');
   Weekly_end_date = intnx ('week', report_date, -1, 'end');

   *  For a week starting on Friday and ending on Thursday. *;
   *  See Intnx documentation for other weeks.              *;
   Weekly_begin_date = intnx ('week.6', report_date, -1, 'begin');
   Weekly_end_date = intnx ('week.6', report_date, -1, 'end');

   Monthly_begin_date = intnx ('month', report_date, -1, 'begin');
   Monthly_end_date = intnx ('month', report_date, -1, 'end');

   Quarterly_begin_date = intnx ('quarter', report_date, -1, 'begin');
   Quarterly_end_date = intnx ('quarter', report_date, -1, 'end');

   *  For a year begining on January and ending on December.  *;
   Yearly_begin_date = intnx ('year', report_date, -1, 'begin');
   Yearly_end_date = intnx ('year', report_date, -1, 'end');

   *  For a year begining on July and ending on June.       *;
   Yearly_begin_date = intnx ('year.7', report_date, -1, 'begin');
   Yearly_end_date = intnx ('year.7', report_date, -1, 'end');

   call symput ('Daily_date', "'" || put (Daily_date, date9.) || "'d");
   call symput ('Weekly_begin_date',
                "'" || put (Weekly_begin_date, date9.) || "'d");
   call symput ('Weekly_end_date',
                "'" || put (Weekly_end_date, date9.) || "'d");
   call symput ('Monthly_begin_date',
                "'" || put (Monthly_begin_date, date9.) || "'d");
   call symput ('Monthly_end_date',
                "'" || put (Monthly_end_date, date9.) || "'d");
   call symput ('Quarterly_begin_date',
                "'" || put (Quarterly_begin_date, date9.) || "'d");
   call symput ('Quarterly_end_date',
                "'" || put (Quarterly_end_date, date9.) || "'d");
   call symput ('Yearly_begin_date',
                "'" || put (Yearly_begin_date, date9.) || "'d");
   call symput ('Yearly_end_date',
                "'" || put (Yearly_end_date, date9.) || "'d");
run;
```

4. SET MACRO VARIABLES FROM EXTERNAL DATA SOURCE

Once you parameterize your program by setting macro variables for all the values you might change, it's relatively painless to change your program – just update the %Let statements whenever you need to change a value. However, this still may come up short of ideal for two reasons: 1) the person or group driving the changes may not be technically savvy, forcing you, the SAS programmer, to do the manual work whenever something must be changed, and 2) some groups institute controls on program code that requires a laborious process every time a change is made.

One way to mitigate this is to set some or all of your macro variables from an external data source, such as a .CSV file or a database table. The Call Symput function allows us to set macro variables dynamically at run time. You may wish to exert some control over what is set, such as only permitting certain variable names to prevent the data source from overwriting other values you don't want changed.

Here's an example that uses a CSV file to set macro variables, limiting the variables to only those on the permitted list. It should be noted that if a variable name appears more than once in the CSV, later values will overwrite earlier ones.

```
filename MVar_CSV "/your_folder/your_file.csv";

data _null_;
   infile MVar_CSV  DSD;

   length variable_name $32  variable_contents $100;
   input variable_name  variable_contents;

   *  Only allow variable names in the list.  *;
   if variable_name in ('some_value1', 'some_value2');

   *  Set the macro variable  *;
   call symput (variable_name, variable_contents);
run;
```

5. BROWSING MACRO VARIABLES

I find it is very useful to be able to browse through a list of all the macro variables that are set on your SAS system. Between what SAS provides and what is set by your SAS administrators, there's amazing amount of information waiting to be discovered. These commands can also be crucial during debugging. You can display all macro variables with:

```
%PUT _ALL_;
```

You may find it more convenient to view a subset of macro variables by category with these statements:

```
%PUT _AUTOMATIC_;
%PUT _GLOBAL_;
%PUT _LOCAL_;  *  Only makes sense within a macro  *;
%PUT _USER_;
%PUT _READONLY_;  *  New to SAS 9.4  *;
%PUT _WRITABLE_;  *  New to SAS 9.4  *;
```

6. ADD MACRO NAME TO %MEND

When ending a macro with %mend, add the name of the macro to the %mend statement, like this:

```
%macro Name_Of_Macro;
%*  Your macro contents here  ;
%mend Name_Of_Macro;
```

This becomes crucial when you have a program with many macros, for macros embedded inside macros, and for macros with more lines than will fit on one screen. Having the macro name on the %mend statement makes the code easier to read, and when chasing certain bugs such as a missing semi-colon or unterminated comment, the warnings about mismatched macro names between the %macro and %mend statements may give a hint about where the problem lies. This is a good habit that takes almost no effort to implement.

7. KEEP PASSWORDS (OR OTHER TEXT) OUT OF SIGHT WITH A MACRO

SAS can easily pull data from a variety of database systems. Typically, the database connection requires providing a username and password. The syntax typically looks something like this:

```
proc sql;
   connect to dbms as dbcon (user="user" password="1234");

   create table mytable as
   select ... ;
quit;
```

Unfortunately, now that I've shared my code with you, you have my username and secret password!

One alternative is to use macro variables to store your username and password, like this:

```
   connect to dbms as dbcon (user="&dbuser." password="&dbpass.");
```

That removes your username and password from the program. Now there's the matter of getting those macro variables set. There are many options such as setting them in your autoexec.sas file. Another option is to %Include an external file, such as this:

```
%include "~/password.sas" /NOSOURCE2;
```

Use the /NOSOURCE2 option to ensure the contents of the included file will not be written to the log. Hint: on a UNIX or Linux system, the tilde represents the user's home directory, allowing each person who runs the code to use their own individualized password file. Now the contents of password.sas look something like this:

```
%let dbuser=user;
%let dbpass=1234;
```

This will keep your username and password out of your code and the log, unless the SYMBOLGEN option is set. With SYMBOLGEN, you'll see this in the log:

```
SYMBOLGEN:  Macro variable DBUSER resolves to user
SYMBOLGEN:  Macro variable DBPASS resolves to 1234
```

To really sequester the username and password, we need to change the tool we are using. Instead of a macro variable, use a macro, like this:

```
%macro dbuser;user%mend dbuser;
%macro dbpass;1234%mend dbpass;
```

As unusual as it might seem, a macro can be used to store a string of characters just like a macro variable, with the added benefit that SYMBOLGEN won't accidentally spill the beans. Your database connection code should now look like this:

```
   connect to dbms as dbcon (user="%dbuser" password="%dbpass");
```

Even with the MLOGIC and MPRINT options set, your username and password will stay out of the log when the macro calls are enclosed in quotation marks. In this way, you can use a macro to prevent your password (or other secret text) from being accidentally revealed when you share your program code and logs. It should be noted that this is only useful to prevent the accidental revelation of these values as it is easy to reveal them intentionally using a %Put statement.

8. KEYWORD VS POSITIONAL PARAMETERS

When writing macros, use keyword parameters instead of positional parameters to make your code easier to read. Furthermore, with positional parameters, it is easy to accidentally swap the order of two variables when writing the macro call, and end up with a bug that is hard to catch. With keyword parameters, you can swap the order all you want as long as the right value is assigned to the correct variable name.

To illustrate this point, take a look at these two macro calls. Which macro call is easier to understand? Which method is easier to catch the bug I intentionally added?

```
%Payment (1234, 0.05, 36)
%Payment (loanamt=1234, months=0.05, rate=36)
```

Unless the parameters to a macro are obvious, I strongly suggest using keyword parameters instead of positional parameters.

9. WITHIN A MACRO, DECLARE SCOPE OF ALL NON-PARAMETER VARIABLES

When writing a macro, declare all non-parameter variables with either %Local or %Global. If you do not declare the variable scope, SAS will decide for you, but you will not want to do this.

For most variables created inside a macro, you will want to declare them as %Local. If you don’t do this, there is a possibility that you will have an identically named variable somewhere else in the program and your macro will have the unintended side-effect of modifying that variable. This is especially likely to happen if someone re-uses your macro code in another program. By declaring the variable as local, your macro gets its own private copy of that variable. On multiple occasions, I’ve spent hours debugging programs where the bug was traced back to two macros using the same temporary variable names.

When you do want to modify the value of a macro variable outside of the macro, declare it as %Global. This is absolutely necessary if the variable doesn’t already exist globally. Even if it does already exist, you should declare the variable as global for these additional reasons: 1) anyone reviewing your code will know that you didn’t accidentally forget to declare the variable as local, and 2) the global statement alerts the reader of your code that your macro may modify that variable so they should be aware of that.

All of the macro parameter variables will automatically be set to local, and it is not possible to change this with a %Global statement.

10. SAVE AND RESTORE SYSTEM OPTIONS

If you want to change any system options as part of your macro, the polite thing to do is save and restore the previous option values when the macro ends.

First, save the option’s original value using the GetOption function. Then when your macro’s code has finished, restore the original values using the OPTIONS statement. The macro below shows how to do this.

```
%macro demo_option;

   %local  OLD_OBS  OLD_SYMBOLGEN  OLD_YEARCUTOFF;

   %*  Save original option values.  *;
   %let OLD_OBS = %sysfunc (GetOption (OBS));
   %let OLD_SYMBOLGEN = %sysfunc (GetOption (SYMBOLGEN));
   %let OLD_YEARCUTOFF = %sysfunc (GetOption (YEARCUTOFF));

   %*  Change options temporarily.  *;
   OPTIONS OBS=5  NOSYMBOLGEN  YEARCUTOFF=1935;

%****  Replace this line with your macro code!!!  ****;

   %*  Restore the original option values. *;
   OPTIONS OBS=&OLD_OBS.  &OLD_SYMBOLGEN.  YEARCUTOFF=&OLD_YEARCUTOFF.;

%mend demo_option;
```

Be sure to declare the old option variables with %Local.

11. KEEP “ERROR” AND “WARNING” FALSE HITS OUT OF LOG

Your programs may contain some problem detection code that will write messages to the log letting you know there is a problem. This is a very good practice as it allows you to proactively search for problems. However if you search for the word “error” or “warning” in your log, you may get some false hits. For example, consider the code below that performs a data validity check.

```
*  Perform validity check - write to log if problem found  *;
data _null_;
   set sashelp.heart;

   if (not missing (AgeAtDeath)) and (AgeAtDeath < AgeAtStart) then do;
      error 'ERROR: AgeAtDeath is less than AgeAtStart';
      stop;
   end;
run;
```

This code will write an error message to the log if some bad data is found. However, even if no problems are found, when the code is echoed to the log, the word “error” will appear twice in the log – this makes it a little more time consuming to search through the log for errors. A sneaky workaround is to use the %UpCase function around the first couple of letters, so that the word error does not appear uninterrupted in the code, like this:

```
      %UpCase(er)ror "%UpCase(ER)ROR: AgeAtDeath is less than AgeAtStart";
```

Now the word “error” will only appear in the log when there’s an actual problem. You can do the same thing for “warning”, or any other word you want to keep out of your log until it is necessary.

12. WRITE FLEXIBLE MACROS WITH PARAMETER DEFAULTS

After you write a macro, you may realize you’d like to extend your original macro by adding another parameter. We can add a new parameter to a macro without breaking existing code that uses the macro by specifying a default value that matches the original macro’s behavior. When the pre-existing code uses the macro without the new parameter, it will still produce the same results as before.

As an example, let’s say we have a parameter-less macro named “Shuffle_Cards” that creates a data set named “Deck” containing a single deck of randomly shuffled playing cards. After creating this macro and putting it to use, we realize we’d like to specify the name of the output data set and to specify the number of decks of cards to be used. We’ll add a new parameter out= with a default value of “Deck”, and a new parameter NumDecks with a default value of 1.

Original macro:

```
%macro Shuffle_Cards;
data Deck;
   array suits  {4} $ _temporary_ ('H' 'D' 'S' 'C');
   array values {13} $ _temporary_
           ('A' '2' '3' '4' '5' '6' '7' '8' '9' '10' 'J' 'Q' 'K');

   drop s v;
   do s = LBound (suits) to HBound (suits);
      do v =  LBound (values) to HBound (values);
         Suit = suits {s};
         Value = values {v};
         Random = Rand ('UNIFORM');
         output;
      end;
   end;
run;
proc sort data=Deck;
   by Random;
run;
%mend Shuffle_Cards;
```

Extended macro:

```
%macro Shuffle_Cards (out=Deck, NumDecks=1);
data &out.;
   array suits  {4} $1 _temporary_ ('H' 'D' 'S' 'C');
   array values {13} $2 _temporary_
           ('A' '2' '3' '4' '5' '6' '7' '8' '9' '10' 'J' 'Q' 'K');

   drop s v i;
   do s = LBound (suits) to HBound (suits);
      do v =  LBound (values) to HBound (values);
         Suit = suits {s};
         Value = values {v};
         do i = 1 to &NumDecks.;
            Random = Rand ('UNIFORM');
            output;
         end;
      end;
   end;
run;
proc sort data=&out.;
   by Random;
run;
%mend Shuffle_Cards;
```

We can call the extended macro without parameters and get the same result as before:

```
%Shuffle_Cards
```

We can also call the macro with any permutation of optional parameters. The omitted parameters will be set to the default values.

```
%Shuffle_Cards (out=My_New_Deck)

%Shuffle_Cards (NumDecks=8)

%Shuffle_Cards (out=My_New_Deck, NumDecks=8)
```

13. PULLING DATA FROM CHRONOLOGICAL DATA SETS

In cases where a lot of data is collected, it is common practice to split the data into data sets organized by a span of time, such as one data set per year. This can speed up the process when you only need data from a limited range of dates, but also provides a coding challenge.

ANNUAL DATA SETS

Consider the case where you need to efficiently gather 18 months of data from very large annual data sets. Depending on what time span you are considering, you will need to open either two or three years of data – see the illustration below. We don't want to waste time and resources opening data sets that we don't use, so what is the best way to handle this situation?

2013	J	F	M	A	M	J	J	A	S	**O**	**N**	**D**
2014	**J**	**F**	**M**	**A**	**M**	**J**	**J**	**A**	**S**	**O**	**N**	**D**
2015	**J**	**F**	**M**	A	M	J	J	A	S	O	N	D

2013	J	F	M	A	M	J	J	A	S	O	N	D
2014	J	F	M	**A**	**M**	**J**	**J**	**A**	**S**	**O**	**N**	**D**
2015	**J**	**F**	**M**	**A**	**M**	**J**	**J**	**A**	**S**	O	N	D

This is a problem that is almost unsolvable without a macro or macro variable. What we can do is write some code that determines the first and last years required, and then uses a Do loop to build a list of data set names. Then we will use Call SymputX to create a macro variable containing a list of only the data sets required to solve the problem.

Here’s an example of a data step that will build a macro variable containing a list of annual data set names:

```
data _null_;
   start_year = year (&First_Day_of_Report.);
   end_year = year (&Last_Day_of_Report.);
   length list_of_datasets $250;

   do i = start_year to end_year;
      list_of_datasets = trim (list_of_datasets)
                         || " annual_data_" || left (i);
   end;

   call symputx ('list_of_datasets', list_of_datasets);
run;
```

Now we can use that macro variable in another data step to collect the data required, like this:

```
data use_annual_data;
   set &list_of_datasets.;
   where data_date between &First_Day_of_Report.
                   and &Last_Day_of_Report.;

***   Add additional processing code here  ***;
run;
```

As an example, if you run this code with the dates 01OCT2013 and 31MAR2015 (an 18 month span), it will produce the following list of data sets:

```
annual_data_2013
annual_data_2014
annual_data_2015
```

MONTHLY DATA SETS

If the data sets are broken up by month, with names ending in YYYYMM, this is more challenging due to gaps in numerical sequence between years (e.g. 201212 to 201301).

Here’s an example of a method to build a macro variable containing a list of data sets using a Do While loop, and has logic to skip from December to January of the next year.

```
data _null_;
   start_year = year (&First_Day_of_Report.);
   end_year = year (&Last_Day_of_Report.);
   start_month = month (&First_Day_of_Report.);
   end_month = month (&Last_Day_of_Report.);
   length list_of_datasets $1000;

   month = start_month;
   year = start_year;

   do while (((year*100)+month) <= ((end_year*100)+end_month));
      list_of_datasets = trim (list_of_datasets)
                         || " monthly_data_" || left (((year*100)+month));
      month = month + 1;
      if (month EQ 13) then do;
         month = 1;
         year = year + 1;
      end;
   end;

   call symputx ('list_of_datasets', list_of_datasets);
run;
```

The code to use the macro variable looks like this:

```
data use_monthly_data;
   set &list_of_datasets.;
   where data_date between &First_Day_of_Report.
                       and &Last_Day_of_Report.;

***   Add additional processing code here  ***;
run;
```

As an example, if you run this code with the dates 17NOV2014 and 14FEB2015 (a 90 day span), it will produce the following list of data sets:

```
monthly_data_201411
monthly_data_201412
monthly_data_201501
monthly_data_201502
```

Using these techniques will allow your program to dynamically determine which data sets are required.

CONCLUSION

The SAS Macro Facility greatly extends the functionality of Base SAS to allow you to automate manual tasks, improve your coding efficiency, and even solve problems that would be unsolvable without Macros and Macro Variables. This paper provides a variety of ideas for putting the SAS Macro Facility to use. Hopefully these techniques will help you to create better programs.

REFERENCES

- SAS Institute Inc. “INTNX Function”. *SAS® 9.3 Functions and CALL Routines: Reference.* http://support.sas.com/documentation/cdl/en/lefunctionsref/63354/HTML/default/viewer.htm#p10v3sa3i4kfxfn1sovhi5xzxh8n.htm
- SAS Institute Inc. "Local Macro Variables". *SAS® 9.3 Macro Language: Reference.* http://support.sas.com/documentation/cdl/en/mcrolref/62978/HTML/default/viewer.htm#p0lo5it6f5i3adn1uldh8qgd2kjx.htm
- SAS Institute Inc. “%MACRO Statement”. *SAS® 9.3 Macro Language: Reference.* http://support.sas.com/documentation/cdl/en/mcrolref/62978/HTML/default/viewer.htm#p1nypovnwon4uyn159rst8pgzqrl.htm
- SAS Institute Inc. “%PUT Statement”. *SAS® 9.3 Macro Language: Reference.* http://support.sas.com/documentation/cdl/en/mcrolref/62978/HTML/default/viewer.htm#n189qvy83pmkt6n1bq2mmwtyb4oe.htm
- SAS Institute Inc. “%PUT Statement”. *SAS® 9.4 Macro Language: Reference, Fourth Edition.* http://support.sas.com/documentation/cdl/en/mcrolref/67912/HTML/default/viewer.htm#n189qvy83pmkt6n1bq2mmwtyb4oe.htm
- SAS Institute Inc. “%SYSFUNC and %QSYSFUNC Functions”. *SAS® 9.3 Macro Language: Reference.* http://support.sas.com/documentation/cdl/en/mcrolref/62978/HTML/default/viewer.htm#p1o13d7wb2zfcnn19s5ssl2zdxvi.htm

ACKNOWLEDGMENTS

Dedicated to Mildred, Jimmie, Ann, Chansouk, and Sophie.

Thank you to Jason Xue and Michelle Berger for proofreading assistance.

RECOMMENDED READING

- Carpenter, Arthur L. (2005). “Five Ways to Create Macro Variables: A Short Introduction to the Macro Language”. SESUG 2005. Paper HW03_05. http://analytics.ncsu.edu/sesug/2005/HW03_05.PDF

- Delaney, Kevin P. and Arthur L. Carpenter (2004). “SAS® Macro: Symbols of Frustration? %Let us help! A Guide to Debugging Macros”. SUGI 29. Paper 128-29. http://www2.sas.com/proceedings/sugi29/128-29.pdf
- Russell, Kevin and Russ Tyndall (2010). “SAS® System Options: The True Heroes of Macro Debugging”. SAS Global Forum 2010. Paper 147-2010. http://support.sas.com/resources/papers/proceedings10/147-2010.pdf
- SAS Institute Inc. (2015). SAS® 9.4 Macro Language: Reference, Fourth Edition. SAS Institute, Cary, NC. http://support.sas.com/documentation/cdl/en/mcrolref/67912/PDF/default/mcrolref.pdf
- Slaughter, Susan J. and Lora D. Delwiche (2011). “SAS® Macro Programming for Beginners”. SAS Global Forum 2011. Paper 258-2011. http://support.sas.com/resources/papers/proceedings11/258-2011.pdf

CONTACT INFORMATION

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

Name: Scott A. Miller
SMS: (515) 537-4288
E-mail: smiller933 @ gmail.com