Paper SBC-121

Techniques for Developing Quality SAS ® Macros

Ginger Redner, Merck Research Laboratories, North Wales, PA
Liping Zhang, Merck Research Laboratories, North Wales, PA
Carl Herremans, MSD Europe, Inc, Belgium, Brussels

ABSTRACT

Defining specific guidelines and standards for designing and developing library macros can significantly increase the quality of those macros. This paper details some of the macro design and development guidelines we've defined in our organization for developing quality, easy to use, flexible macros that have minimal impact on the SAS session in which they are executed. Sample code to assist in adhering to these guidelines will also be discussed. This paper is intended for programmers with a sound foundation in SAS macro programming.

INTRODUCTION

Assuring that SAS macros developed for a common macro library are user-friendly, robust and maintainable can be a challenge. Most SAS macro developers put a considerable amount of effort into building macros that are efficient, and efficiency is an important consideration, but there are other simple techniques that can greatly enhance the quality of a SAS macro. This paper details some important macro design and development guidelines which make macros easier to use, easier to maintain, more flexible, minimize impact on the SAS session in which the macros are executed, and increase the overall quality of our macros.

In our organization, we've included the sample code described in this paper in a macro template. This macro template is a valuable tool for assisting macro developers in adhering to the guidelines and reducing macro development time.

EASE OF USE

An important criterion that can be used to evaluate the quality of a macro is to determine how easy it is to use. Here are a few suggestions that help to assure that a macro is user-friendly.

MESSAGE INDICATING START AND END OF MACRO

For debugging purposes, it is very helpful for users to be able to determine the beginning and end of a particular macro. Sending a note to the SAS log is an easy way to convey this information.

```
 %* include start of macro code.
 %put --- Start of %upcase(&sysmacroname) macro;

%* include end of macro code.
%put --- End of %upcase(&sysmacroname) macro;
```

DISPLAY CODE FOR STATISTICAL PROCEDURES IN LOG

Statisticians often want to review the exact statistical code that is executed when a macro is used. Setting the SAS option MPRINT on right before executing statistical procedures sends all the statements of the statistical procedure to the log. Turning the MPRINT option on can be useful for displaying the code for any procedure or data step in the log. Restoring the original value of MPRINT after the statistical procedure, assists in maintaining a clean and readable SAS log.

```
%* backup mprint setting;
%let mprintSetting = %lowcase(%sysfunc(getoption(mprint));
options mprint;
```

```
proc mixed data=...;
   ...
run;

%* restore original mprint setting;
options &mprintSetting;
```

ASSIGN DEFAULTS TO PARAMETERS WHEN POSSIBLE

If a particular parameter value is commonly used, it's good practice to assign the parameter a default value, so that users do not need to specify the parameter in the call program.

```
libname datadir “c:\analysis-data\”;

%macro a(inputlib =,inputdataset =, outputlib= datadir,outputdatasetds = );
   ...
%mend a;

%a(inputlib= xxx, inputdatasetds = yyy, outputds= zzz);
```

USE COMMON PARAMETER NAMES

When parameter names are intuitive and follow a standard naming convention it's easier for users to figure out which names are used when calling the macro. For example, INPUTDATASET intuitively refers to an input data set name and OUTPUTDATASET refers to an output data set name. Using these same parameter names across all macros in a common library makes it easier for the user to setup the macro calls.

PROVIDE ERROR CHECKS

Most of us have input incorrect values into macro parameters at one point or another. It's frustrating to then run the macro, encounter a problem, and be left to wade through pages of SAS log to determine which parameter value caused the problem. One technique to help alleviate this problem is to include bulletproof error and warning checks on parameter values. The programming code for error checking is simple, and repetitive code can be used for each error check performed. Though additional test cases are needed to force each potential error or warning, the extra time spent validating is made up in time saved when the macros are used.

Error checks should be added to check for required input parameter values that are missing, invalid parameter values, and invalid combinations of parameter values. Additional checks should be added to check for the existence of any input data sets, formats, filenames and variables passed into the macro. Error messages should provide the user with sufficient information in order to fix the parameter values. A few examples follow:

- <u>Check if required input parameter values (&INPUTDATASET) are valued:</u>

```
%local err or;
%let err=ERR;
%let or=OR;
%if %length(&inputdataset)=0  %then %do;
   %put &err&or: (&sysmacroname) Parameters INPUTDATASET can not be empty.;
   %goto exit;
%end;
```

- <u>Check if input parameter values (&DEBUG) are valid :</u>

```
%if "&debug" ^= "Y" and "&debug" ^= "N" %then %do;
    %put &err&or: (&sysmacroname) Invalid value for the macro parameter DEBUG.;
    %put &err&or: (&sysmacroname) Value should be 'N' or 'Y'.;
    %goto exit;
%end;
```

- Check if input data set parameter (&INPUTDATASET) contains an existing data set:

```
%if %sysfunc(exist(%scan(&inputdataset,1,%str(%()))) ne 1 %then %do;
    %put &err&or: (&sysmacroname) The %upcase(%scan(&inputdataset,1,%str(%()))
data set
    %put &err&or: (&sysmacroname) (INPUTDATASET parameter) does not exist.";
    %goto exit;
%end;
```

- Check the existence of temporary formats (_XAXIS, _YAXIS):

```
%local warn ing;
%let warn=WARN;
%let ing=ING;
%let formatclist = _xaxis _yaxis;
%let frmt = %str();
%let i = 1;
%do %while(%length(%scan(&formatclist,&i,%str( ))));
   %let frmt=%scan(&formatclist,&i,%str( ));
        %let format_exist = 0;
        proc sql noprint;
          select count(*) into: format_exist
           from sashelp.vcatalg
           where libname='WORK'
                 and objtype='FORMATC'
                 and memname='FORMATS'
                 and upcase(objname)=upcase("&frmt");
        quit;
        %if &format_exist ne 0 %then %do;
        %put &warn&ing: (&sysmacroname) The format %sysfunc(upcase(&frmt));
        %put &warn&ing: (&sysmacroname) already exist and will be replaced;
        %put &warn&ing: (&sysmacroname) by the macro.";
     %end;
     %let i=%eval(&i+1);
%end;
```

MAINTAINABILITY

A macro can only be used for as long as it can be maintained and enhanced.   The most basic rule for building maintainable macros is to "keep it simple". The following guidelines help to develop maintainable macros.

MODULAR DESIGN

Macros built in a modular fashion, having a distinct function are much easier to work with than large, complex macros that perform a plethora of processing.   There is often a tendency for programmers to develop single macros with the functionality to meet all user requirements for a task. For example, a typical program to produce output tables and graphs for submission to the FDA includes code for data manipulation, statistical analysis, table generation and graph generation. Including all of this code in one program is fine when the program is only expected to be run once of twice. When building macros to repeatedly perform this same processing, designing four separate and distinct macros, one for data manipulation, one for statistical analysis, one for table generation and one for graph generation, follow the "keep it simple" rule and greatly reduces the complexity of a single macro. If one piece of the process is changed, such as the output table layout, only one macro needs to be updated and retested rather than opening up one large macro that could contain interdependencies between the program generation section and other pieces of the program.

Developing smaller functional macros also enables each of these separate macros to be used in a stand-alone fashion or in conjunction with other macros, such as a macro that performs a different type of statistical analysis.   It may be helpful to stack the functional macro calls and build an additional macro around the stacked macro calls, allowing one macro call to perform the complete stream of processing.

INCLUDE DEBUG PARAMETER

Even when following the “keep it simple” rule, there will be times when larger, more complex macros are needed. If a problem is found when running a larger macro, it can be very hard to determine if the problem is a data problem, user error or a problem with the macro code. Adding a debug parameter and including code to assist in problem solving makes investigating problems with the macro much easier, and eliminates the need to manipulate the program code in order to debug. Adding code to execute when the debug parameter is on to retain temporary data sets, print a subset of intermediary data sets, print macro variable values in the SAS log, and send messages to aide in the debugging process to the log, is most helpful. Though more code is developed, the steps for debugging are usually the same steps the developer needs in order to perform normal developer testing. With this technique, the testing code is kept and available for future testing and debugging.

DEFINING SUB-MACROS

Macros defined within main macros are compiled each time the main macro is executed; therefore it’s more efficient to define larger sub-macros as separate macros. Separate macros are only compiled when they are included into a SAS program, or when the macro is used for the first time within a SAS session.

When developing larger macros we often have the need to build many smaller macros specifically designed to be used only with the larger main macro. In this case, it may be better to include the definition of these smaller macros within the main macro to eliminate the need to maintain what could become a very large library of smaller sub-macros. Controlling the names of these independent sub-macros could become tricky.

All macros designed to be called by more than one calling macro should be defined as separate macros and stored in a common library. Defining a macro with similar functionality from within multiple calling macros can lead to duplication of effort if the functionality needs to change.

COMMENT END OF %IF AND %DO BLOCKS

Often including large %do or %if blocks in a macro cannot be avoided. Matching the beginning and ending of these blocks can become quite difficult, especially when they are nested. When more than 10 lines of code exists within a %do or %end block adding a comment to the %end statement greatly enhances the readability of the program.

```
%do _i = 1 %to 100;
  ….
%end;  %*** do _i = 1 %to 100;
```

FLEXIBILITY

The more flexible a macro is, the more likely it will be that the macro can deal with changing context. The fewer assumptions needed for proper functioning of the macro, the better. Macros designed with flexibility in mind require little or no updates in order to deal with a change in context. A typical example of a change in context is the transition from a company specific database structure to the SDTM database structure. The following basic set of guidelines reflects key factors in flexible macro design.

SPECIFY ALL RESOURCES USED AS MACRO PARAMETERS

All resources used by the macro (ex. macro variables, data sets, libnames, formats, titles, footnotes, headers, etc.) should be defined as input/output macro parameters.

```
%macro noflex1(where=);
   data subae;
     set ae;
     where &where;
   run;
%mend noflex1;

%macro flex1(inputdataset=ae, where=, outputdataset=subae);
   data &outputdataset;
     set &inputdataset;
     where &where;
```

```
      run;
   %mend flex1;
```

The above macro, NOFLEX1, is a typical example of a macro that is functioning properly with respect to its requirements. However when the context changes, for example change in naming convention of data set AE, the macro won't be able to do its job. The macro FLEX1 illustrates a more flexible design of the macro while providing the same functionality. This macro can easily deal with the change in context simply by changing the value of the macro parameter &INPUTDATASET.

ALLOW FOR DEFINITION OF BY-VARIABLES

Another typical change in context is the requirement to produce separate tables or graphs for different subgroups of data. Flexible macros allow similar processing to be done for different by groups. Typical examples of using by groups are processing by region, center and/or time-point. It's also good practice to allow multiple variable names to be passed in appropriate parameters, such as parameters defining by-variables and class variables.

MACROS SHOULD BE DATA DRIVEN

Hard-coded information should be avoided when it can be extracted from the data. Typical examples are dynamically defining array sizes and determining the type of a variable.

▪ Dynamically determine if a variable is character or numeric

```
%let dsid=%sysfunc(open(&inputdataset));
%let vartype=%sysfunc(vartype(&dsid, %sysfunc(varnum(&dsid,trt))));
%let rc=%sysfunc(close(&dsid));
```

• Dynamically define the size of an array

  Transposing a data set is a typical data manipulation technique that often results in an unknown number of new variables. The code below determines the number of variables created by transposing a data set, stores the value in the macro variable ARRAYLENGTH, and then uses this macro variable to define the dimensions of the _TRT_ array.

```
proc transpose data=&inputdataset out=tempdataset prefix=trt;
   var treatment;
run;

%let dsid=%sysfunc(open(tempdataset));
%let i=1;
%do %while(%sysfunc(varnum(&dsid,trt%left(&i))) ne 0 );
   %let i=%eval(&i+1);
%end;
%let arraylength=%eval(&i-1);
%let rc=%sysfunc(close(&dsid));

data _null_;
   set tempdataset;
   array _trt_ {*} trt1-trt%left(&arraylength);
   do i = 1 to dim(_trt_);
      trt=_trt_(i);
   end;
run;
```

  In the example below the names of the array variables are data driven due to the use of the ID statement in the TRANSPOSE procedure. This code dynamically constructs the list of variables that compose the array.

```
proc sql noprint;
  select distinct compress('TRT'||put(trt,3.))
     into :array separated by ' '
        from &inputdataset;
quit;
```

```
proc transpose data=&inputdataset out=tempdataset prefix=trt;
   var trt;
   id trt;
run;

data _null_;
   set tempdataset;
   array _trt_ {*} &array;
   do i = 1 to dim(_trt_);
      trtcode=_trt_(i);
   end;
run;
```

ALLOW SAS DATA SET OPTIONS ON INPUT AND OUTPUT DATA SET NAMES

Allowing data set options on the input/output data sets passed into a macro eliminates the need for unnecessary DATA steps prior to the macro call. Typical examples are:

- Applying a where clause to a data set

```
%m_where(inputdataset=demog(where=(gender='M')) ...
```

- Renaming variables used in the input data set to comply with required variable names used in the macro

```
%m_where(inputdataset=demog (rename=(sex=gender)) ...
```

- Compressing an output data set to save disk space

```
%m_where(outputdataset=effic (compress=YES) ...
```

REDUCE IMPACT ON THE SAS SESSION

It is important that a macro tread lightly on the SAS session in which it is run. Care should be taken to reduce the possibility of overwriting elements that may be needed by the calling program, as well as restoring the SAS environment to the state it was in before the macro was executed.

RESET SAS OPTIONS, TITLES AND FOOTNOTES

SAS options, titles and footnotes should only be changed by a macro if necessary for proper macro function. Certain options, such as NOSYMBOLGEN and NOMLOGIC, should only be turned on within a macro is in debug mode, as the output produced from these options greatly increases the size of the SAS log, making it difficult to read. Should any SAS options, titles or footnotes need to be changed within the macro it's important that these values be reset back to the original values, before exiting the macro.

- Reset SAS Options

  In the beginning of the macro the values of MAUTOLOCDISPLAY and MISSING are loaded into macro variables:

```
%let old_mautolocdisplay=%sysfunc(getoption(mautolocdisplay));
%let old_missing         =%sysfunc(getoption(missing));
```

  The options are then changed, as required by the macro:

```
options mautolocdisplay missing=" ";
```

At the very end of the program the options are reset back to the original values:

```
options &old_mautolocdisplay missing="&old_missing";
```

When a macro alters many SAS options, an alternative approach is to use the OPTSAVE and OPTLOAD procedures. In the following example the OPTSAVE procedure writes the values of all the SAS options that can be altered from within a SAS session to a SAS data set called _myoptions. The OPTLOAD procedure restores the SAS session option values from the _myoptions data set.

Before changing any option settings:

```
proc optsave out=_myoptions;
run;
```

Before exiting the macro reset all the SAS options back to the original values:

```
proc optload data=_myoptions;
run;
```

- <u>Reset SAS Titles and Footnotes</u>

  In the beginning of the program a backup data set is created containing one observation for each defined title:

```
proc sql noprint;
   create table _title_ as
      select type, number, text
      from sashelp.vtitle
      where upcase(type)="T";
quit;
```

  Before exiting the program the titles defined by the macro are cleared, the data set containing the backed up titles is read, and call execute is used to declare the original titles:

```
title;
data _null_;
  set _title_;
  call execute('title'||strip(number)|| ' ' || '"' || strip(text))||'";');
run;
```

  The same steps can be used to backup and restore values of footnotes simply by changing the where clause in the previous SQL step to select records with the value “F” instead of “T” `(where upcase(type)="F").`

PROTECT MACRO VARIABLES, FORMATS, TEMPORARY DATA SETS AND SUB-MACROS

A calling program may have macro variables, temporary data sets, formats or sub-macros defined that may be needed after execution of the called macro. The following guidelines help in protecting these entities during execution of a macro.

- <u>Declare macro variables in local symbol table</u>

  The scope of macro variables is extremely important to program functionality. Defining all possible macro variables created by a macro as local variables is the safest way of protecting macro variables defined in the calling program from being altered. Macro variables can be defined to the local symbol table using either the %LOCAL statement, or, in SAS version 9, the CALL SYMPUTX function with the “L” symbol table option.

- <u>Use naming conventions when creating global macro variables, temporary data sets, formats or sub-macros</u>

  If any macro variables are needed after macro execution, or temporary data sets, formats or sub-macros are created, adhering to simple naming conventions can significantly reduce the possibility of overwriting these items. One common convention is for macro developers to prefix and/or suffix the name of these entities with an underscore or double underscore. To protect previously created items, programmers writing calling programs are recommended against using these same naming conventions.

```
%global _an __nan startdt_ enddt__ _npats_;
```

- Document Global Variables and Permanent Data sets in Program Header

  Including sections within your standard macro program header to document the global macro variables and any permanent data sets created by a macro, is an easy way to provide the user with a standard place to verify there is no conflict between global macro variables and permanent data sets created by the calling program and the macro.

- Check if Global Macro Variables, Data sets, Formats and Sub-macros Exist

  Debugging a program where a macro overwrites a macro variable value, data set, format or sub-macro that the calling program has already defined can be quite difficult and frustrating. The only way to be completely sure a macro is not overwriting already existing entities is to check for the existence of each entity being created by a macro, issue an error if the item already exists and stop processing of the macro. In our organization we've decided issuing a warning to the SAS log when one of these items to be created already exists, is more appropriate for our applications.

  Check if Macro Variables, SAS Data Sets and SAS Formats Exist

  This sample code checks to see if the macro variable _ESTIM exists in the global symbol table, or the local symbol tables of previously called macros, and issues a warning if needed:

```
Proc sql noprint;
  select count(*) into :_exists
    from sashelp.vmacro
    where name='_ESTIM' and name not = upcase("&sysmacroname");
quit;
%if &_exists >= 1 %then %do;
   %put &warn&ing: (&sysmacroname) Macro Variable _ESTIM exists and has;
   %put &warn&ing: (&sysmacroname) been overwritten by the macro;
%end;
```

  New functions included with SAS Version 9, %SYMEXIST, %SYMLOCAL and %SYMGLOBAL make these checks a bit easier. The code below checks to see if the macro variable _ESTIM exists in either the global or local symbol tables:

```
%if %symexist(_estim) %then %do;
    %put &warn&ing: (&sysmacroname) Global Macro Variable _ESTIM exists and has;
    %put &warn&ing: (&sysmacroname) been overwritten by the macro.;
%end;
```

  Code similar to that shown on page 2, section "Provide Error Checks", can be used to check for the existence of data sets and formats entered in parameter values.

  Check if a SAS Macro Exists

  This example checks if the SAS macro _CHECKIT already exists:

```
%local macro exist;
proc sql noprint;
   select count(*) into :macro_exist
     from sashelp.vcatalg
     where libname='WORK'
        and objtype='MACRO'
             and memname='SASMACR'
             and upcase(objname)='_CHECKIT';
quit;
%if &macro_exist ne 0 %then
    %put &warn&ing: (&sysmacroname) The format _XASIS already exists and will be;
    %put &warn&ing: (&sysmacroname) replaced by the macro.;
%end;
```

CLEANUP

In order to leave the user of a macro with a clean environment, it's important to delete all unneeded global macro variables, temporary data sets, formats and sub-macros created by a macro prior to exiting the macro.

- Cleanup Unneeded Global Macro Variables

  When in debug mode, the ability to review the values of macro variables after a macro has run is quite helpful. When not in debug mode, deleting all global macro variables that are not needed by external programs is recommended in order to leave the user with a clean global symbol table.

  When the following code is included in the very beginning of a macro, before any global macros are declared, the names of the global macro variables defined by the calling programs are put into a local macro variable called SAVEGLOBAL:

```
%local saveglobal;
proc sql noprint;
   select "'" !! strip(name)!! "'" into :saveglobal
                  separated ","
     from sashelp.vmacro
     where scope="GLOBAL";
quit;
```

  In the end of the macro all the macro variables in the global symbol table that were not defined by the calling programs (not included in macro variable &saveglobal) are deleted:

```
%if &debug = N %then %do;
    data temp;
      set sashelp.vmacro;
     where scope="GLOBAL" and name not in (&saveglobal);
    run;
    data _null_;
      set temp;
      call execute('%symdel ' || name || '/nowarn');
    run;
%end;
```

- Delete Temporary Data Sets, Formats and Sub-macros

  Deleting all data sets, formats and sub-macros defined by the macro takes a small amount of program code and very little time, but can save considerable work space and memory, as well as reduce the chance for entity naming conflicts. In addition the users of your macros will appreciate being left with a clean environment.

  This example deletes three data sets from the temporary work library:

```
%if &debug = N %then %do;
    proc datasets lib=work memtype=data nolist nowarn;
       delete _ds1 _ds2 _ds3;
    quit;
%end;
```

  This sample code deletes two character formats from the temporary formats catalog. The same code can be used to delete numeric formats by using the entry type format (ET=format).

```
proc catalog cat=work.formats;
   delete _agecat _gender /et=formatc;
quit;
```

This example deletes two sub-macros from the temporary macro catalog.

```
proc catalog cat=sasmacr;
   delete _submac1 _submac2 /et=macro ;
quit;
```

OVERALL QUALITY

Following are a few additional techniques that can enhance the quality of macros stored in a common library. .

- Use standard commenting blocks throughout program and use proper macro commenting (%*   ; instead of /*   */ or *;).
- When evaluating equality/inequality conditions for macro variables, either double quotes should surround the parameters and comparator values, or the %quote or %nrbquote function should be used with the macro variables. These functions mask the effect of special characters, such as the dash ('-') which would be interpreted as a minus sign in the condition, as well as SAS reserved words.
  Example:
  Instead of: %if &tblspprs eq N and &DocLibr = %then
  Use: %if "&tblspprs" eq N and "&DocLibr" = "" %then   or
                %if %nrbquote(&tablspprs) eq "N" and %nrbquote(&doclibr) = "" %then
- Use keyword parameters instead of positional parameters.
- The %mend statement should include the name of the macro.
- When using the words ERROR, WARNING or UNINITIALIZED in the macro header or comments, replace the character "O" by the number 0 and replace the character "I" by the number 1 in these words (ERR0R, WARN1NG, UN1NITIALIZED).
- Designing a macro template that models a standard macro layout, includes detailed commenting, as well as standard programming code for routine functions will help maintain consistency in design of all the macros in the common library.

CONCLUSION

There are simple techniques that can be applied to common-library macros to enhance quality.   Adding sample code for these techniques in a sample template program will assist developers in using the techniques and decrease macro development time.

REFERENCES

Bramley, Michael (2004), "Better Clay Builds Better Bricks: Some Simple Suggestions to Writing Professional Macros", *Proceedings of the Twenty-Ninth Annual SAS Users Group International Conference,* Paper 055-29.

DiTommaso, Dante & Szilagyi, Benjamin (2005), *Proceedings of the 2005 Pharmaceutical Users Software Exchange Conference,* Paper TU-08

Morris, Amy (2001), "Can standards be implemented effectively without compromising science?", *Proceedings of the 2001 Pharmaceutical SAS User's Group Conference*, Paper SP-14.

Troxel, John K., (2002), "Bulletproofing and Knowledge Encapsulation in Statistical Macros", *Proceedings of the 2002 Pharmaceutical SAS User's Group Conference*, Paper TT-12

Roland Rashleigh-Berry (2003), "Tips on Writing SAS Macros", www.datasavantconsulting.com/roland/macrotips.html.

ACKNOWLEDGMENTS

The authors would like to thank all the members of Global Macro Strategic Initiatives team who assisted with the development of these guidelines, as well as Allan Glaser and Richard Lowry who provided helpful input.

CONTACT INFORMATION

Your comments and questions are valued and encouraged. Feel free to contact the authors at:

Ginger Redner	**Liping Zhang**	**Carl Herremans**
Merck & Co., Inc.	Merck & Co., Inc.	MSD Europe, Inc.
UG1CD-44	UG1CD-38	Clos du Lynx 5
PO Box 1000	PO Box 1000	Antares Building A + C
North Wales, PA 19454	North Wales, PA 19454	Belgium, Brussels B-1200
Work Phone: (267) 305-7634	Work Phone: (267) 305-7980	Work Phone: +(32 2) 766 6309
Email: Virginia_Redner@merck.com	Email: Liping_Zhang@Merck.com	Email: Carl_Herremans@merck.com