Paper 50-2010

Best Practices and Advanced Tips and Techniques for SAS Macro Programming

Gerry Bonin, We Energies, Milwaukee, WI

Abstract

SAS macros offer a powerful way to extend the SAS language and create reusable SAS code that is a custom fit for you. This paper shows you how to write macros for a production environment and includes some advanced tips and techniques. It includes templates for starting a new macro and for documenting macros, as well as some of the author's favorite coding practices.

Introduction

Many years of SAS coding experience have lead me to develop SAS macros and code that, at least to me, are faster and result in better code. This experience started in 1982 on the IBM mainframe and then in the mid-1990s extended to the Unix platform.
SAS macros are a very flexible and powerful tool in code development and can be considered as a type of reusable code, which results in improved code development time. Written and documented correctly, SAS code can achieve "industrial strength" required for production environments supported by multiple SAS developers and users.

Types of Macros

I tend to think of SAS macros as being one of four types:

Standalone

Utility

Part of a System of Macros

Objects

Standalone Macros

Standalone macro have little or no dependencies on other macros and could often be replaced by in-line SAS code (or a SAS code file that is %INCLUDE'd), except for the need/desire to use some macro features (likes loops) that are much easier to do in a macro.

Utility Macros

A utility macro is often a very small, single function macro, used to encapsulate SAS code that is usually small, especially if its "tricky" or used frequently or in many programs. An example would be to perform a special calculation on a variable, where the call would look something like:

```
result = %my_calc(var1= size, var2= duration) ;
```

Systems of Macros

Complete systems of macros can be created, where the interrelated set of macros get called within each other, often using global macro variables to store information created in the outer-level macros and used by the inner macros. The top-level macro will set up the processing environment subsequently used by the lower-level macros. Values can often be passed to the inner macros via parameters in the macro calls.

A variation of this is seen where the global macro variable environment is set up by a utility macro and that environment is then used by non-macro code.

Object Macros

"Object" type macros are used to create, maintain and report about a single type of data, similar to a real object created in an object oriented language like C++.

In my work with computer performance data, an object might something like servers, routers or databases, where each row in the table would represent one set of dozens of performance metrics (like CPU usage, I/O rates, etc.) for a given server and measurement interval. The table might contain thousands of rows for hundreds of servers.

One of the parameters on the macro would be to tell the macro what function to do (like "load data", summarize, merge, report or graph). Other parameters would be used to give the input file name, output table name, etc.

The advantage of an "object" macro is that it contains all the attributes for all the object's variables in one place, along with the code to maintain them correctly when being summarized, etc.

Industrial Strength Coding Practices

We have a shared SAS environment for my work group, with one set of production SAS databases and one set of production SAS code. Everyone in the work group uses code developed by the others.

We desired a consistent look and feel for the SAS code being developed and needed faster program development cycles.

In addition to the on-line code development, we also have nightly production batch jobs (run under central operator control using Unicenter AutoSys Job Management from Computer Associates).

Note: The SAS server is shared with many other SAS users that are not in my department.

One of the easiest ways to achieve consistent look and feel and fast code development is with macros and template code.

Template Macro

Templates are used for many things: SAS macros, SAS formats and informats and standard SAS code. We rarely start with a "blank screen".

When beginning to write a macro, we always start with the same macro template. The template contains some "starter" code, some of which we keep and modify for the task at hand. The parts that are not needed are either deleted (if we don't think it would ever be needed) or simply commented out (for possible use at a later time).

It's easier to delete unwanted code than to enter it (via typing or cut and paste).

Often we need to do something similar to what we've done before in another macro. In that case, we will "steal" it with copy and paste or INCLUDE if it's really big.

The template macro is comprised of several sections, described below.

Template Macro: Pre-Macro Definition Setup

When developing the macro or other SAS code, I like to clear the SAS log at the start of each development run, so I can quickly review the SAS log, so we have this line at the start of the template:

```
dm log 'clear' ;
```

Next, we turn on the option to display macro compiler notes, which can be very handy:

```
options mcompilenote= all /* Display macro compile information */ ;
```

Template Macro: Macro Prototype Statement

The prototype statement is used to name the macro and define its parameters.

```
%macro xxxxxxxx
     ( p1               /* 1st positional parm                          */
     , p2               /* 2nd positional parm                          */
     , k1= default      /* 1st keyword parm                             */
     , k2= default      /* 2nd keyword parm                             */
     , in=              /* input file name                              */
     , out=             /* output file name                             */
     , protect=         /* output file password                         */
     , where=           /* limit on the input file                      */
     , select=          /* limit during processing                      */
                        /* -------------------------------------------- */
     , mail_to=         /* E-mail addresses to mail the report to       */
                        /* Defaults to usersID@we-energies.com for      */
                        /* testing                                      */
                        /* -------------------------------------------- */
     , mail_cc=         /* E-mail addresses to carbon copy the report   */
                        /* -------------------------------------------- */
     , debug= 0   )     /* Debugging level (0=none, 3=max)              */
     /
/* store                /* Save compiled macro into SASMSTORE lib       */
/* source               /* Save the source with the compiled macro      */
                        /* Retrieve with %copy <macroname> / source;    */
/* secure               /* SAS 9.2 Encrypt the stored compiled macro    */
/* mindelimiter= ','    /* SAS 9.2 delimiter for the IN operator        */
;
```

Each parameter on the macro prototype statement is on a separate line, plus the inclusion of comments to describe the parameters and their acceptable values is most helpful.

While the SAS macro language supports positional parameters, it is usually better to use keyword parameters. This makes reading the macro call statement much more clear. For example:

Positional parameters:

```
%mymacro( value1, value2, value3 ) ;
```

Keyword parameters:

```
%mymacro( input= value1
        , output= value2
        , type= value3
        ) ;
```

Also note that I put each parameter on a separate line, preceded by a comma (rather than putting the comma behind the parameter on the line above it. This makes it easier to add and remove parameters.

Template Macro: Debug Level Option Parameter

All macros, other than the utility macros, have debugging statements used to print the status of the macro and/or DATA steps. The amount of debugging information required can vary, depending on the problem encountered while developing or fixing a macro. We control this with the DEBUG= parameter on the macro prototype statement.

0 = No debugging (default value)

1 = Very high level debugging

Start of major function blocks

2 = Intermediate level debugging

Outer loops

3 = Most detail level debugging

Inner loops

Final table values

Template Macro: Macro Statement Options

Following the macro prototype, we have a few compile time options that are normally commented out. Being commented out allows us to quickly enable them without looking up their syntax.

The STORE and SOURCE options are for saving the compiled macro. STORE saves it and SOURCE saves the source code with the compiled macro in case the original source would be deleted. The saved source can be retrieved with the command:

```
%copy <macroname> / source;
```

While reviewing the SAS 9.2 "What's New" documentation, I came across two new compile time options that may be of interest once we upgrade from 9.1. I added them to the macro template as comments for future use.

SECURE – encrypts the stored compiled macro, protecting your intellectual property.

MINDELIMITER – specifies the character to be used as the delimiter for the macro IN operator, overriding the value of the MINDELIMITER= global option.

Template Macro: Comment Header Block

Every macro has a comment header block, where we briefly document the macro and track revisions.

```
/* ********************************************************************
*
* Refer:    macros(xxxxxxxx)
*
* Function: xxx
*           xxx
*           xxx
*
* Notes:    Change all occurrences of xxxxxxxx to lowercase macro name.
*           and delete or replace these notes.
*           xxx
*
* Globals:  varname1 OUTPUT Usage...
*           varname2 INPUT  Usage...
*
* (C) Copyright 2010 We Energies
*
* Revision history:
*
* Date     Name           Description of Change
* -------- -------------- -------------------------------------------------
* mm/dd/yy J Doe          Initial version.
*
******************************************************************** */
```

The first line is the name of the macro file, using the "macros" fileref syntax. This allows us to use "macros(xxxxxxxx)" fileref name for INCLUDE and FILE commands, with copy and paste.

The Function section briefly describes the function of the macro.

The Notes section any relevant usage notes, describing the origin of any input, special commands required, etc.

The Globals section defines the global macro variables that this macro uses, creates or updates.

We have a short copyright line in the macro, so we may get some credit should we share the macro outside the company.

The Revision History block tracks changes to macro. Each change is documented with the date of change, the name of the person making the change and the description of the change. We always start with an "Initial version" line and then add lines each time the macro is revised. In addition to describing the changes we also document the reason for the changes (when it's not obvious), to help us recall why we did something in the future.

Template Macro: Macro Setup

In this section of the macro, we do all the pre-execution edit checking and validation, along with displaying standard debugging information and define the local macro variables used.

```
%let k1       = %upcase(&k1      ) ;

%if &debug > 0 %then %do ;
    %put %str( );
    %put DEBUG: &sysmacroname: macro starting.;
    %put %str( );
  %end;
%if &debug > 2 %then %do ;
    %put DEBUG: &sysmacroname: _USER_ macro variables and values at start:;
    %put _user_ ;
    %put %str( );
  %end;

%local errflag      /* Error flag                   */
       ts_start     /* Date/time of start           */
       ts_end       /* Date/time of end             */
       ts_elapsed   /* Elapsed time in seconds      */
       now          /* Date/time stamp for messages       */
       dsid         /* Temp dataset ID from open/close    */
       num_obs      /* Temp number of obs in output table */
       rc           /* Temp return code from sysfunc      */
;

%let errflag = 0 ;     /* Set error-found flag OFF */

%if %length(&p1      ) = 0 %then %do ;
    %put ERROR: &sysmacroname: First positional parm cannot be missing. ;
    %let errflag = 1 ;
  %end;

%if %length(&p2      ) = 0 %then %do ;
    %let p2      = default-value ;
    %put INFO: &sysmacroname: P2 is missing, defaulting to &p2.. ;
  %end;

%if %length(&out) = 0 %then %do ;
    %put ERROR: &sysmacroname: OUT= parm cannot be missing. ;
    %let errflag = 1 ;
  %end;

%if &p1       = m        %then %do ;
*   some statement ;
  %end;
%else %if &p1       = n       %then %do ;
*   some statement ;
  %end;
%else %do;
    %put ERROR: &sysmacroname: P1        =&p1 is invalid. ;
    %let errflag = 1 ;
  %end;

%if %length(&mail_to) = 0 %then %do ;
    %let mail_to = &sysuserid.@we-energies.com;
    %put WARNING: &sysmacroname: The MAIL_TO parameter is missing.  %qcmpres(
                  Set) to &mail_to.. ;
  %end;

%if &errflag ^= 0 %then %do ;
    %put ERROR: &sysmacroname: Fatal error(s) encountered.  %qcmpres(
                Macro) expansion aborted. ;
    %goto exit ;
  %end;
```

Our standard is to set all enumerated (key) values parameter values to upper case. This makes checking of values easier throughout the rest of the macro code and the upper case values stand out better.

Next, we report the start of the macro and display macro variable values on startup (if the DEBUG parameter was set to 3).

We then define and describe local macro variables. While SAS does not require local variable to be defined (with the %LOCAL statement), it prevents you from accidentally referencing or changing a global variable.

The next section checks for values that must always be specified, provides the default value if a parameter is set to null, checks values for enumerated parameters and will abort the macro if anything serious is detected. Doing this before the macro starts executing can save significant amounts of machine and user time if an invalid parameter value was entered.

Template Macro: Macro Timing

It is often helpful to display the current date/time at critical points during the execution of a long running macro.

```
%let ts_start = %sysfunc( datetime(), 12. ) ;
%let now = %sysfunc( putn( &ts_start, datetime. ) ) ;
%put INFO: &sysmacroname: &now: Starting execution. ;
```

Points to display the time include:

- At the start of macro.
- At the end of macro, with elapsed time.
- At important interim steps.
- At each iteration of long-running loops. Be sure to put the loop's BY values in message.

Template Macro: Starter Code for Runtime Processing

The template macro also includes often used DATA step and PROC code. Again, it's easier to delete something unwanted than to add it, especially if you have to search for it to copy and paste.

Our sample code includes:

- A DATA step, with the most frequently used statements and options. It also has debugging code controlled by the DEBUG= parameter.
- Macro code to get the number of observations in new output table:

```
%let dsid    = %sysfunc(  open( &out ) ) ;
%let num_obs = %sysfunc( attrn( &dsid, nobs) ) ;
%let rc      = %sysfunc( close( &dsid ) ) ;
```

- Three ways to e-mail reports and graphs.

Template Macro: Macro Termination

In this section of the macro, we:

- Report the ending of the macro, with a timestamp and total elapsed time.
- Display macro variable values on exit (based on the setting of DEBUG=3), to see changes to global variables and to see the final values of local variables.

Template Macro: Sample Testing Code

Then, after the macro's %MEND statement, we have some code to help us develop and debug the macro. This code should be deleted when the final macro is saved to the production macros fileref.

```
options
        mprint      /* Enable macro printing  */
        mprintnest  /* Nest macro calls in MPRINT output */
/*      mlogic      /* Enable macro logic printing  */
/*      mlogicnest  /* Nest macro logic in MPRINT output */
/*      symbolgen   /* Display resolved macro variable values */
        mexecnote   /* Display macro execution information */
        mautolocdisplay  /* Display source of macro when called */
;

%xxxxxxxx( p1= m
         , p2=
         , in= temp (obs=3)
         , out= temp
         , debug=2 ) ;

options nomprint nomprintnest nomlogic nomlogicnest nosymbolgen
        nomexecnote mcompilenote=none ;

/* New automatic macro variables with SAS 9.2 to show last error and warning:

%put Last Warning Message: &syswarningtext.. ;

%put Last Error Message: &syserrortext.. ;

/* ... end of comment block */
```

We start with the macro debugging options, with the most detailed ones commented out.

Then we have the actual macro call. Of course we have to set all the parameters correctly, but it's a start.

We then reset all the macro debugging options to OFF.

Finally, we use the new SAS 9.2 automatic macro variables to display the last warning and error messages. These are commented out for now.

Macro Documentation

In addition to the brief documentation in the comment header section in the macro and the imbedded comments within macro (to break macro into sections, explain "tricky" code or explain why something was done), we also document each macro with a Word document.

We only create Word documentation for production code. It's designed to help the end user (sometimes me at a much later date!) run the macro. It describes:

- The function of the macro.
- A summary of its syntax.
- An overview of the macro's usage.
- The macro's parameters and their values.
- The input and outputs.
- Any special usage notes.
- Examples of calling the macro.

SAS Environment for Macros

A major part of what makes this all work is the environment we setup before starting SAS and during the initial SAS startup.

Create a directory in your shared file system for all SAS code and split it into different subdirectories by code type. In our environment, we have separate subdirectories for:

- Macros
- Formats and informats
- Normal production SAS code
- Template SAS code
- Personal development SAS code, with one subdirectory for each SAS user

All of our SAS code, production, development and personal, is under the same higher level directory. This makes searching the code (with the Unix grep command) for key phrases very easy.

Also, each of these directories has a "backup" and an "obsolete" subdirectory. Before we change a code file, we always copy it, with the current date as the file name's suffix, to the backup subdirectory. This allows us to easily check the differences (with the Unix diff command) and restore the prior version in case of a problem. Instead of deleting files, we move them to the obsolete subdirectory, just in case we need to quickly restore them or want to get some information from them (for those "I know we used to do something like that" situations).

Unix Shell Script to Start SAS

We have written a simple shell script to setup the runtime SAS environment for our work group. It defines several environment variables (to externalize paths to code and data as much as possible) and then calls the standard SAS script with options to complete the environment setup.

```
setenv SASCODE   /<some-path>/sascode
setenv SASDATA   /<some-path>/sasdata
setenv PRINTER   <default-printer-name>

set parms = "$argv[*]"

sas913 \
        -autoexec '$SASCODE/source/autoexec.sas' \
        -insert sasautos '$SASCODE/macros' \
        $parms
```

The two options used are:

-insert -- tells SAS where to find our macros

-autoexec -- tells SAS to run a specific program first to set up the complete environment.

Then, after the autoexec program runs, each person includes and runs a personal startup program they run to further customize their environment to due things like override the default printer names and allocate personal databases.

Miscellaneous Tips and Techniques

Always use filerefs in all SAS code.

Never the physical path names in your SAS code, except on a FILENAME or LIBNAME statement. Using just the fileref in your code improves code portability. When we migrated our SAS work from mainframe to Unix, we ported many of the mainframe macros, formats and informats to Unix with no changes. Also, it simplifies changing environments. For example, when the SAS server was upgraded, all physical paths changed but the only SAS code that was changed was our autoexec program.

Use macros and %INCLUDE in big production jobs

This allows you to break large jobs into small pieces, which are much easier to manage. Just add/remove calls in "mainline" SAS code file. The use of utility macros, to provide special small bits of code and calculations also makes it easier to accommodate changes.

Use the %qcmpres Macro to Put Multiple Lines Together

The %qcmpres macro, provided by SAS, is used to compress multiple blanks to a single blank and remove all leading and trailing blanks. This improves the readability and printability of very long messages written by %PUT in the source code.

Example:

```
%put ERROR: &sysmacroname: Fatal error(s) encountered.  %qcmpres(
                            Macro) expansion aborted. ;
```

This results in the following lines in the SAS log:

```
ERROR: testmacro: Fatal error(s) encountered.  Macro expansion aborted.
```

Code readability

Use lots of white space to make your code more readable. This includes:

- Blank lines
- Spaces
- Indentation

When commenting out a block of code, make it more visible by putting the starting /* and ending */ on separate lines, with blank lines around them, along with comments like:

```
<previous line of used code>

/* Start of comment block...

<long block of unused code here>

/* ...end of comment block */

<next line of used code>
```

This makes it very easy to see that the code is commented out and allows you to easily add a */ line after the “Start of comment block” line to quickly uncomment it. This is particularly useful when developing new programs where you don’t want to keep rerunning the same code again and again.

Single lines of code can be commented out with an asterisk (“*”) in the first column.

Conclusions

You can code higher quality macros to meet the needs of your production environment. The use of templates results in faster code development, results in more uniform code and is easier to maintain in the future.

External documentation makes it easier for others to use your macros.

Contact Information

Gerry Bonin
We Energies
W237 N1500 Busse Rd.
Waukesha, WI 53188
Phone: (262) 544-7091
FAX: (262) 574-6080
E-mail: gerry.bonin@we-energies.com