Paper 19-26

Advanced Macro Topics

Steven First, Systems Seminar Consultants, Madison, WI

Abstract

The SAS macro language continues to be an integral part of the SAS system, and can be a wonderful tool or an overcomplicated solution. This paper will be of interest to macro users that would like to more fully understand the macro system.

This paper will discuss the following topics:

1. Macro data structures and timing
2. Local and Global referencing environments
3. Quoting
4. Macro and data step functions and call routines
5. Debugging and tracing techniques
6. Autocall facilities
7. Design considerations
8. Alternatives to SAS macros
9. Changes and enhancements with Version 7 and 8

Introduction

The SAS macro facility gives the SAS programmer some very interesting power and at the same time requires more understanding than the SAS data or PROC step. While it can make SAS jobs more robust and easier to code, it can also be easily overdone with the result being a more complex system than is sometimes necessary. This paper will explore some of the more advanced features of the macro facility, but at the same time strive to keep things as well documented and simple as possible.

Macro Data Structures And Timing

The SAS Macro facility is a second language that can be intermixed with normal SAS statements. Unlike normal SAS statements, the function of macro statements is to generate SAS statements or perhaps just part of SAS statements during the word scan and compile process.

Without macros, SAS programs are DATA and PROC steps. The program is scanned one statement at a time looking for the beginning of step (step boundary). When the beginning of step is found, all statements in the step are compiled one at a time until end of step is detected. When the end of step is found (the next step boundary), the previous step executes. This interleaving of compiling and execution can be confusing to many users. Other languages also have compile and execute phases, but typically those programs are similar to a single SAS step. Because in SAS a step is compiled, then executed, before the next step is compiled and executed, it is easy to see how the timing can be difficult to comprehend.

Again, most of the work of the macro facility is performed during word scan and compile, with the DATA or PROC step doing the work at step execution time.

SAS Step Boundaries

If it is important for the user to understand the timing of compile and execution, it is critical to define the start and end of a SAS step.
The beginning of steps are obviously the DATA or PROC statements. The ends of steps are a little more difficult to detect. The end of a step for our purposes is whenever SAS encounters another step boundary or end of job.

The following keywords are step boundaries to the SAS system:

DATA	ENDSAS
PROC	LINES
CARDS	LINES4
CARDS4	PARMCARDS
DATALINES	QUIT
DATALINES4	RUN

In the following program, the word scanner can only detect end of step when the next step starts. The last step will start compile, but will only execute if run in batch because there is no explicit end of step give. Interactive sessions will disply the message “PROC MEANS is RUNNING”, but it really is in the word scan phase. This is not a very good way of coding a SAS job even though it may work in some instances.

```
data saleexps;            <--Step, start compile
 infile rawin;
 input name $1-10 division $12
       years 15-16 sales 19-25
       expense 27-34;
proc print data=saleexps; <--Step end, exec prev
                             Step start, compile
proc means data=saleexps; <--Step end, exec prev
 var sales expense;          Step start, compile
```

The RUN statement does not actually end some PROCs, but it does tell the wordscanner to stop compiling and ‘RUN’ the step. It is highly recommended that the RUN statement be coded, as there is then no question as to when compile finishes and execution starts.

```
data saleexps;            <--Step, start compile
 infile rawin;
 input name $1-10 division $12
       years 15-16 sales 19-25
       expense 27-34;
 run;                     <--Step end, exec prev

proc print data=saleexps; <--Step start, start
   compile
 run;                     <--Step end, exec prev
proc means data=saleexps;
 var sales expense;
 run;                     <--Step end, exec prev
```

It shold be noted that global statements such as title, footnote, options, libname etc. are compiled, and then executed immediately. Again by explicitly coding RUN statements, the programmer has absolute control over which steps global statements affect.

Timing with Macro Variables

As %LET statements define macro variables, and those variable are referenced by name and ampersand what are the timing considerations?

%LET and & are processed during the word scan and compile phases, which remember always precede the execution phase. The %LET statement below defines a macro variable called dsname which is then referenced many times later in the program. This is a very simple substitution and it works very well. In fact, quite often an automatic variable is defined by SAS and we don’t even need to code the %LET.

```
%let dsname=saleexp;
data &dsname;            <--Step, start compile
 infile rawin;
 input name $1-10 division $12
       years 15-16 sales 19-25
       expense 27-34;
 run;                    <--Step end, exec prev

proc print data=&dsname; <--Step start, comp
 run;                    <--Step end, exec prev
proc means data=&dsname;
 var sales expense;
 run;                    <--Step end, exec prev
```

A Timing Error

The program below attempted to set a macro variable to either Hardware or Software based on the values read in Division. In this case there were only "H" division records, but yet when the title prints, "Software division" is displayed. Why?

```
data saleexps;
 input name $1-10 division $12 ;
 if division='H' then
    %let mdiv=Hardware;
 if division='S' then
    %let mdiv=Software;
 datalines;
Steve      H
Bob        H
;
run;
proc print data=saleexps;
 title "&mdiv division";
run;
```

The program has timing confused. The %LET statements set the mdiv values at COMPILE time and the first %LET statement moves Hardware to the macro variable, but then the second %LET statement moves Software into the same macro variable. The resulting value is then from the last %LET statement processed during wordscan. It is much later that the data step executes the IF statements which now effectively do nothing since %LET has no effect at data step execution time. I have seen this error in several programs where the timing wasn't understood completely.

Execution Time Macro Components

To solve the problem above, the programmer really needed to use a statement to set the macro variable's value at data step execution. The most commonly used routine is CALL SYMPUT. This statement creates a macro variable at data step EXECUTION time.

```
data saleexps;
 input name $1-10 division $12 ;
 if division='H' then
    call symput('mdiv', 'Hardware');
 if division='S' then
    call symput('mdiv', 'Software');
 datalines;
Steve      H
Bob        H
;
run;
proc print data=saleexps;
 title "&mdiv division";
run;
```

Macro variables created via CALL SYMPUT cannot be referenced with an & until a later step again because of timing.

```
data saleexps;
 input name $1-10 division $12 ;
 if division='H' then
    call symput('mdiv', 'Hardware');
 if division='S' then
    call symput('mdiv', 'Software');
 title "&mdiv division";
 datalines;
Ssteve      H
Bob        H
;
run;
proc print data=saleexps;
 run;
```

The reference to &mdiv occurs at wordscan time which again is before execution time when SYMPUT executes. In order to make the &mdiv resolve correctly, it must be referenced in a later step. This again underscores the importance of step boundaries.

Another Timing Error

Suppose we want to retrieve a macro variable value that was created in the same step. This might be because the user is interacting with the step, or perhaps we are using a stored compiled step or an SCL program.

```
data saleexps;
 input name $1-10 division $12 ;
 if division='H' then
    call symput('mdiv', 'Hardware');
 if division='S' then
    call symput('mdiv', 'Software');
 newvar="&mdiv";
 datalines;
Steve      H
Bob        H
;
run;
```

Again the reference to &mdiv would be attempted before it was created by SYMPUT, even though it appears later in the program. The run time retrieval routine is typically SYMGET which works fine.

```
data saleexps;
 input name $1-10 division $12 ;
 if division='H' then
    call symput('mdiv', 'Hardware');
 if division='S' then
    call symput('mdiv', 'Software');
 newvar=symget('mdiv');
 datalines;
Steve      H
Bob        H
;
run;
```

Macro Structures and Conventions

The structures and storage used by the macro facility are also located in separate areas from the SAS DATA and PROC steps. SAS observations are normally stored on disk or tape, and when processed by the DATA step, values are normally stored in the program data vector. In the DATA step, variables are referenced by name, and they are not quoted. Numeric constants are not quoted, but single or double quotes are used around character constants to differentiate them from numeric constants and variables.

In the macro language since everything is a character value, no quotes are needed, and if quotes are included they are usually treated like any other character. One exception is that macro variables are not resolved within single quotes and they are resolved within double quotes. The semicolon is also not needed as much in the macro lanquage as it is in SAS.

Macro variable names and their corresponding values are stored in memory locations called symbol tables and they are not retained beyond the current job. Macro variables are not normally named with a & when defined with a %LET statement, but are normally prefixed with & when referenced later. In the example below, the macro variable name (dsname) is a constant and the value (saleexps) is also a constant.

Example:

```
%let dsname=saleexps;
data &dsname;
etc.
```

There is no reason the variable name itself can't be a macro variable reference. In the example below the macro variable name includes an & which means that the name itself varies and will need to be resolved before creating the new variable.

Example:

```
              /* name dsname, value saleexps */
%let dsname=saleexps;

              /* name saleexps, value abc */
%let &dsname=abc;

             /* name saleexps, value abc */
data &sallexps;
etc.
```

Multiple Ampersands

The double ampersand (&&) is a special reference that always resolved into a single ampersand.

Example:

```
%let c=hallo;
%put &c &&c;
```

The macro processor resolves the first reference into hallo. In the second reference, the macro processor resolves && into & and the scans the letter c. Next, because the macro processor always rescans an item resulting from a resolution, it scans the remaining &c giving the same result hallo and displaying:

```
hallo hallo
```

Using a double ampersand in this case gives the same result as a single ampersand. Why then would we ever need multiple ampersands? One example is when we have a series of similarly named variables then end in consecutive numbers that we may want to index through.

For example, we have a series of county datasets that we would like to print with a macro loop. A very common technique is to store the names in separate variables along with a final variable containing the number of other variables, then loop through them.

Solution 1: Incorrect

```
%let cnt1=ashland;
%let cnt2=bayfield;
%let cnt3=washington;
%let totcnt=3;
%macro cntprt;
 %do I=1 %to 3;
   proc print data=&cnt&I;
   run;
%end;
%cntprt
```

The above fails, because there is no variable called cnt, so we have to delay the resolution with multiple ampersands. Using &&cnt&I resolves &&cnt to &cnt and &I to 1 on pass 1, then rescans the result (&cnt1) to resolve to ashland etc. This indirect referencing technique is used in many macros that use looping.

Solution2: Correct

```
%let cnt1=ashland;
%let cnt2=bayfield;
%let cnt3=washington;
%let totcnt=3;
%macro cntprt;
 %do I=1 %to 3;
   proc print data=&&cnt&I;
   run;
%end;
%cntprt
```

Triple Ampersands

Three ampersands can be used in the case where the value of one variable is the name of a second variable whose value you would like to retrieve.

Example:

```
%let c=data;
%let data=year2001;
data &&&c;
 etc.
```

During scan one, && resolves to &, &c resolves to data. Scan 2 resolves the remaining &data to year2001.
More than three ampersands can be used though this is done rarely. In any case the macro processor resolves two ampersands into a single ampersand and then rescans all text generated by the previous scan.

Macros Versus Macro Variables

SAS macros (macro programs) allow much more than simple substitution of values, they introduce logic. This can also introduce unneeded complication to the program if only substitution is needed.

The macros themselves go through the word scan process and are normally stored in a work area until invoked. Another way of saying this is that macros themselves are compiled and invoked (executed). Remember that the macro may generate steps that are compiled and executed as well.

Referencing Environments

As macros define their own macro variables, the concept of referencing environments is necessary. Every SAS program has one or more referencing environments. A referencing environment is the area in which a macro variable is stored and later retrieved. Referencing environments are especially important when using multiple macros that could conceivably use the same variable names for different values, and thus contaminate or destroy results.

Referencing environments are large areas surrounding progressively smaller areas. The largest and outermost area is the global area. The automatic SAS variables, those variables created outside of macros, as well as most macro variables created with SYMPUT exist in the global environment. Variables stored in the global environment are available anywhere in the SAS job.

Each macro that you invoke creates its own local referencing environment. The local referencing environment is empty until the macro creates a macro variable. This environment exists only while the macro is executing at which time the storage is freed to the system. Also any nested calls to macros will result in

nested local environments.

The environment for the currently existing macro is called the current macro environment.

How the Macro Processor Uses Referencing Environments

When creating macro variables, the macro facility tries to change any existing macro variable rather than creating a new one. If no existing variable is found, the macro processor creates the new variable in the current environment. The %LOCAL and %GLOBAL statements can be used by the programmer, to control these actions.

While this seems complicated, in practice it is not usually very difficult to deal with. In most cases, if all macros are defined in the same job, each macro creates it's own variables as the macro executes, and then those variables are deleted. When macros call one another is the case that usually causes problems.

If a calling macro and the called macro accidentally use the same variable names for different values, there is a good chance that the inner macro may contaminate the value of the outer macro. To prevent this the %LOCAL statement can tell the macro facility to define it in the local environment, regardless of where else it may be defined. It is always good practice to define all variables in a macro as local unless there is some reason to not do so. Unfortunately, almost nobody uses %LOCAL even though they probably should. This is especially important to utility-type macros that could be called by other macros.

Using %LOCAL isn't always the answer to all problems, though. Suppose we want to write a utility macro that performs some task, and then reports the success to the caller through some sort of macro variable used as a return code. Without taking special action that variable would only be defined in the local environment, and as such the caller would have no access to it. In this case, the only option is to define the return code variable as global and hope that the caller or other macros are not using the same named variable.

Obviously well chosen names and good documentation can help tremendously when building a library of macros. Without good conventions and documentation, utility macros can be very difficult to use and understand, and some very strange results can occur.

Macro Quoting

All programming languages need some way to differentiate that certain characters are to be treated as a text value instead of some instruction or operator used in the language. In the data and PROC steps, and in most other programming languages, a single or double quote mark serves this role.

In the example below when var1 is not quoted, it refers to the name of an existing data step variable. When it is quoted it is simply four constant characters. Likewise the semi-colon marks the end of all SAS statements, but in addition when Z is set to the quoted semicolon, the quotes cause the system to treat the semicolon as it would any other character.

```
data temp;
  set ds1;
  x=var1;
  y='var1';
  z=";";
run;
```

Obviously the macro facility has special symbols that may need similar treatment. However the macro facility does not use the single or double quote character to mask these characters. Instead it uses a variety of functions to do what quotes do in other languages, thus the name 'macro quoting'. Another way of stating this is that the actions that you take to cause the macro facility to treat a certain character as text rather than part the macro language is called "macro quoting". Similarly when the literature mentions that the "result is quoted", it does not mean that actual quote marks are inserted, but rather the result is treated like text and the meaning of the special characters are removed.

Different Kinds of Quoting Functions

Compilation Functions

There are functions that cause the macro facility to treat items as text during macro compilation or when used in open code. %STR and %NRSTR are examples. %STR removes meaning of the following: ;'+-*/**~=<>,*LT LE EQ NE GT OR AND & trailing and leading blanks. %NRSTR (no rescan string) processes all of the above and also prevents rescanning (ignore meaning of & %).

Examples:

```
%let name=%str(proc print;run;);
%let name=%str(    &company);
%let desc=%nrstr(    %of total report);
```

Execution Functions

There are functions cause the macro facility to treat items as text during macro execution as well. %QUOTE and %NRQUOTE operate at execution time and remove meaning of most characters in the result of the macro call. %BQUOTE and %NRBQUOTE should be used if the value contains unmatched quotes or parentheses.

Examples:

```
%macro getst(state,employee);
%if quote(&state) = %str(NE) %then
   %do;
      %let longst=Nebraska;
  %end;
  %put hello &employee from &longst;
%mend;
```

%getstate(NE,O'brien)

Functions to Prevent Resolution

%SUPERQ is a function that will prevent starting any resolution of macro variables. This is most needed in windowing operations such as SAS/AF, %WINDOW, or SAS/INTRNET screens, or after CALL SYMPUT if there is any chance the input value could contain an ampersand or percent sign. Other functions do not work as well in this case, and would issue warnings and recursion messages.

Example:

```
data _null_;
   call symput('mv1','Bob&Fred %macro report');
   run;
%let mv2=%superq(mv1);
%put mv2=&mv2;
```

The Result:

```
mv2=Bob&Fred %macro report
```

Unquoting

The effects of quoting can be removed if desired by the %UNQUOTE function. In the example below var2 is not resolved but using %UNQUOTE later allows the resolution to take place.

Example:

```
%let mv1= hallo;
%let mv2=%nrstr(&mv1);
%let mv3=%unquote(&mv2);
%put mv1=&mv1 mv2=&mv2 mv3=&mv3;
```

The Result:

```
mv1=hallo mv2=&mv1 mv3=hallo;
```

Other Macro Functions

There are several other character functions that do various manipulations on macro values. Most of these functions are very similar to functions in the data step and perform the same way to substring data, left align, right align, parse words etc. In many cases there is a quoted version of the function where the name starts with a Q (ex. %QSUBSTR). These functions are sometimes more difficult to code and debug than counterparts in the DATA step mostly because the DATA step is a more comprehensive and robust language.

Using Data Step Functions Within Macros

The macro facility can have access to the DATA steps function library of over 300 functions via the %SYSFUNC macro function. This can be extremely handy to include any of the logic from existing functions. %SYSFUNC can also apply a format to the result from the function. There are some functions such as the LAG function that cannot be used because LAG needs to read prior records in a data step which might not exist in a macro application.

Example: Extract today's date, format it as a worddate, store the final result in a macro variable.

```
%let mydate=%sysfunc(date(),worddate.);
%put &mydate;
```

Displays:

```
January 15, 2001
```

Debugging and Tracing Techniques

There are essentially five debugging tools in the macro facility.

1. %PUT
2. SYMBOLGEN
3. MLOGIC
4. MPRINT
5. MFILE

The %PUT statement is probably the best and simplest method of displaying values at word scan time. %PUT can display text or macro variable references and calls. Like the PUT statement from the DATA step, %PUT can be placed at strategic spots in your program to display debugging information. Like other items in the macro facility, quotes are not needed to display text.

Example:

```
%let var1=Stevo;
%put ***** var1=&var1;
```

Displays:

```
**** var1=Stevo
```

A fairly recent addition is parameters to %PUT that not only display a single variable, but can show several.

ALL	Shows all variables and there respective symbol table
AUTOMATIC	shows only the system variables
USER	displays only user variables
LOCAL	displays only local variables
GLOBAL	displays only global variables

Using _ALL_ is really the only practical way to determine whether a variable is defined locally, globally or both. This was requested for many years, and it's now available.

The other four debugging tools are system options.
SYMBOLGEN shows macro variables as they are being resolved.
MLOGIC displays decisions and looping that the system makes with %DO, %IF etc.
MPRINT displays the generated code sent to the SAS compiler.
MFILE routes the MPRINT results to a separate file.

MFILE can be useful to give us an answer to a very difficult problem. Suppose the following macro is submitted with a semicolon missing after the VAR statement. The statement number displayed in the log is the line number of the statement that called the macro. This isn't overly useful, since the macro could contain or generate thousands of lines, and we have an error somewhere in it.

```
1    options mprint nomfile;
2     %macro printmac(msasds=invoice);
3      proc print data=&msasds;
4       by company;
5       id days;
6       var rate age    /* note missing ;*/
7       title "listing of &msasds";
8      run;
9     %mend printmac;
10
11    %printmac(msasds=invoice)
MPRINT(PRINTMAC):   proc print data=invoice;
MPRINT(PRINTMAC):   by company;
MPRINT(PRINTMAC):   id days;
NOTE: Line generated by the invoked macro
"PRINTMAC".
11      proc print data=&msasds;    by company;
id days;    var rate age             title
"listing of &msasds";   run;

-

200
ERROR 200-322: The symbol is not recognized and
will be ignored.

NOTE: Line generated by the macro variable
"MSASDS".
11   "listing of invoice
      --------------------
      22
MPRINT(PRINTMAC):   var rate age title "listing
of invoice" run;

ERROR 22-322: Syntax error, expecting one of the
following: a name, ;, -, :, _ALL_, _CHARACTER_,
_CHAR_, _NUMERIC_.
```

As a last resort, the generated code could be routed to a file, manually included and run. Since the code no longer contains any macro statements or conventions, statement numbers will be more meaningful.

```
filename mprint 'c:\temp\pmaccap.sas';
options mprint mfile;
 %macro printmac(msasds=invoice);
  proc print data=&msasds;
   by company;
   id days;
   var rate age     /* note missing ;*/
```

```
   title "listing of &msasds";
  run;
 %mend printmac;

 %printmac(msasds=invoice)

options nomfile;
```

The captured program looks like the next few lines from the SAS log, where it easier to find the offending statement.

```
53   proc print data=invoice;
54   by company;
55   id days;
56   var rate age title "listing of invoice"
run;
                         ------------------- ---
                         22                  202
ERROR 22-322: Syntax error, expecting one of the
following: a name, ;, -, :, _ALL_, _CHARACTER_,
_CHAR_, _NUMERIC_.
```

The SAS Autocall Facility

There are five methods of making a macro available to your SAS job:

1. Define a macro in the same program that uses it.
2. Use %INCLUDE to include macros stored in external files.
3. Use the INCLUDE display manager command to manually include the macro.
4. Use the autocall facility to search a predefined macro library.
5. Use the Stored Compiled Macro Facility.

In my opinion the first three methods work fine for many systems, and the compiled macro facility is probably not worth the bother. While it might save a slight amount of word scanning, the user now has to manage both compiled code and source code which can be very difficult to deal with especially with new releases.

A good option to handle production macros that will be used by many users is to store them in an autocall library. By making this library available to all users, they can call macros without including them first.

Using the autocall facility requires five steps:

1. Create a macros source library appropriate for your operating system.
2. Place each macro as a separate member in the library. The member name must be the same as the macro name
3. Associate the fileref SASAUTOS with the library.
4. Turn on the MAUTOSOURCE SAS system option.
5. Invoke any stored macro.

Invoking a macro that has NOT been defined in the current session causes the macro facility to:

1. Search the autocall library for the member named the same as the invoked macro.
2. Issue an error message if the macro is not found.
3. Automatically include the macro source if found.
4. Submit the macro.
5. Call the macro.

The autocall facility is a PDS under OS/390 and is a directory in the directory based systems. You can also concatenate several libraries and in fact several macros are shipped with the SAS system and the user only needs to call them as they would a macro function.

Examples of SAS-Institute Autocall Macros (partial list)

```
%COMPSTOR(arg)          compiles autocall members
%CMPRES(arg)            remove multiple blanks
%DATATYP(arg)           determine argument data
                        type
%LEFT(arg)              remove leading blanks
%LOWCASE(arg)           translate to lower case
%MDARRY                 simulate multi-dimension
                        arrays
%QCMPPRES(arg)          quoted form of %CMPRES
%QLEFT(arg)             quoted form of %LEFT
%QLOWCASE(arg)          quoted form of %LOWCASE
%QTRIM(arg)             quoted form of %TRIM
%SUPERQ(variable)       remove meaning of all
                        special characters
%SYSGET(arg)            returns oper system vars
%SYSRC(arg)             translates numerics to
                        mnemonics
%TRIM(arg)              trims trailing blanks from
                        arguments value
```

An example of using an autocall macro:

%TRIM can be used to trim trailing blanks from macro values.

```
%LET X=%STR(MADISON   );
%LET Y=%TRIM(&X);
%PUT ---&X--- ---&Y---;
```

Generates:

```
---MADISON   --- ---MADISON---
```

Design Considerations

Since the macro system has a unique purpose, it stands to reason that design may be unique.

It is my opinion that the best systems:

1. Minimize macros
2. Clearly document macros and their purpose
3. Display source of all modules at least while testing.
4. Keep things simple

Minimizing macros

The are many cases where macros have been written where a SAS option or DATA statement may work much better. Examples are the CNTLIN option for building formats, the #BYVAL variable for displaying by variables in titles. Macros are simply more difficult to work with if there is another reasonable alternative.

Below is a pair of macros that create titles. The first technique uses only macro statements and the second one does most of the work in the DATA step. Because of the robustness of the DATA step and the limitations of the macro language, the second approach may be simpler, though this approach obviously wont work for all cases.

In both cases the macro call below generates the title following the call.

```
%ssctitlm(1,
          3,'left edge',
         20,'sample center',
         40,"date run:&sysdate")
```

Generates

```
title1
"  left edge        sample center
            date     run:13DEC00";
```

Solution one done completely with macros:

```
%macro ssctitlm(mtitlno,mcol1, mtext1,
                mcol2, mtext2,
                mcol3, mtext3);
/*************************************************/
/* macro ssctitlm                                */
/*  purpose:  create sas title variables forprocs*/
/*    input:  title number, starting col, text up */
/*            to 3 segments of title.             */
/*   output:  title vars that can be used by sas  */
/*************************************************/
/* need data step logic to build data step vars, */
/* put in macro variables at end.                */
/*************************************************/
%let noblank=%eval(&mcol1-1);         /* # blks left */
%let mtext=%str();                    /* set to null */
%do i=1 %to &noblank;                 /* do # blanks */
 %let mtext=%str(&mtext)%str( );      /* insert blk  */
%end;                                 /* end of loop */

%let tlen=%eval(%length(&mtext1)-2);  /* no 's       */
                                      /* strip out 's*/
%let mtext1=%substr(%str(&mtext1),2,&tlen);
%let mtext=%str(&mtext)%str(&mtext1); /* app to mtext*/
                                      /*# bls text2  */
%let noblank=%eval(&mcol2-1-%length(&mtext));
%do i=1 %to &noblank;                 /* do # blanks */
 %let mtext=%str(&mtext)%str( );      /* insert blks */
%end;                                 /* end blk loop*/

%let tlen=%eval(%length(&mtext2)-2);  /* no 's       */
                                      /* strip 's    */
%let mtext2=%substr(%str(&mtext2),2,&tlen);
%let mtext=%str(&mtext)%str(&mtext2); /* app mtext   */
                                      /*# bls mtext3 */
%let noblank=%eval(&mcol3-1-%length(&mtext));
%do i=1 %to &noblank;                 /* do # blks   */
 %let mtext=%str(&mtext)%str( );      /* insert blk  */
%end;                                 /* end blk loop*/

%let tlen=%eval(%length(&mtext3)-2);  /* no 's       */
                                      /* strip 's    */
%let mtext3=%substr(%str(&mtext3),2,&tlen);
%let mtext=%str(&mtext)%str(&mtext3); /* app to mtext*/

 title&mtitlno "&mtext";              /* make title  */
/***********   end of macro ssctitlm  ***************/
 %mend ssctitlm;
```

The following macro generated the same title statement, but with a lot less coding effort.

```
 %macro ssctitle(mtitlno,mcol1,mtext1,
                 mcol2,mtext2,mcol3,mtext3);
  /*************************************************/
  /* macro ssctitle                                */
  /*  purpose:  create sas title variables for procs */
  /*    input:  title number, starting col, text for */
  /*            3 segments of title.                 */
  /*   output:  title vars that can be used by sas   */
  /*************************************************/
  /* need data step logic to build data step vars,   */
  /* put in macro variables at end.                  */
  /*************************************************/
data _null_;                          /* use data step*/
  length text $ 160;                  /* ds var mx len*/
  text=' ';                           /* blank it out */
  substr(text,&mcol1)=&mtext1;        /* first text   */
  substr(text,&mcol2)=&mtext2;        /* second text  */
  substr(text,&mcol3)=&mtext3;        /* third text   */
  call symput("mtitle&mtitlno",text);/* ds vars->mac */
 run;                                 /* end of step  */
 title&mtitlno "&&mtitle&mtitlno";    /* move to title*/
  /************** end of macro ssctitle **************/
 %mend ssctitle;
```

Clearly Documenting and Keeping It Simple

A recent program that a student offered used all of the below techniques:

1. Autocalled macros
2. Generated Code to a file, then %included it back to run.
3. %included macros
4. Inline macros
5. Nested macros
6. All source display (2000 lines) turned off
7. No commenting.

While any of the above techniques are fine, having so many pieces and no documentation really makes the job much more difficult than a few, well-documented techniques.

Changes and Enhancements

The major changes to the macro system is 32 character names and the %PUT _All_ Statement mentioned earlier. Below are some of the other changes implemented.

The following automatic macro variables are available in all operating environments.

SYSCC — contains the current condition code that SAS returns to your operating environment (the operating environment condition code).

SYSCHARWIDTH — contains the character width value.

SYSDATE9 — contains a SAS date value in DATE9. format, which displays a 2-digit date, the first three letters of the month name, and a 4-digit year.

SYSDMG — contains a return code that reflects an action taken on a damaged data set.

SYSPROCESSID — contains the process ID of the current SAS process.

SYSPROCESSNAME — contains the process name of the current SAS process.

SYSSTARTID — contains the identification number that was generated by the last STARTSAS statement.

SYSSTARTNAME — contains the process name that was generated by the last STARTSAS statement.

SYSUSERID — contains the user ID or login of the current SAS process.

%PUT — In Version 8, the %PUT statement has been enhanced. It displays text in different colors to generate messages that look like SAS-generated ERROR, NOTE, and WARNING messages.

%SYSLPUT — In Version 8, this new macro statement creates a new macro variable or modifies the value of an existing macro variable on a remote host or server.

The following new macro statement invokes a SAS CALL routine:

%SYSCALL

The following macro functions are new:

%SYSEVALF evaluates arithmetic and logical expressions using floating-point arithmetic.

%SYSFUNC and %QSYSFUNC
execute SAS functions or user-written functions

Conclusion

The SAS macro facility continues to have incremental improvements with each new SAS release, and with when used properly, it gives the tremendous power and capabilities.

Contact Information

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

Steven First
Systems Seminar Consultants
2997 Yarmouth Greenway Drive
Madison, WI 53716
608 278-9964 x 302 voice
608 278-0065 fax
sfirst@sys-seminar.com
www.sys-seminar.com