Macros from Beginning to Mend
A Simple and Practical Approach to the SAS® Macro Facility

Michael G. Sadof, MGS Associates, Inc.

ABSTRACT

The macro facility is an important feature of the SAS Language. Mastering this facility will enable the novice user to become an accomplished programmer. This tutorial session will use a building block approach taking the user from the most basic %LET statements to parameter driven, code generating macros that can save time and effort. The use of Macros will reduce the repetitiveness of manipulating data and will facilitate uniform and easy maintenance of the system. This paper will use step by step examples to show the user how to use macro substitution to provide simple yet powerful SAS code. The user will learn how to create variables on the fly, and how to use the macro facility to generate code as well as how to save sections of macro code for future use.

INTRODUCTION

This paper will discuss an easy way to get started with the SAS macro facility. It will help the novice SAS user bridge the gap between beginner and expert programmer. The SAS® Macro Language: Reference, First Edition states that: "The macro facility is a tool for extending and customizing the SAS system and for reducing the amount of text you must enter to do common tasks." It is a method of creating shortcuts in SAS programs. It enhances the SAS programming facilities and works with existing code. Macros can be used in the data step, in procedures, and Screen Control Language (SCL) code. The Macro facility works with all operating systems and comes as part of base SAS. It can be used as a way to create streamlined code. It will not necessarily make code in the data step run faster or more efficiently (although the stored compiled macro facility will eliminate the need for run time compilation). In some cases it may slow down the program due to the need to compile code at execution time. However it will reduce the amount of coding you actually have to write to accomplish data tasks. It will provide for easier maintenance, parameter driven, modularized code. It is true that the code looks more complicated and requires more testing but once you learn it, it will be easy and you will learn to rely upon it and enjoy using the SAS macro facility. This paper will not cover every aspect of the macro facility. It will, however, assist the novice user in developing familiarity and expertise with SAS macros.

%LET

There are several types of macro statements. The simplest, but very useful, form of the macro language is the assignment statement. The assignment statement in its simplest form looks like:

```
%LET macvar = text_string ;
```

This statement creates a macro variable named *macvar* and assigns it the value of *text_string*. Under most conditions (some of which will be discussed later) this newly created macro variable is available to every DATA step and PROC step to follow in the program. It must be referred to as *&macvar* and will evaluate to whatever is in *text_string*. The %LET statement may be used outside or within a data step. First let us see how you use the macro variable in a program. Since we are all familiar with the TITLE statement it will be used to demonstrate the evaluation of macro variables in programs. Please note that quotation marks are very significant in working with macros but there are a few tricks that will help you code efficiently without errors. Consider program fragment:

```
%LET city = Washington ;
TITLE "This city is &city" ;
```

When the TITLE statement executes it produces the title:

```
This city is Washington
```

Double quotes (**"**) rather than single quotes(**'**) are significant and necessary and the TITLE statement with single quotes (**'**) would evaluate with an undesirable result at execution time as:

```
This city is &city
```

If you use quotes when creating a macro variable in an assignment statement the quotes will be saved as part of the macro but this may be the desired result:

Code:

```
%LET city = 'Washington, DC' ;
TITLE "The address is &city" ;
```

The resulting title will look like this:

```
The address is 'Washington, DC'
```

It generally a good idea not to use quotes when

assigning macro variables with the %LET statement but rather use them as needed when recalling the macro value. Once assigned in a program (but not necessarily within a macro definition) the %LET statement creates a GLOBAL variable that is available for your use almost anywhere in your program after the assignment statement. (Global and local variables will be discussed subsequently under the subheading of SCOPE.) The macro is recalled by placing an ampersand(&) before the macro name. To insure that a macro variable will be available for the entire SAS session you may mention the variable in a %GLOBAL statement. Before we go into the scope of macro variables, it is important to look at some examples.

The most common uses for macro variables are as data file names or targets of logical-if statements. Consider the code:

```
%LET filemac = WASH ;
%LET city    = Washington ;
%LET select  = 14;

DATA &filemac;  SET scores ;
IF age = &select ;
IF city = "&city";
```

The code produced by this set of macro statements is as follows:

```
DATA WASH; SET scores;
IF age = 14 ;
IF city = "Washington";
```

When executed the program produces a dataset named WASH with information from the scores database where age = 14 for the city of Washington. The age variable has been defined as numeric in the Scores dataset. Notice that quotes are not used because we want the statement to evaluate as:

```
IF age = 14 ;
```

Rather than:

```
IF age = "14" ;
```

This will prevent numeric to character conversion. However double quotes are used in the *IF "&city"* ; statement because we want it to evaluate with the quotes as:

```
If city = "Washington";
```

There are many macro quoting and unquoting functions that will facilitate coding character strings under the most complex situations but these are not the subject of this paper. Some simple tricks, which work most of the time concerning the use of quotes, are as follows:

1. Do not use quotes in assigning macro variables with the %LET statement.
2. When using the ampersand variables in code use quotes if the expression should evaluate to a character string.
3. Do not use quotes if it should evaluate to a number.
4. Do not use quotes when invoking macro variables as file names.

In its most basic form the %LET assignment statements assigns a series of characters to a macro variable and places that GLOBAL variable into the global symbol table and makes that GLOBAL variable (or colloquially referred to as amper-variable) available to the rest of the program.

What is the advantage of using the %LET statement if it only substituting character strings in other places in the program? The macro variable will allow you to structure your programs in a more readable and maintainable fashion. Consider multiple TITLE statements scattered around in many places throughout your program along with multiple 'select IFs' or 'WHERE clauses' that subset the data. You would like the ability to identify the various subsets of data with an appropriate TITLE statement. By placing the %LET statements at the beginning of the code you can easily change cities or selections throughout the program.

```
%LET city = 'Washington, DC' ;
%LET filen = %substr(&city,2,4);
. . .
DATA &filen;  SET scores ;
IF city = "&city";
. . .
PROC PRINT;
TITLE
"Data for the city of &city";
```

As you build more and more complex programs and the programs get turned over to others to run the advantage of this syntax becomes more evident. This construction will simplify changes and might prevent errors especially those reruns of long data intensive programs because the title was displaying the wrong month or selection criteria.

CONCATENATION

Sometimes we want to build complex expressions with macro substitutions and combine macros in a linked series with static text. A simple example would be building dataset names with an embedded year for identification. Let's say we want to build the code that evaluates to the following:

```
DATA Sale1999 Sale1998;
```

```
SET AllSales;
IF year=1999 then OUTPUT Sale1999;
ELSE
IF year=1998 THEN OUTPUT Sale1998;
```

Converting to macros and parameterizing the code would look like this:

```
%LET curyr=1999;
%LET preyr=1998;

DATA Sale&curyr Sale&preyr;
     SET Allsales;
IF year=&curyr THEN OUTPUT Sale&curyr;
ELSE
IF year=&preyr then output sale&preyr;
```

Note that when Sale&curyr evaluates the result is Sale1999. There are no spaces in the resulting text which is exactly what we want. Now let's build a two part dataset name for a permanent SAS dataset.

```
Data Sale&yr..data; set temp;
```

In this instance we need two consecutive periods because the first period indicates to the macro word scanner that it is the end of the macro name (*&curyr.*) and the second period is treated as static text. "In a macro variable reference, the word scanner recognizes that a macro variable name has ended when it encounters a character that is not allowed in a SAS name."[1] The most common character used is the period. Two macros may be placed adjacent to each other and will evaluate properly such as:

```
%LET pyr=98;
%LET cyr=99;

Data Sale&pyr&cyr;
```

which evaluates to:

```
Data Sale9899;
```

The period is not needed here since the second ampersand indicates the end of one macro name and the beginning of another but it may be used for clarification such as:

```
Data Sale&pyr.&cyr. ;
```

Linking macros together in this fashion is very important in building complex macro expressions. With these techniques you can build programs that will always read data for the current and previous years without having to recode the actual year values each time you run.

CALL SYMPUT

Before we go on to macro definitions let us remember that the %LET statement assigns static (non-data step dependent) text to macro variables. However to put data from a DATA step into a macro variable you may use a macro function "SYMPUT". It looks and sounds complicated but it is really very simple. This function takes a data value from a DATA step and puts it into the GLOBAL or LOCAL macro symbol table. The syntax to get the *value of the variable* ***State*** from a DATA step into an amper-variable named *mystate* is as follows:

```
DATA one; SET two;
WHERE zipcode='12345';
CALL SYMPUT ('mystate',state);
RUN; ** required ;

PROC PRINT;
TITLE "Zip: 12345  State: &mystate";
```

The RUN statement is required in this case. This code will extract the state value for a particular zipcode (assuming zipcode and state are variables in the SAS dataset two) and place it in a GLOBAL variable for your later use. The evaluation will take place each time a record is read but only the last value will be maintained. Therefore, if there are several records in dataset one only the last occurrence of state in zipcode '12345' will be kept. The syntax of CALL SYMPUT looks slightly strange at first. Remember to use single quotes surrounding the macro variable name as ***'mystate'*** and place a variable name or expression as the second parameter. In the previous example the variable *state* was used. As you become more familiar with the macros and the CALL SYMPUT function you will use expressions and formats in place of simple variable names. The following expression will create a macro variable *&mydate* that can be used in TITLE statements or reports. The result is a formatted date that can enhance the look of output. This example uses the *&sysdate* automatic macro variable and reformats it since the automatic macro variable *&sysdate* is in the form 04OCT99. The following code will convert *&sysdate* to a longer more readable format:

```
data _null_;
CALL SYMPUT ('mydate',
TRIM(PUT("&sysdate"d,worddate22.)));
run;
%put &mydate;
```

Result: `October 4, 1999`

Observe the use of the expression "&sysdate"d and notice the use of double quotes. The expression will expand to "04OCT99"d when run which is then reformatted with the PUT() function which is described in the next section.

%PUT

Another statement that is quite simple but very useful is the %PUT statement. This statement is very helpful in debugging code. It may be used virtually anywhere and will write to the SAS Log any values of user defined or system defined macro variables such as:

```
%PUT "City: &city   State: &mystate" ;
%PUT "Day of Week: &sysday" ;
```

This %PUT statement will write to the SAS log:

City: Washington State: DC
Day of Week: Monday

Notice the use of an automatic system generated macro variable *&sysday*. Automatic system variables can always be used in code by preceding the name with an ampersand as in: *&sysdate* to get the current date. Several keywords may be used with the %PUT statement to write all or part of the GLOBAL symbol table. These are very helpful in debugging and testing macros within programs.

To write all global macro variables to the log:

```
%PUT _GLOBAL_;
```

To write all user defined global variables to the log:

```
%PUT _USER_;
```

To write all user defined local variables (those built inside macro definitions).

```
%PUT _LOCAL_;
```

Each SAS session maintains a series of automatic, system defined variable such as date and time and operating system. To write all system defined automatic variables to the log:

```
%PUT _AUTOMATIC_
```

To see the effect of these %PUT statements just place them anywhere in your code and submit. The evaluation of the macro will be written to the SAS log next to the name of the variable.

MACRO DEFINITIONS

Now that you are familiar with GLOBAL variables such as *&city, &sysday* (a system defined variable), and *&filemac* let us go on to developing macro definitions which can be used as subroutines in your program that can be executed at various places and under various conditions. The simplest form of the macro definition begins with %MACRO and ends with %MEND.

```
%MACRO firstmac;
   %PUT Hello World - ;
   %PUT This is my first 'Macro' ;
%MEND firstmac;
```

This macro can be executed or called in a program with the following line:

```
%firstmac;
```

And will produce the following message to the SAS log:

```
Hello World -
This is my first 'Macro'
```

Remember a macro definition begins with %MACRO *mac_name*; and ends with %MEND *mac_name*; The invocation of that macro somewhere else in the code is *%mac_name*. The macro must be defined before it can be invoked or executed in a program.

Note here that macro definition executions use the percent(%) in front of the name and macro variables use ampersands (&). For uncomplicated macros it is generally easier to use the amper-variable rather than the %MACRO definition however as you build more intricate macros the value of the definition style macro become more evident. Within the macro definition (between the %MACRO and the %MEND) you can place any valid SAS code statements or statement fragments. The macro name (*%mac_name*) will expand out when it is executed. Consider the following sample, which will create code shorthand for your programs:

```
%LET avars =  height age city ;
%MACRO bvars;
    score1 score2 midterm final
%MEND bvars;

DATA one;
SET two (KEEP = &avars %bvars);

PROC SUMMARY;
CLASS &avars;
VAR %bvars;
OUTPUT OUT = three sum=;

PROC PRINT DATA = three;
VAR &avars %bvars; run;
```

The code produced by the above macro statements is as follows:

```
DATA one;
SET two (KEEP = height age city
        score1 score2 midterm
        final);

PROC SUMMARY;
CLASS height age city;
VAR score1 score2 midterm final;
OUTPUT OUT = three sum=;

PROC PRINT data = three;
```

```
VAR  height age city
     score1 score2 midterm final;
RUN;
```

The above example shows that the %LET statement and the %MACRO statement are similar. Both are used to define a string of variables that will be used later in the program. Notice semicolons (;) are not used inside these macro definitions because we are trying to generate strings of variable names not full statements. The main difference between the two forms of macros(amper- or percent-) is the way they are invoked. When invoking the macro defined with the %LET statement you must use an ampersand (as in *&avars*) while the %MACRO-%MEND definition uses a percent sign (as in *%bvars*). You can place either version in various places inside the program to save time and assist in maintaining or modifying programs. In this case either form is essentially equivalent and suggests a type of programming style rather than a correct or incorrect method. It is generally advisable to use the %LET statement for simpler definitions and reserve the %MACRO-%MEND construct for more involved definitions. A good rule of thumb is to use the %LET statement for single variables and the %MACRO-%MEND definition for complicated or just long definitions. An easy way to build macros and remember the %MEND statement is to start the macro by coding:

```
%MACRO mymac ;
%MEND mymac ;
```

and then fill in the center later. In this manner you will never forget the %MEND. I always do the same for if-thens, and do-ends. To say that I never get the 'Unclosed DO blocks' error message would be lying but it does cut down on errors.

PARAMETER DRIVEN MACRO DEFINITIONS

The next step is to introduce parameters into the macro definition that will make them more flexible and pave the way for data driven selection of code. Parameters are values that are passed to the macro at the time of invocation. They are defined in a set of parentheses following the macro name. When referred to inside the definition they need preceding ampersands. There are two styles for coding the %MACRO parameters: positional and keyword. This next example shows the positional style:

```
%MACRO second (xx,yy,zz);
  %PUT Second Mac with &xx &yy &zz;

  PROC PRINT;
  VAR &xx &yy;
  FORMAT &xx  &zz.  &yy  5.0;
%MEND second;
```

To invoke the macro use the following syntax with the variables to be substituted in the proper position:

```
%second (age,height,3.0);
```

Notice that you can place variable names or values in the parameter list when calling the macro. When executed the *&xx*, *&yy*, and *&zz* are replaced with the variables and literals that are in the macro calling statement. When the program executes the processor sees the code:

```
%put Second Mac with age height 3.0;

PROC PRINT;
VAR age height;
SUM age height;
FORMAT age 3.0 height 5.0;
```

Note that the format statement evaluates correctly with the inclusion of a period(.) after the *&zz* in the code. You can invoke the macro again with:

```
%second (height,age,4.2);
```

to produce another section of code with different parameters. Remember that the macro definitions produce 'code'. They do not execute until they are invoked in the appropriate place in the program with the %MACRO-name invocation.

SCOPE

When the %LET statements are placed in open code (outside of any DATA step or %MACRO definition) the variables they define become GLOBAL. That is to say they are available to any and all statements that follow within that SAS session. When a %LET statement is found within a %MACRO definition and the same variable has not been previously defined the variable becomes LOCAL to that %MACRO definition and is not available outside of that macro unless it is defined with the %GLOBAL statement such as:

```
%GLOBAL x ;
```

If you define a macro variable within a %MACRO-%MEND definition with the %LOCAL statement that variable is local to that macro subroutine and will be placed in the LOCAL symbol table. Its value will not be passed outside of the definition subroutine to the main program. The exact hierarchy of global and local variables and how the buffers are maintained is complicated but be aware that if you defined a variable and then get the message: “Apparent symbolic reference not resolved” then the macro might have been defined locally and is not available in the GLOBAL symbol table at the time of invocation. Be sure the macro was spelled correctly. Be sure there

are run statements after any CALL SYMPUT's, and be sure to mention the macro in a %GLOBAL statement if it was defined inside a %MACRO definition.

EXPANDING MACRO DEFINITIONS

In the previous example you can see how parameters can be used to execute code differently in different places in your program. Now let us see how we can build macros that will create different code depending upon the data. For instance let's say you want to execute a PROC PRINT if there are records in the dataset but print an error message if there is an empty dataset. Or you might want to execute a frequency for one type of data or a means for another type of data. You can build if-then logic into the macro definition thereby selectively building code. Notice that in the following example several statements are outside the %IF-%THEN-%DO-%END logic and will execute for any value of *&type*. The correct syntax for IF-THEN logic within a macro requires a percent (%) before the following elements: (%IF, %THEN, %ELSE, %DO, %END). This tells the macro processor to interpret the %IF's as macro statements and conditionally generate program code.

```
%MACRO thirdmac ( type , title );
  %PUT Mac3 type=&type Title=&title;

  DATA three; SET one ;
  IF dat_type = &type;

  %IF &type = 1 %THEN %DO;
    PROC PRINT;
    TITLE "Report for: &title";
    %END;

  %ELSE %DO;
    PROC MEANS;
    TITLE "Sample Statistics for:
    &title";
  %END;

run;
%PUT Mac3 Now Finished ;
%MEND thirdmac;
```

To execute or invoke this macro you might use the following statement to invoke a PROC MEANS on the dataset three:

```
    %thirdmac(2,Type 2 Data) ;
```

The expanded code will look like this at execution time:

```
%PUT Mac3 type=2 Title=Type 2 Data;

DATA three; SET one ;
IF dat_type = 2;
PROC MEANS;
TITLE
  "Sample Statistics for: Type 2 Data";
run;
%PUT Mac3 Now Finished ;
```

PORTABILITY

So far we have discussed simple code and simple invocations of macros which may well have been performed without macros and may be just easier to replicate code instead of trying to be fancy. Now we will begin to look at situations where macros really make things easier. Let's say you want to develop code that will run in both a Unix and a Windows environment. No problem for the SAS System, you say, just remember to change the file names. Well when you have hundreds of files it becomes a problem. You can develop a series of macros that determine which operating system is active and build the file names accordingly. These macros are slightly tricky and require some testing but can save many hours of changing LIBNAMEs and FILENAMEs each time you run since the programs become truly portable. Here is a simplified example:

```
%MACRO nameit ;

  %LET path = mypath;
  %LET file = FILENAME;
  %IF &sysscp = WIN
    %THEN %LET DLM = \ ;
    %ELSE %LET DLM = / ;

  %IF &sysscp = WIN
    %THEN %LET DIR = c:\sas;
    %ELSE %LET DIR = m:/unix/direc ;

  %LET libnm = &dir.&dlm;
  %LET flnm  =
       &dir.&dlm.&path.&dlm.&file..dat;

FILENAME file1 "&flnm"; ** note double quotes;
LIBNAME lib1 "&libnm";

      %MEND nameit ;
```

The macro variable &sysscp is a SAS automatic macro variable that contains the name of the operating system. Notice the period(.) after the invocation of the *&dir* variable. This is necessary to indicate the end of the local variable *&dir* and the start of the next variable *&dlm*. If you actually want a period in the file name such as FILENAME.ext you must use two periods as in the *flnm* definition. When invoked the macro *%nameit* will produce the correct LIBNAME and FILENAME statements depending upon the operating system. The double quotes will enable the macro to evaluate properly in a SAS program, which requires FILENAMEs to be in quoted strings. When invoked with *%nameit* the following code results:

For Windows:

```
FILENAME file1
            "c:\sas\mypath\mydata.dat" ;
LIBNAME  lib1  "c:\sas\" ;
```

For UNIX:

```
FILENAME  file1
      "m:/unix/direc/mypath/mydata.dat" ;
LIBNAME   lib1 "m:/unix/direc/";
```

MACRO ARRAYS

As we have seen the %DO and %IF statements in the macro facility enable us to create long statements and code with little effort. The %DO statement has various forms (%DO-%WHILE, %DO-%UNTIL) and the iterative %DO as shown below:

```
%MACRO arrayme;
    %DO i = 1  %TO  5;
        file&i
    %END;
%MEND arrayme;

DATA one; SET %arrayme;
```

The macro evaluates to the following at execution time:

```
DATA one;
SET file1 file2 file3 file4 file5;
```

In this macro we generate a list of 5 file names. The values of 1 to 5 are substituted in the expression *file&i* to produce (file1 file2 file3 file4 file5). The above code will set five files into dataset one.

DOUBLE AMPERSANDS

In our previous example our files were named file1, file2, file3, file4, and file5. Let us suppose that our files are named for the state data that they contain. It will be necessary to build code that has amper variables in it. If we have 5 macro variables defined as follows:

```
%LET a1=KS;
%LET a2=MD;
%LET a3=CA;
%LET a4=NY;
%LET a5=KY;
```

We can generate code that will set all these files into one file as before but with the correct file names.

```
%MACRO stname;
    %DO i = 1  %TO  5 ;
        &&a&i
    %END;
%MEND stname;

      DATA one; SET %stname;
```

The first pass of the macro processor creates the variables &a1-&a5 and the second pass evaluates the amper-variables to their text strings (KS MD CA NY KY). We actually want to build code that looks like:

```
      &a1 &a2 &a3 &a4 &a5
```

The macro processor evaluates each occurrence of multiple ampersands on a separate pass so the double ampersands give the desired effect.

Consider this example of census data and its solution. We have a data file of several million records containing the names, information and zipcodes of residents a region of the US. We want to merge this with zipcode information. The zipcode information is divided into 50 files, one for each state. You guessed it the file names are the names of the states like AZ, KS, NY etc. These zipcode files are too big to set together and merge with the regional resident file so we use macros to select only those state zipcode files of the states that are contained in the regional residents file. Of course there are many approaches but this is just one using macros.

```
DATA _NULL_;
   SET region(KEEP=state) END=eof;
   LENGTH st_mac $ 150 ;
   RETAIN st_mac ' ' num_st 0 ;
   IF INDEX(st_mac,state)=0 THEN DO;
      num_st=num_st+1;
      st_mac=trim(st_mac)||' '||state;
   END;
   IF eof THEN
      CALL SYMPUT ('st_mac',st_mac);
RUN;

 ** now select only state zipcode files
we need **;

   DATA regzip;
      SET &st_mac ;
   PROC SORT;
      BY zip;
   DATA region;
      MERGE region regzip;
      BY zip;
   RUN;
```

The DATA _NULL_ step builds a string of state file names with each new state it reads from the region dataset. It checks with the index function to be sure it has not already added that state to the string. The amper-variable *&st_mac* becomes a string of unique state names like (AK KS CA NY) depending upon which states are in the regional residents file. You must set the initial length to 150 to leave enough room for 50 states. Also you must trim the result before added a new state otherwise the string will be greater

than 150. In this manner we can build macro strings based upon the data without knowing ahead of time what states are in the dataset. As we combine these various techniques we can build programs that generate code based upon the incoming data.

STORING MACRO DEFINITIONS

Normally when you submit a program containing a macro definition the macro processor compiles and stores the executable version of the macro in a catalog in the WORK library. These "session compiled macros" will not be saved when the session ends. There are several ways, however, that you may save macros for reuse at a later time. The simplest way would be to save the macro in a library and %INCLUDE that saved macro when needed. A slightly more sophisticated approach is to use the "autocall library". On directory based systems like Unix and Windows you can build autocall library by creating a separate subdirectory and save definition style macro source code in this directory. Each macro (as defined by the standard %MACRO-MEND construct) should be stored in an individual and separate file named the same as the name of the macro. The steps are as follows:

1. Build and test a macro named *macname.*
2. Insure that all variables are created with the appropriate scope so that variables designed outside of the macro are not unknowingly changed.
3. Save *macname* to subdirectory *c:\sas\mymacs* (or any valid subdirectory on your system). Macros can also be saved in SAS catalogs as source entries.
4. When needed in a subsequent program define the autocall library and turn on the autocall facility with:

```
OPTIONS MAUTOSOURCE
SASAUTOS='c:\sas\mymacs';

** for catalog entries use this *;
LIBNAME mymacs 'c:\sas\mymacs';
FILENAME mimac catalog
'mymacs.macname';
OPTIONS MAUTOSOURCE SASAUTOS=mimac;
```

5. Call the macro with:

```
%macname
```

6. Subsequent Macros in the library may be called throughout the session.

Another way to save macros for future use is with the *stored compiled macro* facility. This method stores only a compiled version of the macro. The macro is not recompiled at execution time and can be more efficient for production systems. The exact syntax may vary from platform to platform but looks similar to this windows example:

```
%MACRO macname / store ;
   %put Hello World ;
%MEND;
```

Invocation is accomplished with:

```
LIBNAME lib1 'c:\sas\compiled';
OPTIONS MSTORED SASMSTORE=lib1;
```

The stored compiled macro facility should only be used for well tested macros that will be used over and over on a regular basis. The source code must be stored separately and can not be recovered from the compiled version.

DEBUGGING TECHNIQUES

Debugging a macro is about the same as debugging regular SAS code. Many times the error can be masked or compounded. Here are a few tips:

- Make sure statements match such as %MACRO-%MEND, %DO-%END, %IF-%THEN-%ELSE.
- CALL SYMPUTS will not execute until a RUN or another step is encountered.
- Be careful about balancing single and double quotes.
- Use %LOCAL and %GLOBAL when clarification of scope is necessary.
- Use the SYMBOLGEN and MACROGEN options to trace what the macro is doing.
- Use %PUT statements during testing including %PUT _USER_ ;
- Be careful to spell all keywords including macronames correctly.

CONCLUSION

We have seen how you can use the SAS Macro Facility to enhance your programs. Starting with the %LET statement and advancing to macro definitions we have learned that macros enable you to build complex and repetitive code with ease. Macro variables and macro definitions must be defined before they can be invoked. Macros are essentially strings of data saved in a buffer available to every subsequent data step or procedure in the current SAS session (or job step). The execution of macros in code is simply a replacement of the code generated by the macro definition at the point where it is invoked.

The use of macros in SAS code can streamline your programs and enable you to modularize your code. It will assist you in developing parameter driven and data driven programs that will require less maintenance. I hope this short presentation will assist you in developing elaborate and convenient reusable code. The use of macros should only be limited by your imagination.

FOOTNOTES

1. p29: SAS Institute Inc., SAS® Guide to Macro Processing, Version 6. Second Edition, Cary, NC: SAS Institute Inc., 1990

REFERENCES

SAS Institute Inc., SAS® Guide to Macro Processing, Version 6. Second Edition, Cary, NC: SAS Institute Inc., 1990

SAS Institute Inc., SAS® Macro Language: Reference, First Edition, Cary, NC: SAS Institute Inc., 1997

Michael G. Sadof
MGS Associates, Inc.
A SAS® Quality Partner
Bedford, NH
mgs@mgsnet.net

SAS is a registered trademark or a trademark of the SAS Institute Inc. in the USA and other countries. ® indicates USA registration.

Other brand and product names are registered trademarks or trademarks of their respective companies.