THE SAS® MACRO FACILITY

Maria Nicholson, ORI, Inc.
Bob Pulgino, ORI, Inc.

1. Introduction

1.1. Abstract

One of the most common questions asked by SAS programmers who are just beginning their introduction to the macro facility is: "Why use macro... what's it for?". This tutorial attempts to demonstrate the advantages that macro can offer and the motivations that a SAS programmer who uses macro will have. Starting with an example SAS program that uses no macro, features are added which demonstrate the common uses of macro and the exceptional power it can give the SAS programmer.

1.2. What is the Macro Facility?

To understand how the macro facility works and how to best use it, you first must completely understand what it is and how it fits into a conceptual model of the SAS System as a whole.

The macro facility can best be thought of as a separate language that works with the standard SAS language. It contains many of the same type of programming statements and functions as the SAS DATA-step language. While the standard SAS language provides instructions on how to manipulate and process data in the form of SAS datasets, the macro facility provides instructions for processing the *text* of your SAS program itself. It treats the text in a program file that make up other SAS statements as its data, and can perform many of the same operations on this text that DATA-step code can perform on the contents of SAS datasets.

This is often the most difficult concept for new macro programmers to grasp, but it is the single fundamental concept that must be understood in order to use macro effectively. The diagram in Figure 1 shows a model of how a SAS program without any macro code is processed. The lines of code in the program are stored in a temporary region called the Input Stack. The SAS processor reads the text from the input stack, one word at a time, until it has a complete SAS DATA or PROC step. The programming statements defining this step are checked for errors, and if none are found, the step is compiled and executed. This process is repeated for all the DATA and PROC steps in the program.

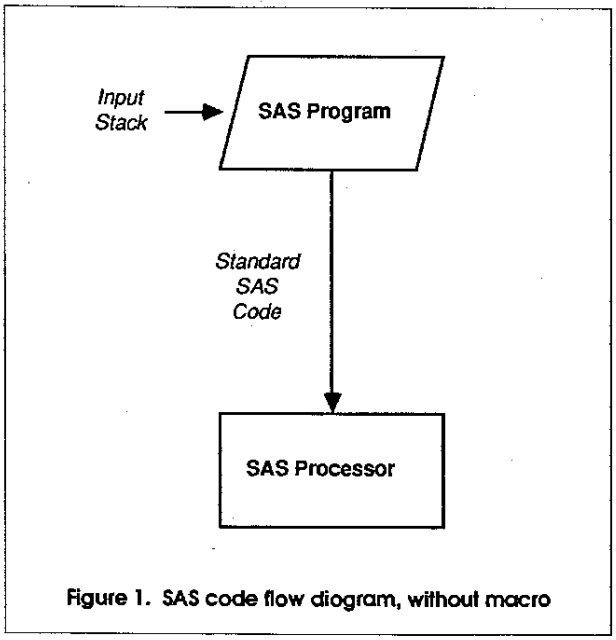

Figure 1. SAS code flow diagram, without macro

When the SAS program contains macro variables or programming statements, however, the program code is processed differently, as shown in Figure 2. As the program is read from the input stack, macro statements will not be sent to the SAS processor, but to the macro processor instead. Here the macro statements are evaluated, any logic defined by the macro's programming language will be executed, and the results of the evaluations — program text — will be reprocessed as if it were part of the original SAS program. In other words, the macro code in your program is used to *generate* SAS code that will be *substituted* in place of the macro code before the SAS processor sees it.

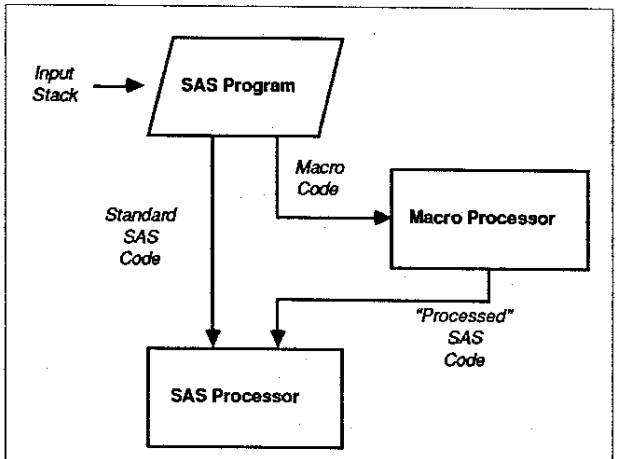

Figure 2. SAS code flow diagram with macro

1.3. Macro Programming Techniques

In this paper we will study the macro facility by example; we will take a simple program and apply catagorical macro programming techniques so that the example can be adapted to fit various changing requirements.

Keep in mind as you evaluate the specific advantages & capabilities offered by each adaptation, that very few real SAS programs will be this simple and small. Examine each coding example shown and picture its benefits in terms of the relative scope of your applications.

Also, keep in mind that we are limited to briefly discussing a very complex and sophisticated topic. The purpose of the paper is to bring a general introduction of this topic to an audience that has little or no experience with it. Use these examples as a foundation upon which to build your understanding of the macro facility and its many uses.

2. Symbolic Substitution

2.1. Using Symbolic Substitution

Symbolic substitution is a term used to describe the process of using macros or macro variables as "symbols" that mark parts of your program that you wish to change. Rather than using a text editor to locate these symbols and change them manually, the macro facility can do the substitution when the program is run.

The re-written version of our REPORTS macro, which uses a parameter instead of a regular macro variable for symbolically referencing the input data set name (DATASET) is shown in Figure 12.

```
%MACRO REPORTS(DATASET = );

  PROC PRINT DATA = &DATASET SPLIT = "*" DOUBLE;
    BY SALES_REP;
    SUM SALES PROFIT COMMISSION;
    SUMBY SALES_REP;
    TITLE "Sales Listing Report for Dataset: &DATASET";
  RUN;

  PROC FREQ DATA = &DATASET;
    TABLES SALES_REP * SALES;
    TITLE "Sales Frequency Report for Dataset:&DATASET";
  RUN;

%MEND REPORTS;

%REPORTS(DATASET = SALES.QTR1)

%REPORTS(DATASET = SALES.QTR2)
```

Figure 12. Using a parameter with the reporting macro

This form of macro parameter definition is referred to as a *keyword* parameter, since the name of the parameter is used as a keyword when the macro is called to assign it a value. An alternative to the keyword form is the *positional* parameter definition, which does not use the parameter name when the macro is called. For more information on the differences between these types of macro parameters, see reference 1.

3. Conditional Substitution

3.1. What is Conditional Substitution Logic?

Conditional substitution is the term used to describe the technique of using macro logic to determine if a specific segment of code is to be substituted into your program, based on the value of a macro variable or parameter, or some other logical test. This is the first example of a technique that uses the macro programming statements as a part of the body of your macro. Just like the DATA step language allows you to use the IF/THEN/ELSE statement structure to conditionally execute sections of code in a DATA step, the macro language allows you to use %IF/%THEN/%ELSE to conditionally substitute text in the body of a macro.

The general syntax of the macro's %IF/%THEN/%ELSE statement structure is shown in Figure 13. As you can see, it is very similar to the DATA step language version; the %IF keyword is followed by a boolean expression that should evaluate to a value of true or false. If the expression evaluates to true, the macro statement or text to be substituted following the %THEN keyword is executed or substituted. If the %ELSE statement is used, and the boolean expression on the %IF statement is false, the macro statement or text to be substituted following the %ELSE keyword is executed or substituted. If you wish to follow the %THEN or %ELSE with multiple macro statements or text containing semi-colons (or a combination), you can use the %DO/%END structure to contain them, just as with the DATA step language.

Syntax:

```
%IF (expression) %THEN macro-statement or text ;

%ELSE macro-statement or text ;
```

Figure 13. %IF statement syntax

3.2. A Multiple-Choice Reporting Program

Lets assume that our REPORTS macro program has been working fine — each time it is called in a program, it dutifully generates the code for printing both reports for the dataset specified with the DATASET parameter, and all our non-programming users have become very comfortable with its use. Several of our users have mentioned, however, that they often run the program and get both reports, even at times when they only really want to see one of them. How can we modify our REPORTS macro so that the users can indicate which report of the two they want to turn "off", or prevent from being generated by the macro, when they use it? Figure 14 shows one way of rewriting our macro that uses %IF/%THEN to control whether the code for each of the two reports will be substituted. Two new parameters (REPORT1 and REPORT2) are defined to allow the user to set the values that are tested by the %IF/%THEN statements.

```
%MACRO REPORTS(DATASET=, REPORT1=YES, REPORT2=YES);

  %IF &REPORT1 = YES %THEN %DO;
    PROC PRINT DATA = &DATASET SPLIT = "*" DOUBLE;
      BY SALES_REP;
      SUM SALES PROFIT COMMISSION;
      SUMBY SALES_REP;
      TITLE
      "Sales Listing Report for Dataset: &DATASET";
    RUN;
  %END;

  %IF &REPORT2 = YES %THEN %DO;
    PROC FREQ DATA = &DATASET;
      TABLES SALES_REP * SALES;
      TITLE
      "Sales Frequency Report for Dataset: &DATASET";
    RUN;
  %END;

%MEND REPORTS;

%REPORTS(DATASET=SALES.QTR1, REPORT1=NO);
```

Figure 14. Using conditional execution

Each of the two reporting steps in the macro have been enclosed in a %DO/%END structure preceded by an %IF/%THEN control statement. The text for each step will only be substituted when the macro is called if the conditional expression on the %IF statement is true.

The conditional expression for each %IF statement is a test to see if the associated parameter (REPORT1 or REPORT2) has a value of YES. Since the two parameters are defined with a default value of YES, all the user has to do to "turn off" one of the reports for a given call of the macro is override this default value and assign the parameter some other value such as NO.

At the bottom of Figure 14, we see such a case where the user wishes to see only the second report of the two. Therefore, when the macro is called, the REPORT1 parameter is assigned the value NO. This results in the conditional expression on the first %IF statement to be false, and therefore the code for the first report is never generated by the macro and substituted into the program.

```
PROC FREQ DATA = SALES.QTR1;
  TABLES SALES_REP * SALES;
  TITLE "Sales Frequency Report for Dataset:SALES.QTR1";
RUN;
```

Figure 15. Resolution of the macro call

Syntax:

```
%LET  macrovar  =  string ;

 or

%LET  macrovar  =  %EVAL( expression ) ;
```

Figure 7. %LET statement syntax

Figure 7 shows the syntax rules for the %LET statement in two forms. The first form is used when the value we wish to assign is a simple character string. The string can include other macro variable references (symbolic substitution can be used on macro statements as well as standard SAS statements) and one or more of the special macro functions used to enable powerful manipulation of strings, in the same way DATA-step string functions are used (these macro functions are documented in references 1 and 2).

One very important macro function is shown in the second form of the %LET statement syntax: the %EVAL function. Recall from our earlier discussion the important distinction of the macro facility: it is designed to process text strings as its data. Numeric expressions are usually not the concern of the macro processor, and it does not bother to look for and recognize them. The %EVAL function is used to point out a numeric expression that is to be resolved, and replaced in the assignment statement with its evaluated result.

Let's try to clarify all this using the examples shown in Figure 8.

	STATEMENT	*VALUE ASSIGNED TO X*
[1]	`%LET X = Hello there!;`	Hello there!
[2]	`%LET X = 4+2*5;`	4+2*5
[3]	`%LET X = %EVAL(4+2*5);`	14
[4]	`%LET Y = 4;` `%LET X = &Y+2*5;`	4+2*5
[5]	`%LET X = %EVAL(&Y+2*5);`	14

Figure 8. Macro variable assignment examples.

In example [1], we see a simple case of assigning the string "Hello there!" to the macro variable named "X". The macro facility does not need any quoting for the string constant since it only works with text strings — it knows that the expression starts following the equal-sign, and ends immediately before the semi-colon; that variables in the expression will begin with an ampersand, and that macros and macro-functions such as %EVAL will begin with a percent-sign. Everything else in the string expression must be treated as constant text, so the variable "X" receives its value as shown.

In example [2], we see what at first appears to be a numeric expression, and one without the benefit of our discussion to this point might think that "X" will be assigned the value "14." As we now know, however, macro doesn't recognize this string as a numeric expression; instead, it will dutifully assign the string "4+2*5" to X. In order to force the macro facility to evaluate the numeric expression and assign the result to X, we must use the %EVAL function, as shown in example [3].

Example [4] demonstrates the use of symbolic substitution in a macro assignment — using the value of one macro variable to determine the value of another. Here, we've assigned the string "4" to a macro variable named "Y". In the assignment for X, we reference the value of Y by including it in the expression with a preceding ampersand (See if you can determine what the value of X would be if we used Y without the ampersand). Again, since we did not use the %EVAL function, the unevaluated string is assigned to X. In order to use the value of Y as part of a numeric expression whose result would be assigned to X, we need to use the %EVAL function as shown in example [5].

2.2. Defining Macros

2.2.1. What is a Macro?

In its simplest form a macro is, much like a macro variable, a way of assigning a name to a piece of text in your SAS program. Once a macro has been defined (analogous to assigning a value to a macro variable), they can be referenced symbolically in your SAS program to substitute their text values in place in your program.

Unlike macro variables, however, macros themselves can use programming logic to modify the strings they refer to. We will see examples of how they can use the values of macro variables in the same way that a DATA step uses dataset-variable values to change the result of the functions they perform.

2.2.2. The %MACRO/%MEND Statement

Macros are defined by using the %MACRO and %MEND statements. The syntax for using these statements is shown in Figure 9.

Syntax:

```
%MACRO macroname (parameter definitions) ;

   text to be substituted and/or
   macro programming statements.

%MEND  macroname ;
```

Figure 9. %MACRO statement syntax

The definition of the macro always begins with the %MACRO statement, which is used to declare the name of the macro that we will use in our program to reference its contents. The %MACRO statement can optionally also be used to define one or more macro parameters, special macro variables that are used when the macro is referenced (we will discuss macro parameters in more detail shortly).

Following the %MACRO statement in the macro definition is what we will call the "body" of the macro. The macro body can be comprised of text that we want to substitute into our SAS program when we reference, or "call" the macro, macro programming statements that will perform logical processing on text, macro variables, or other macros, or a combination of the two. Examples of macro definitions that should clarify this will be presented.

The text that could make up the body of a macro could be parts of SAS keywords, complete keywords, parts of statements, complete statements, parts of DATA or PROC steps, or complete DATA and PROC steps.

Finally, the end of the macro definition is marked by the %MEND statement. This statement should include the name of the macro, exactly as it is spelled on the corresponding %MACRO statement that started the macro definition (the SAS System allows you to omit the macro name on the %MEND, but good programming style demands that it be included).

2.2.3. Using Macros for Symbolic Substitution

Why would one want to use macro for symbolic substitution when we have seen how nicely macro variables work for this purpose? First of all, macro variables cannot make use of the macro programming statements we mentioned. But even if we have no need for logical processing of the text, the macro definition provides us with a cleaner and more readable way to define a symbolic name for a large block of code that makes up a significant portion of our program, and that may require symbolic substitution within itself in order to be useful.

As an example, let's return to our simple reporting program, and ponder a common problem. Our use of macro variables for symbolic substitution is a fine solution to our need to run the program against different datasets each time. What if the need arose to run the same reports against different datasets at the same time? In order to accomplish this with our program as it is currently written, we would have to either run a separate job for each input dataset, or to duplicate the entire block of code within the program file, once for each dataset.

Using macro, however, we can let the SAS System do the duplication for us. By defining a macro that contains the code for generating the reports as its body, we could symbolically substitute as many copies of this code into our program as we needed, simply by referencing the macro by name. Figure 10 shows how we would rewrite our program as a macro, and how we would then generate the report for two datasets.

```
%MACRO REPORTS;

  PROC PRINT DATA = &DATASET SPLIT = "*" DOUBLE;
    BY SALES_REP;
    SUM SALES PROFIT COMMISSION;
    SUMBY SALES_REP;
    TITLE "Sales Listing Report for Dataset:&DATASET";
  RUN;

  PROC FREQ DATA = &DATASET;
    TABLES SALES_REP * SALES;
    TITLE "Sales Frequency Report for Dataset:&DATASET";
  RUN;

%MEND REPORTS;

%LET DATASET = SALES.QTR1;
%REPORTS

%LET DATASET = SALES.QTR2;
%REPORTS
```

Figure 10. A macro definition for the reporting program

Notice that the body of the macro (we chose the name REPORTS) contains the complete text of our original program, except the %LET statement. Once this code is read, starting at the %MACRO statement through the %MEND statement, the macro processor knows that whenever the symbolic reference %REPORTS is encountered in the program, the text in the body of this macro is to be substituted.

Following the macro definition, we see the familiar %LET statement we use to define the name of the dataset that we wish to use as input for our reporting step. Then we see the symbolic reference to the REPORTS macro — the macro processor will read this statement, and substitute all the text from the body of the REPORTS macro into the top of the input stack. As this text is read, each occurance of the symbolic reference to the DATASET macro variable will be replaced by the variable's value, just as before. So these two lines of code are replaced by a complete copy of our reporting program, with the dataset name SALES.QTR1 inserted at the appropriate places. The last two lines of code repeat this process for a different value of DATASET — the name of the second dataset we wish reports for. The complete program resulting from macro processing, as seen by the standard SAS processor, is shown in Figure 11.

```
PROC PRINT DATA = SALES.QTR1 SPLIT = "*" DOUBLE;
  BY SALES_REP;
  SUM SALES PROFIT COMMISSION;
  SUMBY SALES_REP;
  TITLE "Sales Listing Report for Dataset:SALES.QTR1";
RUN;

PROC FREQ DATA = SALES.QTR1;
  TABLES SALES_REP * SALES;
  TITLE "Sales Frequency Report for Dataset:SALES.QTR1";
RUN;


PROC PRINT DATA = SALES.QTR2 SPLIT = "*" DOUBLE;
  BY SALES_REP;
  SUM SALES PROFIT COMMISSION;
  SUMBY SALES_REP;
  TITLE "Sales Listing Report for Dataset: SALES.QTR2";
RUN;

PROC FREQ DATA = SALES.QTR2;
  TABLES SALES_REP * SALES;
  TITLE "Sales Frequency Report for Dataset:SALES.QTR2";
RUN;
```

Figure 11. The reporting macro resolved

2.2.4. Macro Parameters

Our example program can be simplified further through the use of macro parameters. As mentioned above, macro parameters are special macro variables that are intended to be used with a specific macro, are defined at the same time as the macro, and are assigned a value when the macro is called. In the example reporting program, you can see that the macro variable DATASET is used specifically by the REPORTS macro; its value is assigned immediately before the macro is called, and its value is only needed in the body of the macro. Under these conditions, you should define the macro variable as a parameter.

There are several reasons why using parameters is usually preferable to using regular macro variables, most of which are beyond the scope of this paper. The most obvious and relevant to our example is the way it simplifies the use of the macro and makes the programming easier to read.

To create a macro parameter, we simply add its definition to the %MACRO statement. Following the name of the macro (but before the semi-colon), we add a pair of parentheses. Within the parentheses, we give the name of the parameter, followed by an equal-sign. This can optionally be followed by a default value, one that we wish to be assigned to the parameter whenever the macro is called in a program and an explicit value is not given.

A common use for this technique is to enable a given program to be run using a different dataset as its source of input on different occasions. The example in Fignre 3 defines an example of a two-step reporting program that could make use of the symbolic substitution technique.

```
PROC PRINT DATA = SALES.QTR1 SPLIT = "*" DOUBLE;
  BY SALES_REP;
  SUM SALES PROFIT COMMISSION;
  SUMBY SALES_REP;
  TITLE "Sales Listing Report for Dataset:SALES.QTR1";
RUN;

PROC FREQ DATA = SALES.QTR1;
  TABLES SALES_REP * SALES;
  TITLE "Sales Frequency Report for Dataset:SALES.QTR1";
RUN;
```

Figure 3. Simple reporting program

In order to change this program so that a dataset other than SALES.QTR1 will be used as the basis of the reports, a programmer would have to search the program to locate all four occurances of the dataset name and recode them to refer to the name of the new dataset. Not a big chore for this simple example, but in a larger, more complicated program, it could be a very time-consuming and error-prone task. By changing the program to take advantage of the symbolic-substitution capabilities offered by macro, we can reduce the number of places in the code that the dataset name needs to be changed to one.

Changing a SAS program to use symbolic substitution requires three steps. First, we must locate all the explicit references to the code that will change from run to run in the program. We have already done this by showing the dataset name references in bold-face in Figure 3.

Second, we need to choose a "symbol" (macro variable) name that will be used to indirectly reference this code in the revised program, and substitute this name, preceded by an ampersand ("&") for each occurance of the explicit references. This revision to our reporting example is shown in Figure 4 (we chose the name "DATASET" to use as our symbol).

```
PROC PRINT DATA = &DATASET SPLIT = "*" DOUBLE;
   BY SALES_REP;
   SUM SALES PROFIT COMMISSION;
   SUMBY SALES_REP;
   TITLE "Sales Listing Report for Dataset: &DATASET";
RUN;

PROC FREQ DATA = &DATASET;
   TABLES SALES_REP * SALES;
   TITLE "Sales Frequency Report for Dataset: &DATASET";
RUN;
```

Figure 4. Using a macro-variable symbol

The final step in changing our program to use symbolic substitution is to devise a mechanism for assigning a value to our macro-variable "symbol" — a means of telling the macro facility that we want the name of the dataset that we are interested in using for the reports substituted in place of each occurance of "&DATASET" in the program before the standard SAS processor sees the program. There are many ways to approach this, depending on the ways in which we choose to use the many other features of the macro facility (all of the macro programming techniques we will be discussing in this paper use macro-variable values in one way or another).

The simplest way to assign a value to a macro variable is to use the macro facility's assignment statement, %LET. This statement assigns values to a macro variable by simply following the %LET keyword with the name of the macro variable, an equals-sign ("="), and the string value to be assigned. Using this approach, our completed example is shown in Figure 5.

```
%LET DATASET = SALES.QTR1;

PROC PRINT DATA = &DATASET SPLIT = "*" DOUBLE;
  BY SALES_REP;
  SUM SALES PROFIT COMMISSION;
  SUMBY SALES_REP;
  TITLE "Sales Listing Report for Dataset:&DATASET";
RUN;

PROC FREQ DATA = &DATASET;
  TABLES SALES_REP * SALES;
  TITLE "Sales Frequency Report for Dataset:&DATASET";
RUN;
```

Figure 5. Assigning a value to the symbol

Let's stop and study what will happen when this job is submitted and processed by the macro and SAS processors that make up the SAS System. First, the entire program is stored in the memory region we referred to as the input stack. Each word of code will be read from this region in turn and processed by the appropriate processor. When the first word is read, "%LET", the SAS System is able to recognize it as a keyword for a statement that the macro processor is supposed to handle, so the statement is sent there. The macro processor then executes the statement, making a note that the symbol "&DATASET" should be replaced with the value "SALES.QTR1".

As the next line of code is read, each standard SAS keyword is sent to the standard SAS processor for handling. When the symbol "&DATASET" is encountered, however, the SAS System recognizes it as a word that the macro processor must handle, because it starts with an ampersand. The macro processor, reviewing its note that the symbol "&DATASET" is to be replaced with the value "SALES.QTR1", generates this string and sends it back to the input stack, so that "SALES.QTR1" will be the next word read by the SAS System for processing. When this word is read, because it does not begin with an ampersand (&) or percent-sign (%), it will be sent to the standard SAS processor, where it will be considered a part of the PROC PRINT statement. Thus, the value of our symbol "&DATASET" has been substituted in place of its name in the statement.

This process of SAS code processing and symbolic substitution continues for the rest of our program. The completed program, as the standard SAS processor sees it, is shown in Figure 6.

```
PROC PRINT DATA = SALES.QTR1 SPLIT = "*" DOUBLE;
  BY SALES_REP;
  SUM SALES PROFIT COMMISSION;
  SUMBY SALES_REP;
  TITLE "Sales Listing Report for Dataset:SALES.QTR1";
RUN;

PROC FREQ DATA = SALES.QTR1;
  TABLES SALES_REP * SALES;
  TITLE "Sales Frequency Report for Dataset:SALES.QTR1";
RUN;
```

Figure 6. Results of the symbolic substitution

4. Iterative Substitution

4.1. What is Iterative Substitution Logic?

Iterative substitution is the term used to describe the technique of using macro logic to repeat a segment of SAS code in your program, so that when combined with symbolic substitution, a similar process can be performed on many data entities or with changing parameters without unnecessary redundencies in your program. This technique usually makes use of the %DO/%END statement structure in one of its many forms. Figure 16 shows the syntax of the three forms that this statement can have in the macro facility language.

Syntax:

```
%DO %WHILE ( expression );

   text to be iteratively-substituted

%END;

or

%DO %UNTIL ( expression );

   text to be iteratively-substituted

%END;

or

%DO macro-var = start-value %TO end-value
    [%BY increment];

   text to be iteratively-substituted

%END;
```

Figure 16. Iterative %DO/%END statement syntax

All three of these syntax forms have equivalent forms with the DO/END statement structure of the DATA-step language. The first form uses a "while" clause to control the number of times the text in the body of the structure is to be repeated. This clause contains a boolean expression that is evaluated at the start of each repetition, and if the expression evaluates to "true", the text in the body will be substituted and repetition will continue. If it evaluates to a "false" however, the body of the structure will not be substituted, and the looping will stop. The manner in which the code reads — "DO the following program text WHILE this condition is true" — portrays its functioning very well.

The second form is very similar to the first, in that it also uses a boolean expression to control the number of times the structure will repeat itself in the program. The "until" clause specifies that looping should continue only if the expression is NOT true — "DO the following program text UNTIL this condition is true". Another subtle difference between the WHILE and UNTIL forms is that the WHILE loop tests its condition at the *start* of each iteration, and the UNTIL loop tests at the *end*. What this means is that if the expression is false at the start of the first cycle of a WHILE loop, the loop's contents will never be substituted. An UNTIL loop will always be executed at least once, however, since its test is performed *after* the loop is performed.

The third form of the %DO/%END structure uses a macro variable as a counter, to perform the repetition as the variable takes on a set of ordinal, discretely-spaced values. The values used for the *start-value, end-value,* and the optional *increment* (if the %BY *increment* component of the statement is not used, %BY 1 is assumed), are all used to determine the number of times the loop will repeat and what the value of *macro-var* will be during each cycle. The first time through the loop, *macro-var* will have the value of *start-value*. At the start of each successive cycle of the loop, the value of *macro-var* will be increased by the amount of *increment*. If the new value of *macro-var* is greater than *end-value*, the loop will not be repeated again.

4.2. A Year-End Reporting Program

How can we make use of this iterative substitution technique to improve our reporting program example? Let's assume that our users have been taking advantage of the %REPORTS macro for some time and are very happy with the convenience it provides, but that the end of the fiscal year for our company is almost upon us. We can anticipate the need to generate reports for the entire year based on the final, updated contents of each of the quarterly-sales datasets. There will still be times when we will want to limit the scope of the report request to one report format or the other. Can we use macro to make this task easier, to eliminate the need to repeat calls to our %REPORTS macro, and to make the same parameter changes to each one in the program? Figure 17 shows how we would have to code a request for the PROC PRINT using the macro we have defined so far.

```
%REPORTS(DATASET=SALES.QTR1, REPORT2=NO)

%REPORTS(DATASET=SALES.QTR2, REPORT2=NO)

%REPORTS(DATASET=SALES.QTR3, REPORT2=NO)

%REPORTS(DATASET=SALES.QTR4, REPORT2=NO)
```

Figure 17. Multiple reports without iterative substitution

As you can see, each line of code is exactly the same except for the dataset-name reference, and that is only different by the last character, a number that steps from the value 1 to the value 4 with each line of code. Since we just discussed a form of the %DO/%END structure that will enable us to repeat code as we increment a macro-variable from some starting value to an ending value, it may occur to you that we could use this structure to generate the repetitive code.

Figure 18 shows a macro that can do just that for us. The %ALLQTRS macro is defined with the REPORT1 and REPORT2 parameters just as our previous macro was, so that we can "turn off" one report or the other for all of the quarterly data as a group. It uses the %DO statement to define a loop that will repeat its contents in our SAS program as the macro-variable INDEX steps in value from 1 to 4. The contents of this loop is our familiar %REPORTS macro, which will itself be processed by the macro, and generate its defined SAS code, for each cycle of the %DO loop. Each time %REPORTS is processed, its parameters REPORT1 and REPORT2 will take on the same value of the %ALLQTRS' parameters that happen to have the same name, since we are using symbolic substitution on the %REPORTS call to assign these values.

```
%MACRO ALLQTRS(REPORT1=NO, REPORT2=NO);

  %DO INDEX = 1 %TO 4;

    %REPORTS(DATASET=SALES.QTR&INDEX,
             REPORT1=&REPORT1, REPORT2=&REPORT2)

  %END;

%MEND ALLQTRS;
```

Figure 18. Macro to generate reports for all quarters

Look very closely at the assignment of the DATASET parameter on the %REPORTS call in the %DO loop. Part of the value assigned to this parameter is constant text ("SALES.QTR"). Also part of the assigned value, however, is a symbolic substitution of our %DO loop's macro-variable INDEX. When the loop is on its first cycle, INDEX has the value 1, so the DATASET parameter will have the value SALES.QTR, followed immediately by this value of 1, or SALES.QTR1, the name of our first-quarter sales dataset. The second time through, INDEX has the value 2, so DATASET will be assigned the value SALES.QTR2, and so on.

By combining the iterative substitution technique with the others such as symbolic substitution, redundant processing can be reduced to a minimum number of lines of code in your program file. Since the "common" code is not repeated in your source file, changes that will inevitably have to be made only need to be made once.

5. Summary

Effective use of the SAS macro facility requires a basic understanding of its intended purpose and how it works in conjunction with the rest of the SAS System. The macro facility provides a separate language that is used to create and modify standard SAS programs by symbolically, conditionally, or iteratively substituting text within the body of the program when the macros or macro variables are referenced. Proper use of this capability will enable you to write program systems that are smaller, more readable, and reuseable in similar applications. The benefits that these characteristics provide will be directly measurable in the time you save in maintaining and rewriting your programs as the users needs change.

The authors can be contacted at:

ORI, Inc.
601 Indiana Avenue, NW, Suite 1000
Washington, DC 20004
(202) 737-2666

6. References

1. SAS Institute, Inc., SAS Users' Guide: Basics, Version 5 Edition, Cary, NC: SAS Institute, Inc., 1985.

2. SAS Institute, Inc., SAS Guide to Macro Processing, Version 5 Edition, Cary, NC: SAS Institute, Inc., 1986.

3. Phillips, Jeff, "The SAS Macro Facility," Proceedings of the Twelfth SAS Users' Group International Conference, Cary, NC: SAS Institute, Inc., 1987.

4. Phillips, Jeff, "The Use of the Macro Facility in a Systems Development Environment," Proceedings of the Eleventh SAS Users' Group International Conference, Cary, NC: SAS Institute, Inc., 1986.

5. Merlin, Ross, "Sample Utility Macros," Proceedings of the Eleventh SAS Users' Group International Conference, Cary, NC: SAS Institute, Inc., 1986.

6. Pulgino, Robert, "User Interface Tools and the Design of Interactive Systems," Proceedings of the Twelfth SAS Users' Group International Conference, Cary, NC: SAS Institute, Inc., 1987.