Paper IS01
An Animated Guide: The Map of the SAS® Macro Facility
By Russell Lavery
Thanks to Ian Whitlock, Saad Anbari and Musa Nsereko

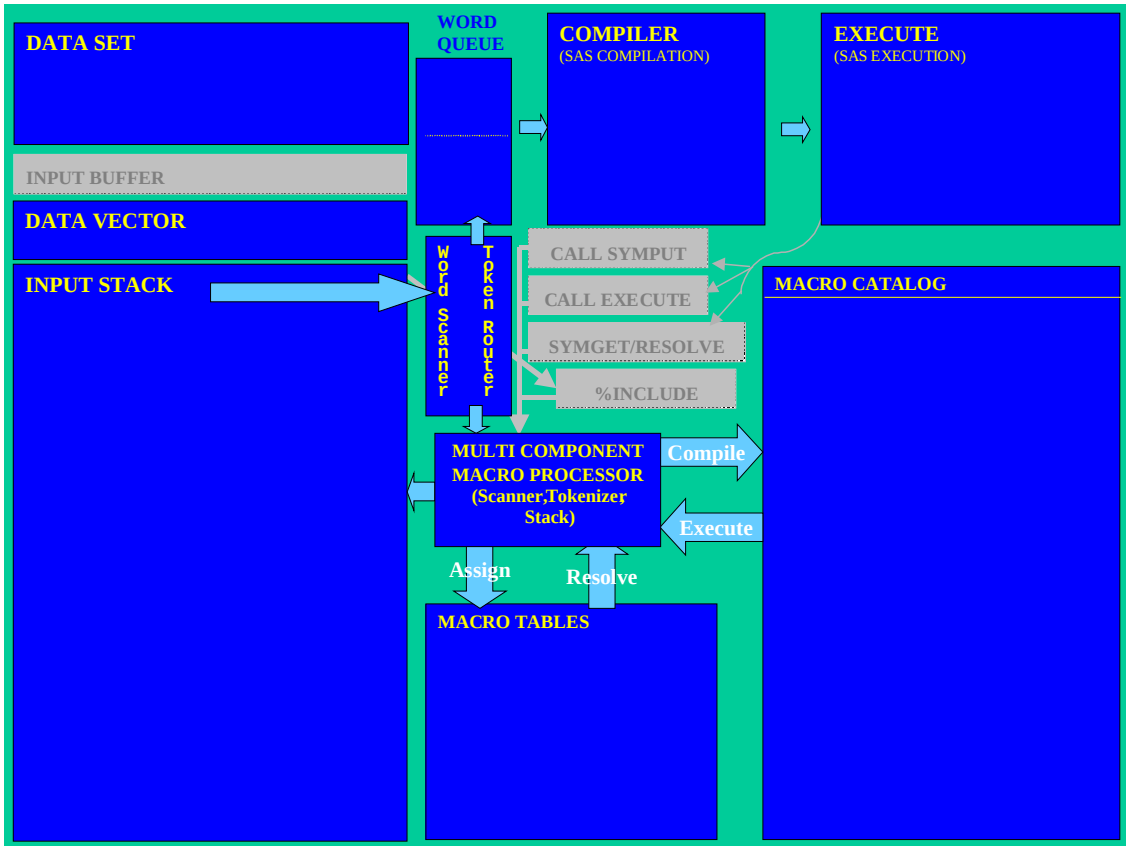

Figure 1

ABSTRACT
As all SAS ® programmers should understand the Program Data Vector, SAS macro programmers should understand the Map of the Macro Facility. The map is a powerful conceptual tool for understanding the macro process.

INTRODUCTION
This paper will present material from a presentation titled "An Animated Guide: The Map of the SAS Macro Facility". Two of the main deliverables of the presentation were the animation of the macro process and the ability of the PowerPoint slides to show the many conditions prevailing in the system. Showing many conditions at once allows the viewer to check his/her understanding of the process. Animation, while tedious to create, allows a presenter to show details of the processing. It is the opinion of the authors that macros have been difficult to understand because of use of the static teaching methodologies. The paper will cover the Map of the SAS system (not just the Macro Facility) and how it can be used as a mental framework for SAS macro processing. The map presented here is an integration of maps found in several books and I wish to acknowledge intellectual debt to the authors of the materials cited at the end of the paper.

OVERVIEW OF THE MAP
Figure 1 shows a simplified map of the SAS system, not just the Macro Facility. It is impossible to discuss the SAS Macro Facility without placing it in context of the SAS system. The control program for the SAS system is sometimes called the SAS Supervisor and the description of SAS processing in this paper is a description of the functioning of the SAS Supervisor.

REVIEW OF THE NON-MACRO COMPONENTS OF THE MAP & PROCESSING NON-MACRO SAS CODE
Each box in Figure 1 is a subroutine, a component, of the SAS Supervisor or the Macro Facility and serves a different purpose. An overview of the functions of each of the boxes follows.

The data set is a typical SAS data set and can be millions of lines long.

If the data source is a text file, the data will flow from the text file into the Input Buffer and then to the Program Data Vector (PDV). The Input buffer holds one line of unparsed data from an input text file in preparation for passing it to the PDV. It is not used in this presentation, but is included on the map for completeness. The PDV is a critical concept for SAS programmers. It can be thought of as a one-row spreadsheet. Data is read into the PDV from the input Buffer or SAS data set. Calculations are performed in the PDV and, when a data step has finished processing an observation, the values in the PDV are written to the output file.

The Input Stack is not a well known part of the SAS system. When you submit code, your code does not go directly to be compiled. It goes to a holding area called the Input Stack. The SAS Compiler can not use your code without pre-processing and the Input Stack holds code until it can be processed.

The Word Scanner/Token Router has two functions: 1) It takes characters off the Input Stack and assembles the characters into tokens (groups of characters that the compiler can process). 2) It also decides if the token should go to the Word Queue or to the "Multi-Component-Macro Processor. The Word Queue holds six tokens and allows the SAS supervisor to access "previously assembled" tokens to build context for a token currently being assembled by the Word Scanner.

Characters flow from the Input Stack to the Word Scanner. Characters are assembled in the Word Scanner into tokens. The tokens then flow through the system (either from the Word Scanner to the Word Queue or from the Word Scanner to the Macro Processor). The SAS Compiler is the boss of the system. It requests tokens from the Word Scanner until it is passed a token that indicates a step boundary (e.g. run, quit, proc).

When the SAS Compiler receives a step boundary token, it takes total control of the system and attempts to compile your code. No tokens are assembled, or move, while the SAS Compiler is in control. If the code is correct (matching quotes, spelling, semicolons etc.), it will be compiled and the compiled code is passed to SAS Execute module.

The SAS Execute module then takes total control of the process. The SAS Compiler, and other parts of the map, become inactive. If the job has no run errors (e.g. data mismatch etc) it will run. No tokens move while the SAS Execute module is in total control.

SAS TOKENS

The conversion of text to SAS tokens is an important, and basic, part of the SAS system. There are many kinds of tokens but we will only discuss the four most common. They are:
Character Tokens: Strings of characters in single/double quotes. (note that SAS handles single quoted character strings different from double quoted character strings)
Numeric Tokens: A string of digits, decimals (dates & times)
Name Tokens: The words that SAS recognizes (e.g. proc, var1, _n_)
Special Tokens: Characters other than letters/numbers (eg. / + = ;)

The Word Scanner, as it takes characters off the Input Stack and tries to assemble them into tokens, checks for a couple of things. First it checks every character it pulls off the Input Stack to see if the token currently being assembled has ended. The end of a token is either a blank space, a period or the start of another token (e.g. + or ;). It also checks for two characters called "Macro Triggers". These characters (the & followed by a letter or underscore or the % followed by a letter or underscore) are signals that (for a while) following text should be routed to the Macro Processor and not to the Word Queue.

REVIEW OF THE MACRO COMPONENTS OF THE MAP & PROCESSING MACRO SAS CODE

The SAS Macro Facility is often described as a "text processing system". That means that the Macro Processor stores text in either of two locations and you can command SAS to recall the text from these locations to the Input Stack. The two locations are the Macro Symbol Table (or Macro Table) and the Macro Catalogue. Moving text back and forth between the storage areas and the Input Stack is the function of the Macro Processor. By using statements like %if you can control *whether* a certain section of code gets recalled and moved to the Input Stack. Using %do allows you to control *how many times* a section of code is recalled from macro storage areas and moved to the Input Stack.

THE MAIN CONFUSION

One of the main sources of confusion in most macro documentation is the inappropriate re-use of the words compile and execute. SAS performs at least three “compiles” and three “executes” in a complicated macro program. These three compiles and executes have very different rules, do very different things and fail in very different ways. You get errors when rules are violated and when tasks can not be performed. Calling these different processes by the same names makes the overall process harder to understand.

We will create names to differentiate between the processes. As Figure 1 shows, this paper will give the different compiles and executes different names. The names are: SAS Compile, SAS Execute, Macro Compile, Macro Execute, Assign and Resolve.

We all have experience with SAS Compile and SAS Execute. They are the modules that process regular SAS code. The other compiles and executes are associated with loading code into, and removing code from, the two macro storage areas. The two macro storage areas, the Macro Symbol Table and the Macro Catalog, perform different functions.

MACRO SYMBOL TABLE

The Macro Table (or Macro Symbol Table or Symbol Table) is used to store strings that you want to recall later in your code. It generally does not have the ability to conditionally process code and it can be thought of as something like the clipboard in Windows® products. This paper calls putting values into the Macro Table "assigning a macro value". Most basically, the text strings are assigned (put into the Macro Table) with commands like:

```
%let comp= The Acme Storage Company         ;
or
%let year=    1988     ;
```

These commands are typed into your the body of your SAS code and when they reach the top of the Input Stack the Word Scanner examines them and passes them to the Macro Processor as is shown in Figure 2. The *%let* statement takes the text you place vbetween the equal sign and the first semicolon and puts the text into a named storage area (here, the name is month) in the Macro Table.

`%put Variablename` is a macro language statement that writes the values of a macro variable to the log.
The command *`%put _user_`* will write the names and values of user defined macro values to the log.

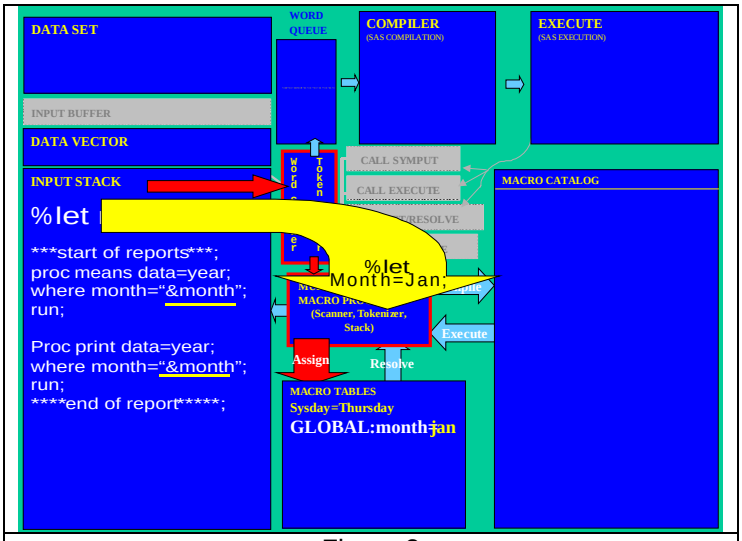

Figure 2

Below we use `%put _user_` to examine the contents of the macro variables we created above.

```
%put _user_;
GLOBAL COMP The Acme Storage Company
GLOBAL YEAR 1988
```

%let will automatically trim blanks between the equal sign and the first character in your text string. Note that the semicolons that were at the end of the statements (e.g. *%let year= 1988;*) were not stored in the Macro Table. The semicolon was part of (it was the end of) the *%let* instruction and was "consumed" by the Macro Processor. Values are usually recalled from the Macro Table by putting *&varname* (ampersand followed by the variable name) in your code. Values are recalled from the Macro Table and replace the original text **on the Input Stack**. A pictorial example of this process is given in Figure 3 and Figure 4.

%let will automatically trim blanks between the equal sign and the first character in your text string. Note that the semicolons that were at the end of the statements (e.g. *%let year= 1988;*) were not stored in the Macro Table. The semicolon was part of (it was the end of) the *%let* instruction and was "consumed" by the Macro Processor. Values are usually recalled from the Macro Table by putting *&varname* (ampersand followed by the variable name) in your code. Values are recalled from the Macro Table and replace the original text **on the Input Stack**. A pictorial example of this process is given in Figure 3 and Figure 4.

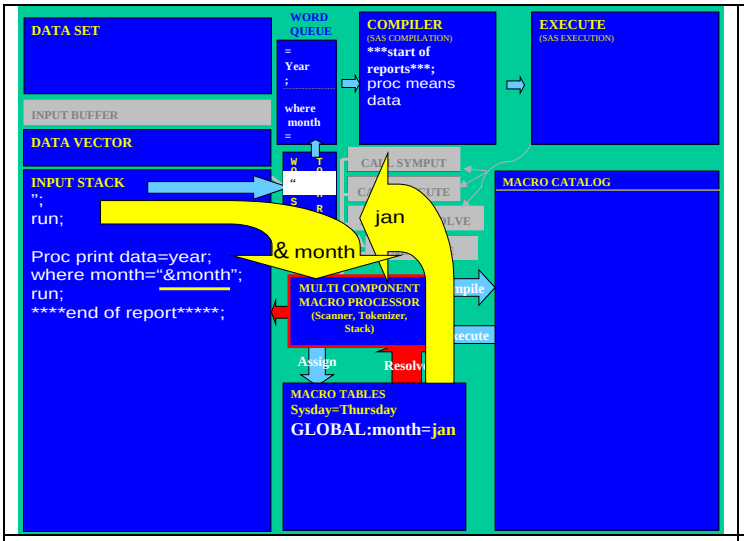

Figure 3

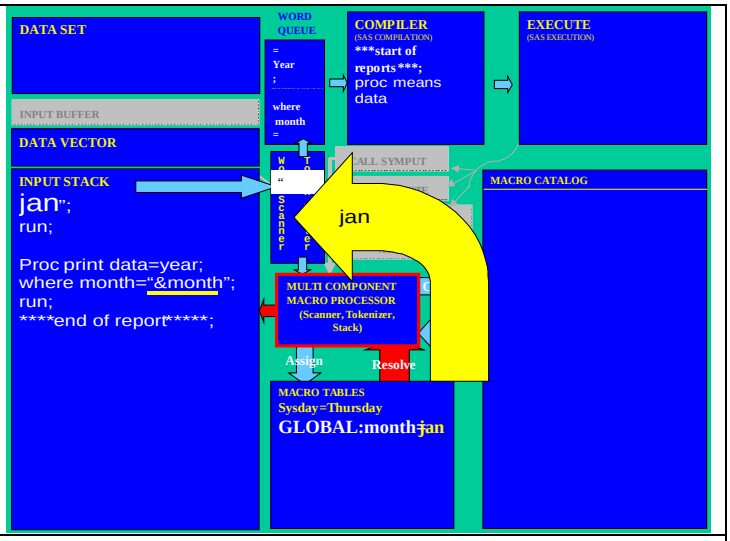

Figure 4

Some actions are taken when code is moved into and out of the Macro Table. The Macro Processor attempts to evaluate macro references/invocations as they are assigned. A reference is an attempt to resolve a macro variable (done with a *&varname*). A macro invocation is an attempt to execute a macro program (done with a *%macroname*). An example of this evaluation action, when assigning a macro, is illustrated below left where we issue a *%let* that contains an &.

```
/*use %put show values in the Macro Table*/
%put _user_;
      GLOBAL COMP The Acme Storage Company
      GLOBAL YEAR 1988


/*issue % let command with the & in it*/
%let fyr=fiscal_&year;

/*use %put to show values in Macro Table*/
%put _user_;
      GLOBAL FYR fiscal_1988
%put &fyr;
   fiscal_1988
```

&year is resolved as it goes into storage

Note that the Macro Processor did not store GLOBAL FYR fiscal_&year in the Macro Table. It checked the Macro Table for a variable with a name of year and evaluated &year to 1988 before storing the text in the Macro Table. If you want to store the string &year in the table you must tell SAS that &year is not a macro trigger by “Macro Quoting” it.

If you were to get the string fiscal_&year stored in the global macro variable FYR in the Macro Table, it would be evaluated when it is recalled and reaches the top of the input stack. There are ways to block evaluation, on assignment and on resolution, but they involve macro quoting. Quoting is addressed in parts two and three of this series of presentations.

MACRO CATALOG

The Macro Catalog is where SAS stores macro definitions. A macro definition is any code between a *%macro Macroname;* and a *%mend;*. Macro definitions can be simple or complex. Complex macro definitions use macro statements like *%if, %do and %end* to execute conditionally. We call putting text into the Macro Catalog “ Macro Compilation and this is illustrated in Figures 5 and 6. Macro definitions compile macros and put code into the Macro Catalog. Figure 6 summarizes some of the rules for "Macro Compilation".

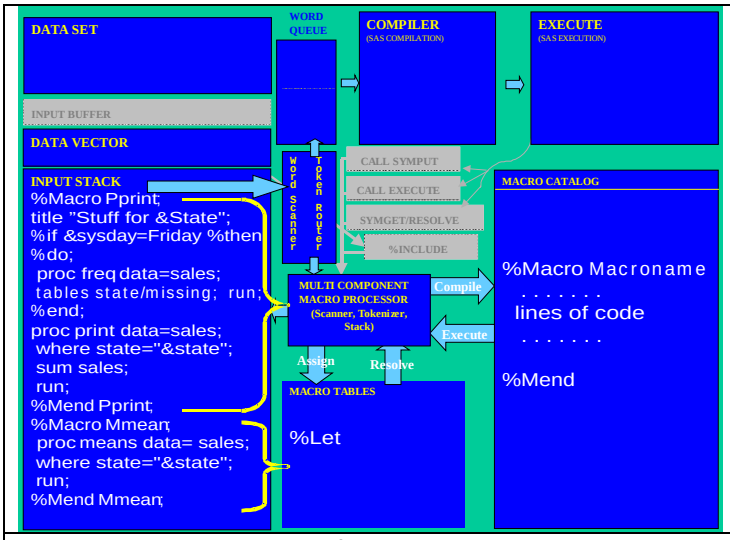

Figure 5

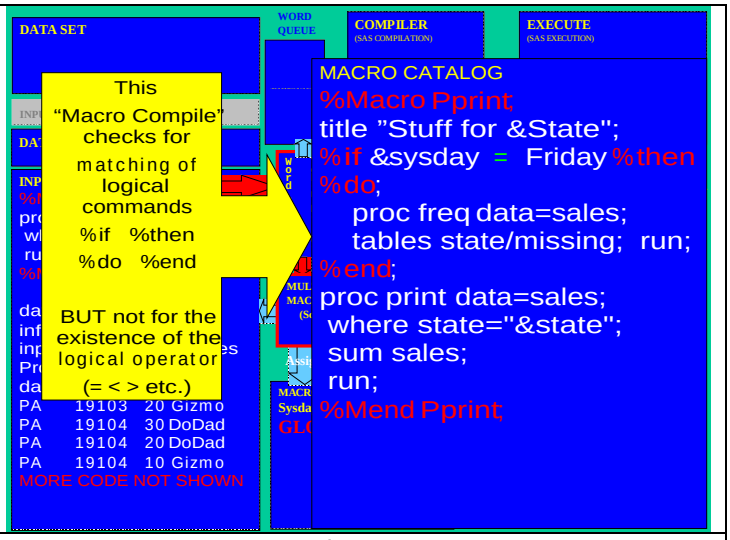

Figure 6

The Macro Compile process (moving code into the Macro Catalog) does not evaluate macro invocations (&state and/or a macro call) in the macro definition. Those macro references/invocations move, as text, from the Input Stack into the Macro Catalog. Figure 6 shows that the macro reference, &state, is stored in the catalog as &state. It will be resolved when &state is removed from the Macro Catalog and placed on the Input Stack.

The macro statements %if, %then, %do and %end are stored in the catalog in a “partially compiled” format. When the Macro Processor takes code from the catalog and puts it on the Input Stack, it never puts the %statements on the Input Stack! They are used by the Macro Processor to determine if *other text* in *the macro definition* should be moved to the Input Stack - as can be seen in Figure 7. Macro commands (%if, %then etc.) are instructions to the Macro Processor and are "consumed" by the Macro Processor as the other text in the macro definition is moved to the Input Stack.

Figure 7 shows the result of invoking the macro Pprint on a Thursday. %Pprint had reached the top of the Input Stack and had been passed to the Macro Processor. The Macro Processor started Macro Executing Pprint and, inside the Macro Processor, evaluated the %if to determine if the “proc freq” code should be moved onto the Input Stack. Since the logical condition %if Sysday=Friday was false, the code between the %do and the %end was not moved to the Input Stack.

Note that the code that was moved contains the string&state. It is worth noting where &state is evaluated. It gets put on the Input Stack and substitution (of PA for &state) occurs when &state reaches the top of the Input Stack and the &s and triggers the Macro Processor.

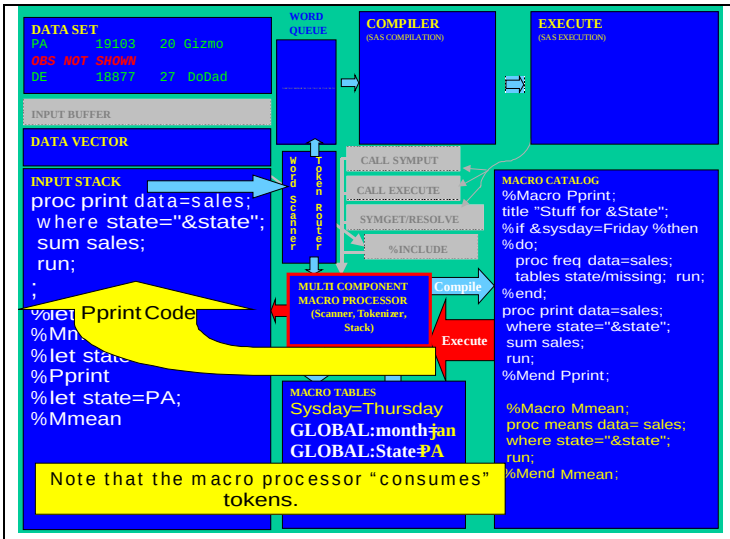

Figure 7

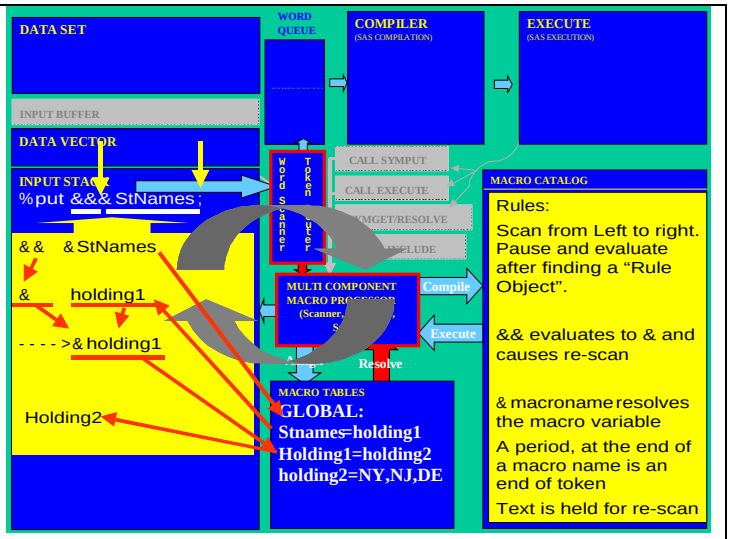

Figure 8

EVALUATING THE && OR &&& OR &&&

Tokens like &&&Vname&I are put on the Input Stack and evaluated or cycling between the Macro Processor and the Input Stack. The Input Stack is a slight misnomer. It is a buffer that allows characters to be taken off the queue but it also allows tokens to be pushed back onto the Input Stack (queue). An &&&Varname&i is taken off of the Input Queue, and sent to the Macro Processor. The Macro Processor partially evaluates the token and the "partial results" are pushed back on the queue. Then the cycle repeats until the &&&varname is fully evaluated. Examine Figure 8 and note the values in the Macro Table. Let us examine what happens when %put &&&StNames is processed.

The rules for evaluating && are in the box in the Macro Catalog. The rules are:

First: Scan from Left to right. Pause and evaluate after finding a "Rule Object". A rule object is something for which we have a rule. More rules follow immediately below.

Second: && evaluates to & and causes re-scan (push the results back on the Input Stack and prepare to take them off again). SAS knows that when it sees a && there must be additional macro resolution work to be done.

Third: &macroname causes a resolution of a macro variable. &macroname is just a request for SAS to resolve a macro variable.

Fourth: A period, at the end of a macro name, is an end of token flag.

Fifth: Text is held for re-scan. Remember, scanning proceeds in steps. If your scanning finds just text (no &), hold the text for re-scanning. Unless you made a coding error, the only way that SAS can scan your macro invocation and get just text is if you have previously coded an &&

Sixth: After applying rules one through five, assemble all objects on the Input Stack before the next rescan.

What SAS does with &&&stnames is shown in Figure 8:
It scans from left to right. When it recognizes && it stops and evaluates the && to an & and raises a flag indicating that the macro invocation must be scanned again. Then it scans &stnames. It recognizes this as a macro invocation and gets Holding1 from the Macro Table. Then it re-assembles the objects prior to rescanning. The rescan recognizes &holding1 as a macro invocation and gets the value holding2 from the Macro Table.

CALL SYMPUT

Call Symput allows your program to interact with the Macro Table at SAS Execution time not at the top of the input stack. It allows you to write strings into the Macro Table during the SAS Execution of a data step. Remember data is only available to your program when your code is SAS Executing. Call Symput allows you to take values from a data set, or from data step calculations, and load them into macro variables. This must happen while the data step is SAS Executing. Note the change in timing. All the macro processing, we have seen so far, has occurred when the macro trigger reached to top of the Input Stack and was passed to the Macro Processor.

Call Symput creates an entry in the Macro Table at SAS Execution time and takes two parameters, separated by a comma. Call Symput is difficult to understand because of the many options you can use to pass values to its two parameters, but the basic structure of the command is simple. You must tell Call Symput the name of the macro variable you want to create (parameter 1) and the value you want that variable to have (parameter 2). The syntax is: Call Symput(name, value);

Figure 9 shows a fairly complicated use of Call Symput (but still not all the options).
In this paper, we will only apply two rules for getting the parameter values into Call Symput. First; if the parameter value is quoted, consider it to be a text string, a constant. Second; if the parameter value is not quoted consider it a variable name. If the parameter is a variable name, go to the PDV and use the current value of the variable in the PDV as the parameter value. In Figure 9, you are using a data set to create macro variables. The data step has two Call Symput statements and we will examine the first. Look inside the parenthesis in Figure 9 for the details of the processing. The inside of the parenthesis is complex code because we are instructing SAS to assemble the parameters, and we are not just typing parameters into the code.

For the first (name) parameter we start with the text string "state" and concatenate information from the variable _N_ in the PDV. Then there is a comma to separate the parameters. Since the second parameter, state, is not quoted it is an instruction to go to the PDV and get the value of the variable named state for the observation being processed. State has the value PA.

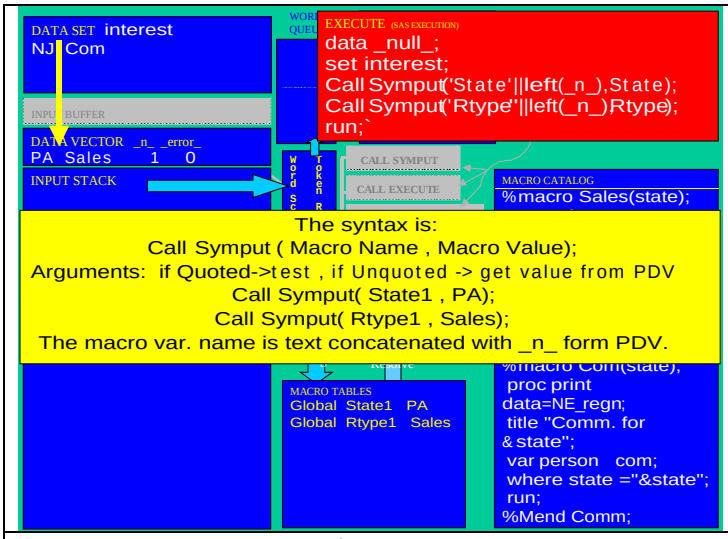

Figure 9

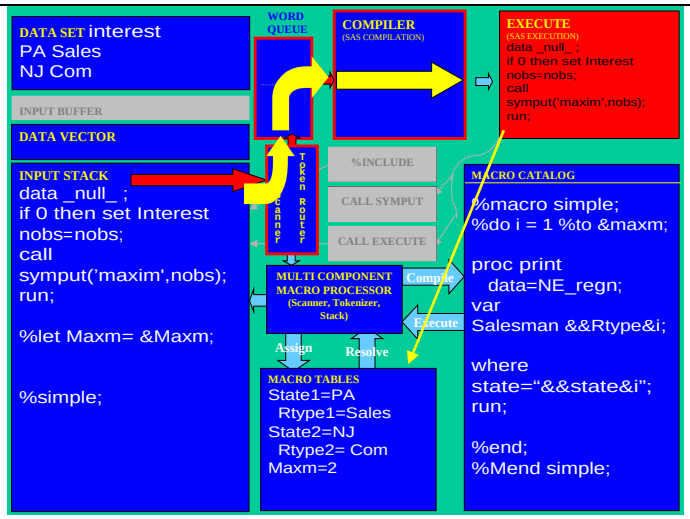

Figure 10

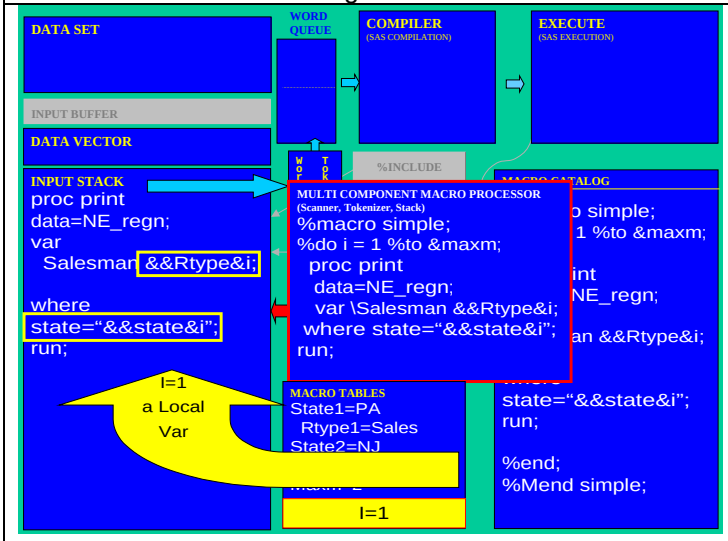

Figure 11

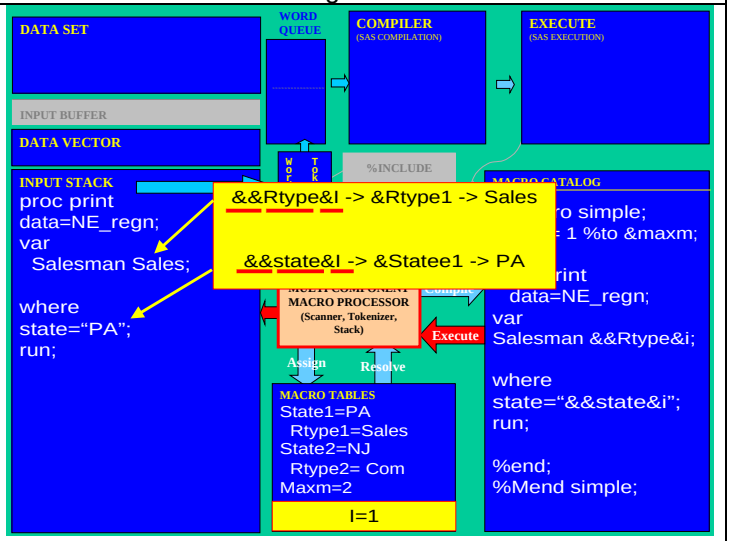

Figure 12

This code, executing on the values in observation 1, creates the macro variable state1 in the Macro Table and assigns it the value PA. This is shown in Figure 9. Results of fully processing the two observations are shown in the Macro Table in Figure 10.

Figure 10 starts a series of figures that show how the macro variables that you put into the Macro Table can be used in an automatic manner. The Input Stack in Figure 10 has a bit of code that finds out how many rows are in the data set “interest” and puts that number into the Macro Table in a variable called Maxm. You will want to process each observation you had in the data set “interest” and the code in Figure 10 loads the number of observations into a macro variable (Maxm) that you can use as an upper limit on the %do loop. Note that Figure 10 shows the macro call for the macro %simple working it’s way up the input stack.

When the string %simple reaches the top of the input stack, SAS will invoke and then Macro Execute the macro %simple.

While Macro Executing %simple, the Macro Processor will perform the %do loop twice (the value of &Maxm is the upper bound on the loop). SAS creates a local macro variable, called I, just for use in the looping.

The Macro Processor moves code to the Input Stack, as is shown in Figure 11. The rules for evaluating && calls are the same as were described in the previous section. Figure 12 shows the result of evaluating the &&. It is left to the reader to determine what SAS will put on the Input Stack when the macro variable &i has a value of 2.

CALL EXECUTE

Call Execute simply writes text to the Input Stack as can be seen in Figure 13. The key to the power of Call Execute is what is in the text you write to the Input Stack. With Call Execute, you can use a data set as a control file (a file of parameters) and use the control file to write macro calls to the Input Stack.

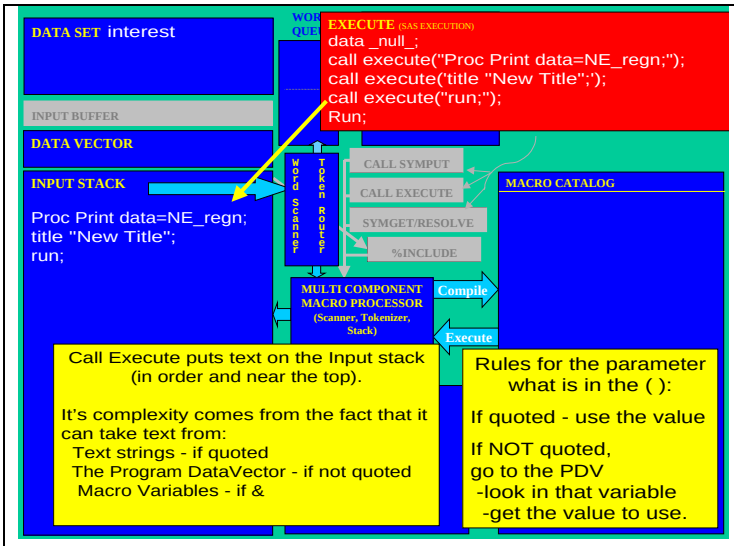

Figure 13

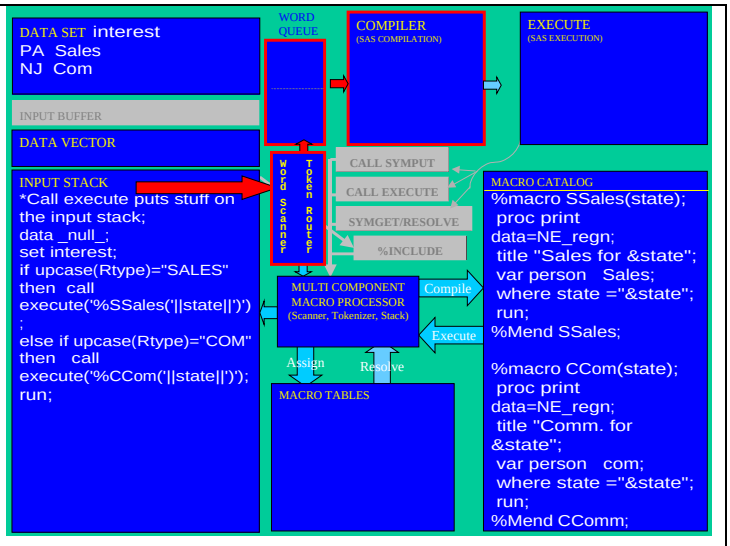

Figure 14

Figure 13 shows the trivial example of Call Execute assembling a proc by writing lines of text to the Input Stack. The writing to the Input Stack happens at SAS Execution time. When SAS Execution is happening, SAS Execute mode is in total control and no tokens move through the system. While the data step is SAS Executing, Call Execute statements put code on the Input Stack. That code must wait **on the Input Stack** for SAS Execution mode to give up control of the process. No characters are processed by the Word Scanner, and no tokens flow through the system while SAS Execute is in control.

Call Execute only has one parameter, the text inside the parentheses. As Figure 13 shows; when the Call Execute statement executes, it just puts that parameter on the Input Stack. The difficulty, and the flexibility, in using Call Execute is when the parameter is not a simple text string, as was shown in Figure 13. We can assemble the parameter from text strings and the PDV. Lets look at a more complicated parameter for Call Execute.

Figure 14 shows two compiled macros in the Macro Catalog and a data step containing a Call Execute that is about to be Word Scanned. Interest is a control file and variable values in the file named interest will both direct the creation of two macros and be passed as macro parameters to those macros. In the data set interest Figure 14, the variable rtype (with values of Sales and Com) will be used to select which of the two macro programs in the Macro Catalog is to be put on the Input Stack. The variable state (with values of PA and NJ) will be a parameter passed to a macro.

Figure 15 shows the Call Execute in operation. To make things easier to read , the SAS Execute box is larger than normal size and the Call Execute is in its own special box. The line executing in the data step is underlined in gold and the first observation is in the PVD. The user defined variables in the PDV are state and rtype. The statement if upcase(rtype)="SALES" statement checks the PDV for the value of rtype (report type) and since rtype is sales SAS processes the end of that if statement. It processes Call Execute shown below.

```
if upcase(Rtype)="SALES" then Call Execute('%SSales('||state||')');
```

The complex part of this code is the fact that SAS is assembling the Call Execute parameter (A parameter which, Call Execute will simply put on the Input Stack). Let us examine the process of assembling the parameter.

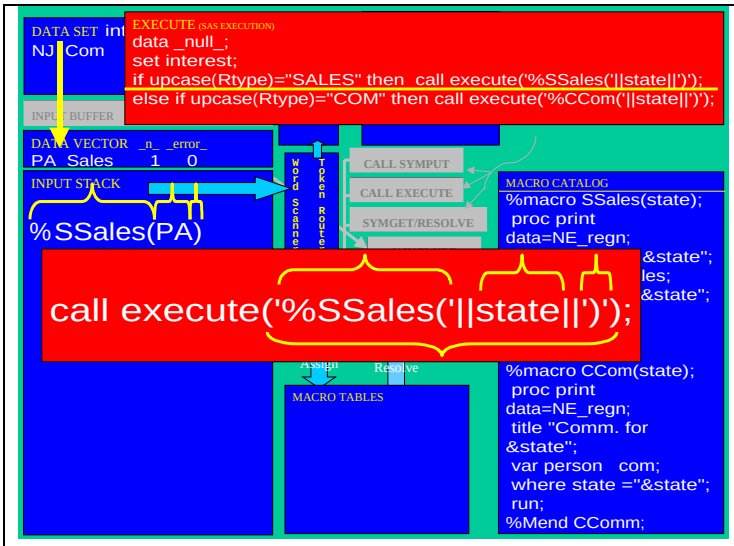

Figure 15

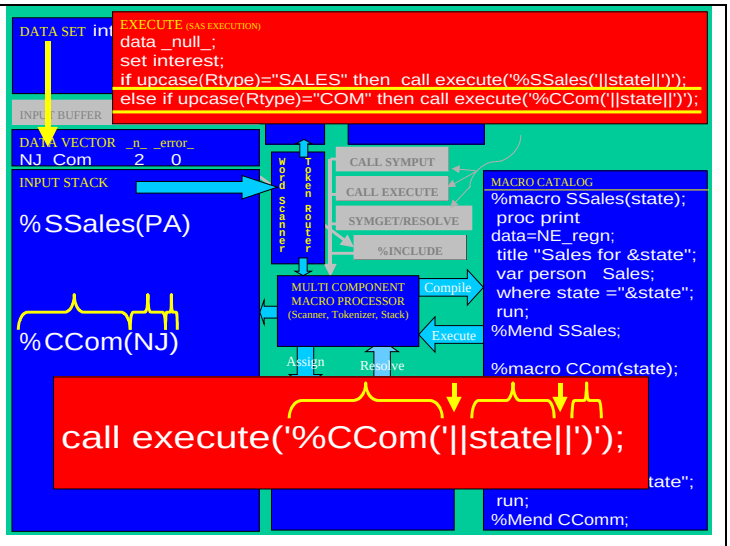

Figure 16

Call Execute assembles the parameter by concatenating three things. First is the quoted string '%Ssales' . Note that it has been single quoted so that the Word Scanner will not attempt to evaluate %Ssales at SAS Compilation. The || symbols are concatenation operators. Then, since the next element (state) is unquoted, SAS will get the value of state from the PDV. Then we have || to indicate another concatenation. Finally, when we add the string ')'.

After the concatenations, the parameter for Call Execute is %sales(PA) and that parameter is placed on the Input Stack. What has been placed on the input stack is a request to run a macro and the parameter that the macro requires. The string waits on the Input Stack until the data step finishes processing. Call Execute has allowed us to use a data set , containing conditions and parameter values, to "make" many macro calls. Figure 16 shows the Input Stack after processing the second observation in the data set. Interest is a control file. It controls what macros will be run and what parameters the macros will have.

SYMGET & RESOLVE

These two functions are mirror images of Call Symput. They both get values from the Macro Table during SAS Execution and can often be substituted for each other. They both take just one parameter, the macro variable to be recalled from the macro table. They differ in their syntax (use of quotes) and Resolve is better able to handle && references. There is little written about these functions because, it is rare to find a compelling business reason to use them.

The map shows that SAS Execution follows SAS Compilation and that means that Symget and Resolve will perform if the SAS code that contains these functions has been SAS Compiled. Programmers using compiled SCL code or large precompiled data steps might profit from using these functions to pass information from the “run time environment” to the compiled code.

Ian Whitlock(1998) shows an interesting use of these functions. He uses Symget to load a data step array with values from macro variables. His code follows immediately.

```
%let state1=NY;
%let State2=DE;
%let State3=PA;
%let State4=NJ;

%put _user_;

data _null_;
  array states(4) $ _temporary_;
   Do i=1 to dim(states);
      states(i) = symget("state"||left(put(i,2.))) ;
   end;
   Do i=1 to dim(states);
      Put @1"state" @7 I= @12 States(i);
   end;
run;

The output is:
state i=1 NY
state i=2 DE
state i=3 PA
state i=4 NJ
```

Dr. Whitlock uses looping, inside a data step, to access macro variable information.

This code takes information from the Macro Symbol Table and puts it into the PDV.

CONCLUSION

The map of the macro system is a powerful conceptual tool for understanding the macro system and how it works.

REFERENCES

SAS Macro Programming Made Easy By: Michele M. Burlew
Carpenter's Complete Guide to the SAS Macro Language By: Art Carpenter
SAS Macro Language: Reference, Version 6, First Edition Whitlock, Ian(1997) Call Execute: How and Why
Proceedings of the Twenty-first Annual SAS Users Group International, pp. 410-413

Whitlock, Ian(1998) The Resolve Function- What is it good for? Proceedings of the Fifteenth Annual NorthEast SAS
Users Group Conference, pp. 352-353

CONTACT INFORMATION

Your comments and questions are valued and encouraged. Contact the author at: russ.lavery@verizon.net