

Macro Internals for the User – Developer’s Overview

Susan O’Connor, SAS Institute Inc., Cary, NC

ABSTRACT

You have used the macro language software that is part of base software from SAS Institute Inc. or you have inherited some SAS macros to support. Whether you have some working knowledge of the SAS macro facility or you have been afraid to change those legacy macros, some knowledge of macro internals would empower you to use macro with less anxiety. Just as you might be a more confident driver if you knew the internals of your automobile engine, you might be a more confident SAS macro user if you knew the internal details of the SAS macro processor.

As a macro software developer, I have been conducting macro seminars in-house for our SAS testing groups, SAS quality assurance and SAS technical support. These sessions deal with macro topics and questions, and our SAS folks tell me that macro internal knowledge has helped them with writing and supporting their macros. As a result of these seminars, I have developed a series of internal overviews that are designed to help eliminate confusion when using macro.

The internal overview includes:

- Tokens, timing, and macro generated code
- Macro compilation and execution
- Interpreted macro code and constant text
- Macro variables, their creation, and their scope
- Examples of macro variables with the DATA step and macros with SAS/CONNECT®
- Session compiled, autocall, and compiled-stored macros

INTRODUCTION

I want to define some terms and relate them to the SAS System and to the SAS macro facility particularly.

A classic definition of a compiler is a program that decodes instructions written in pseudo codes and produces a machine language program to be executed at a later time. A compiler is a program that translates instructions written in a high level

programming language into a lower-level language or set of instructions that can be executed. There are several parts of the SAS® System that can be referred to as a compiler: SCL, the DATA step, some PROCs and macro. Sometimes, the term compiler is used in a cavalier manner and, in one sentence, “compiler” can mean DATA step compiler or macro compiler. This confusion is found in SAS documentation and in SAS courses. While there are many compilers in the SAS System, they are all different. The SAS macro compiler will be the focus of this paper. For the purpose of illustration, I will create a simple macro pseudo code to illustrate compiled macro instructions.

This SAS macro compiler is also sometimes referred to as the macro interpreter. An interpreter, in the classic sense, is a program that translates and executes each programming instruction of a high-level programming language before it translates and executes the next instruction. The SAS macro facility translates all statements of a SAS macro definition into instructions in one compilation phase. The macro definition is between the %MACRO statement and the %MEND statement. Then the macro facility executes or runs these instructions in another phase, when the macro is invoked. This behavior makes it a compiler, but let’s not quibble if some prefer to refer to it as an interpreter.

To add to the confusion, the SAS macro facility is sometimes referred to as a preprocessor, which is a program which examines the source program for preprocessor statements which are then executed, resulting in the alteration of the source program. This is a fair description of the macro facility too. So whether we refer to macro as a compiler, interpreter or preprocessor, the important thing is to understand what happens internally. Macro creates instructions and constant text.

The last term to clarify is tokenization. This means the breaking down a character string into the smallest independent unit, a token that has meaning to the programming language.

TOKENS FROM WORD SCANNER

One of the main jobs of the SAS supervisor is to split incoming character strings of code into tokens. These tokens are classified into SAS token types: words, numbers, integers, compound words, missing values, semicolons, special characters, formats and quoted strings. The different types of tokens may have subtypes, for example, there are single and double quoted strings as well as hexadecimal quoted strings and binary quoted strings. It is the word scanner that separates the string into small tokens with token types.

Some examples of token types include:

- Quoted strings - "O'Connor", 'This is easy', 54321'x, 010111010101b
- Word tokens - DATA, WORK, x, EQ, _ALL_, Y2K, MIN
- Integers -100, 700,000, 0, 123456
- Numbers - 123.50, 1.2e23, 0c1x
- Date, time, and date time constants – '9:25't, '01jan2000'd
- Special tokens - =, *, !, %, &, /, :, +, “, ‘

These tokens are passed off from the word scanner to waiting parts of the SAS System. A string of code might come into the SAS supervisor from many sources. For example, source might come from something you type at the keyboard, from a %INCLUDE statement, from a DMS command-line command, or be pushed on by a SAS/AF® application. The string of code might also come in from constant text generated by macro execution. The string of code might come into the word scanner from macro variable values substituting for macro variable names. Parts of the system are waiting for that string to be delivered to it in small tokens. If you typed in this source code at your terminal:

```
DATA WORK.B;  
  X=5*10.10;  
  PUT X DOLLAR10.2;  
RUN;
```

the stream of characters that you type would pass through the word scanner. Small tokens would be chopped out of the character stream. In this case, they would go up to the tokenization routine, passing through a DATA step grammar, and on to the DATA step compiler. Grammar processing is between the word scanner and

tokenizer for the DATA step and PROCs. But for our macro purposes we will talk about what tokens the word scanner produces since it is at the word scanner level that most macro occurs:

- DATA - Word type
- WORK.B - Compound type
- ; - Special semicolon type
- X - Word type
- = - Special type
- 5 -Integer type
- * - Special type
- 10.10 - Number type
- ; - Special semicolon type
- PUT - Word type
- X - Word type
- DOLLAR10.2 - Format type
- ; - Special semicolon type
- RUN - Word type
- ; - special semicolon type

Basic SAS tokens are then processed by different parts of SAS with different compilers, different grammars, and parsers. But this process comes after the word scanner works on the incoming string of code. The word scanner separates the asterisk character above from the string of code. It does not care if the asterisk is followed by leading or trailing blanks, but recognizes that it is a unique token. Later in processing, the DATA step grammar will determine that this is a multiplication sign instead, for example, the start of an asterisk style comment. After the word scanner sends up the token RUN, the DATA step compiler will recognize that token as the end of the DATA step.

But this is just one example. The following SAS/CONNECT code strings would be broken into tokens by the scanner in a manner identical to the DATA step example above, but these tokens would be sent from the scanner to the CONNECT code on the client side once the word token RSUBMIT is detected.

```
RSUBMIT;  
  DATA WORK.B;  
    X=5*10.10;  
    PUT X DOLLAR10.2;  
  RUN;  
ENDRSUBMIT;
```

CONNECT builds a buffer with these tokens to send from the client to the server. The client would build the buffers and send them to the

server until the CONNECT code on the client detects the token ENDRSUBMIT. In this case, the buffer will contain the code following and pass it to the server for processing:

```
DATA WORK.B; X=5*10.10; PUT X  
DOLLAR10.; RUN;
```

On the server, the SAS System will pass the incoming character stream from the buffer into the scanner on the server side so that the incoming stream will be broken into tokens. On the server side, the tokens will be passed to the DATA step compiler on the server side.

WORD SCANNER

Tokens are generated by the word scanner one token at a time. The word scanner examines incoming streams of code looking at every character and following rules for making SAS tokens. This knowledge of how the word scanner processes incoming lines of code might help programmers select better methods and coding standards.

The word scanner, for example, recognizes the PL/I style comment beginning with `/*` and eats characters and white space until it sees the closing `*/` characters.

```
/* The word scanner eats every character */  
/* in a PL/I style comment. */
```

These characters, in effect, never get out of the word scanner to the rest of the SAS System. The word scanner does not even bother examining the characters to build tokens, but just eats character by character until it sees the closing `*/`. However, an asterisk style comment begins with a special `/*` token which is sent out of the scanner. SAS needs to decide if this is a multiplication sign or the start of an asterisk style comment. As soon as SAS recognizes the special token `/*` as the start of an asterisk style comment the system will call the word scanner for the next token and the next and so on until the asterisk style comment handling code finally sees a semicolon token which ends SAS statements.

* Asterisk style sends up tokens from scanner;

You can see that it might be a bit faster to use slash asterisk style comments in your code and avoid the overhead of tokenizing asterisk style comments. It is faster indeed to eat the characters, rather than to tokenize them, but both methods will print your comments on the SAS log.

The word scanner counts the number of blank spaces and white space before a token and passes that information on with the token to other parts of the SAS System. The word scanner is the lowest level where input is separated into tokens and it is also a logical place to insert new tokens or streams of code.

TRIGGERS IN THE SCANNER

It makes sense that this word scanner is the place to insert some new characters. You can think of macro as a way to insert new code into existing code. Macro generates strings of code that can be placed in incoming strings of code. The macro facility is triggered in the word scanner as it is looking at each character in a string of incoming code and it sees a percent sign or an ampersand.

Consider the following incoming code, which will be sent on as tokens from the word scanner to the DATA step compiler. The developer wants the DATA set name to reflect the day of the week.

```
DATA &SYSDAY;
```

As this string of code comes into the scanner each character is examined. The first token DATA, a word token, is sent on up to the DATA step compiler. The next token is examined. It is the special character `&` that is a "trigger" for the macro facility. When the word scanner examines the character `&`, the scanner calls the macro facility's macro symbolic substitutor. The macro symbolic substitution part of the macro facility needs a token to decide what to do. Like the rest of the SAS System, macro calls the word scanner to get the next token, which is SYSDAY. This call by macro into the word scanner is recursive; the scanner is waiting to see if the macro facility is going to add a new string of code while the macro facility is waiting for the scanner to pass it a new token to see if there is anything to substitute.

The SAS word scanner is recursive and repeatedly calls into itself to substitute strings of

code and build tokens for the rest of the system. Macro triggers in the word scanner are the ampersand and the percent sign. Most parts of the SAS System that are waiting for tokens allow these triggers to call macro. However, sometimes a part of the system might "turn off" these triggers in order to build a string that would not substitute text. While macro triggers are rarely "turned off", one case used in this paper is when SAS/CONNECT is building a buffer on the client after it has seen RSUBMIT. More about this later.

AUTOMATIC MACRO VARIABLES

One part of macro is macro variables. The easiest macro variables to understand at first are automatic macro variables. At initialization SAS supplies a list of automatic macro variable values. These macro variables usually begin with the prefix SYS with a few AF prefixes and one FSP prefix. Examples are SYSDATE, SYSDATE9, SYSDEVIC, SYSSCP, and SYSSITE. To get the complete list of these values do:

```
%PUT _AUTOMATIC_;
```

These automatic macro variable values are the simplest place to begin to understand macro. Note, however, if you use those prefixes for your macro variable names, first SAS will first check to make sure that you are not using an automatic macro variable it reserves. It may be a bit faster to avoid those prefixes for macro names.

The SYSPARM automatic macro variable can supply a macro variable name SYSPARM with a value at SAS invocation on the command line. (SYSPARM works on all operating system command lines but the operating system command line syntax will differ.) Here we are invoking SAS on a system with a UNIX ® operating system command line

```
SAS -DMS -SYSPARM "This is &SYSDAY"
```

During SAS initialization the values for automatic macro variables are created. The host code calls the operating system for information. For example, the day of the week will be determined for possible use as a value for SYSDAY resolution. The host code will determine an abbreviation for the name for the operating system. Examples include for AIX® AIX_370, for HP® HP300, for Windows®

WIN_98 and so on, for the values for SYSSCP and a longer version for SYSSCPL.

In the SYSPARM case above, the value "This is &SYSDAY" will be stored in memory for access if the macro variable value SYSPARM is referenced later. While the value is stored in memory, the value of the macro variable &SYSDAY is not resolved. If the invocation command did not have the SYSPARM option create the value, the SYSPARM automatic variable would still exist and would have a null value.

One simple method to check the value of SYSPARM is:

```
PROC OPTIONS OPTION=SYSPARM; RUN;
```

or:

```
DATA _NULL_;  
    X= SYSPARM( );  
    PUT X=;  
RUN;
```

In both these cases, the character string that was passed into the SAS program will be listed because that is how it was stored at initialization of SAS. The SYSPARM function would simply copy the string stored by the host as would PROC OPTION.

This is &SYSDAY

However, if we were to reference SYSPARM using macro:

```
%PUT &SYSPARM;
```

the macro variable SYSPARM would resolve and the string would be substituted. Then that string would be pulled through the scanner and SYSDAY would be resolved for the day of the week saved at initialization.

This is Monday

Most automatic macro variables, except SYSPBUFF, are referred to as global macro variable values because they are available globally through the session.

OPEN CODE

Open code is code that is not inside compiling or executing macros. When a macro statement in open code is executed, it completes its task immediately. Macro statements that are allowed in open code are %PUT, %LET, %GLOBAL, %SYSEXEC, %WINDOW, %DISPLAY, and %INPUT.

When the %' is seen in the word scanner the macro open code handler is called. This open code handler requests a token of the word scanner in a recursive manner. For example, in open code, the following statement:

```
%LET EXAMPLE = THIS OPERATING  
SYSTEM IS &SYSSCPL;
```

will enter the word scanner as a character stream. When the %' is seen the word scanner calls the open code handler. The open code handler recursively asks the scanner for the next token. The word scanner sends the open code handler the token LET. The open code handler recognizes this as a macro keyword and recursively asks the scanner for the next token, EXAMPLE. It checks to see if it is a valid macro name that begins with a letter or underscore, contains alphanumeric characters or underscores, and is not greater than 32 characters in length (or 8 characters in versions before 7.01). When the macro variable name passes inspection, the open code handler asks the scanner for the next token, which must be '='. Then there are repeated requests for tokens from the word scanner until a semicolon is seen.

The %LET code in the open code handler immediately creates a global macro variable EXAMPLE with a value of THIS OPERATING SYSTEM IS OS/390 if we were on that operating system. Notice that the word scanner had resolved the automatic macro variable SYSSCPL when it saw the ampersand and used the resolved value in the string it was using to build the value for example.

The key here is that in open code the macro statements are immediately executed. The %LET statement immediately creates the macro variables in the global symbol table. The %PUT statement immediately writes the immediately resolved string until the semicolon to the SAS log. %GLOBAL immediately creates macro

variables with null values in the global symbol table.

```
%GLOBAL A B C;
```

Immediately the open code handler creates macro variables A, B, and C in the global symbol table and these values are null values.

MACRO COMPILATION – A SIMPLE CASE

Macro compilation is the activity that happens between a %MACRO statement and a matching %MEND statement, the macro definition. When the percent sign is seen in the word scanner, the open code handler is called. The open code handler calls the scanner for the next token, MACRO. This keyword MACRO causes the macro facility to begin compiling a macro.

```
%MACRO SIMPLE;  
DATA A;  
X=1;  
RUN;  
%MEND SIMPLE;
```

The open code handler calls the word scanner for the next token, SIMPLE, and makes sure it is a valid macro name following the rules previously stated. If everything looks good, the statement is read until the semicolon is detected and a compiled macro header instruction is made. The rest of SAS cannot read this instruction but later the macro compiler can interpret it when the macro is executed. The compiled macro header instruction is a record containing information such as the date compiled, version, keyword or positional parameters, and options.

In addition to the header instruction there are pseudo instructions for parameters, constant text, %MEND, %IF, %PUT, left and right hand parts of %LET, %LOCAL, %GLOBAL, labels, %GOTO, jumps around unexecuted code, iterative %DO, %DO %WHILE and %DO UNTIL, ends of the various do instructions and so on. For illustration purposes, I will use only a few of these instructions and represent them in a simple readable manner using a pseudo code language.

Using a type of pseudo language we can understand the compiler instruction might look like the following and be numbered in order with a zero base:

```
0 HDR SIMPLE key=0 pos=0 ver=8.00
  date=13998
```

The next part of the code is the constant text that is gathered a token at a time until there is another macro statement to compile. This constant text is gathered as a macro compiled text instruction containing the length of the constant text and the original case. The maximum length is 256 and then a new instruction of constant text would be written. In the pseudo code we are using it might look like:

```
1 TEXT LENG =20 DATA A; X=1; RUN;
```

Then the scanner sees the percent sign, goes into the open code handler, gets the token MEND, and gets the tokens until the next semicolon. Checks are made for extraneous information on the %MEND statement and a compiled %MEND statement that in our pseudo code might look like:

```
2 MEND
```

These instructions are written as three records in an object SIMPLE in the SAS macro catalog in WORK.SASMACR.

This is the first pass of the macro compiler. There is a second pass to resolve things unknown in the first pass such as labels for %GOTO statement and ends for %DO loops. This simple case gives you an idea of pseudo code might look like for a compiled macro; it is a set of instructions and constant text instruction.

COMPLEX MACRO COMPILATION EXAMPLE

Using our pseudo code to represent compiled macro instructions, here is a more complex example with compiled macro statements instructions and constant text.

```
%MACRO COMPLEX;
  %DO I = 1 %TO 3;
    DATA A; X=1; RUN;
    %LET Y&I = &I;
  %END;
  %PUT _LOCAL_;
%MEND COMPLEX;
```

The basic items discussed above are here with new complexity:

```
0 HDR COMPLEX key=0 pos=0 ver=8.00
  date=13998
1 ITERATIVE_DO ID I START 1 STOP 3 BY
  1 EXIT 6
2 TEXT LENG =18 DATA A; X=1; RUN;
3 LETLEFT LENG = 3 Y&I
4 LETRIGHT LENG=2 &I
5 IDOEND ID 1
6 PUT LENG=6 _LOCAL_
7 MEND
```

In the compilation phase the macro COMPLEX has compiled a lower level machine language, which will be executed when the compiled macro %COMPLEX is invoked. Notice how the %LET code now has pseudo instructions with a left side for the macro variable name and right side for the macro variable value. The pseudo code instructions exist but no macro variable has been created. %PUT is just an instruction and nothing has been written to the SAS log. These instructions are stored on disk in the catalog WORK.SASMACR and will be read and executed by the macro compiler when the macro is invoked.

UNDERSTANDING MACRO COMPILATION

Only one macro at a time may compile. If you are compiling a macro every macro statement between the %MACRO and the %MEND will become a macro instruction. Everything else will become constant text instructions.

Since macros may only compile one at a time once we are inside a compiling macro another %MACRO statement will just become constant text. If the definitions are nested, then macro statements inside nested macro definitions will become constant text. For example, consider the following macro definition for FIRST with the macro definition for SECOND nested inside it

```
%MACRO FIRST (PARM1, PARM2);
  %IF &PARM1 > 0 %THEN %DO;
    %MACRO SECOND;
      %LET FRED =;
      PROC PRINT; RUN;
    %MEND SECOND;
  %END;
%ELSE %DO;
  %LET FRED = 500;
  DATA A; X=1; RUN;
  %PUT &FRED;
```

```
%END;  
%MEND FIRST;
```

The pseudo code for this compiled macro FIRST would look like:

```
0 HDR FIRST key=0 pos=2 ver=8.00  
  date=13998  
1 PARM pos=1 PARM1  
2 PARM pos=2 PARM2  
3 IF falseaddr=6 leng=10 &PARM1 > 0  
4 TEXT LENG=90 %MACRO SECOND;  
  %LET FRED = ; PROC PRINT; RUN;  
  %MEND SECOND;  
5 JUMP addr= 10  
6 LETLEFT LENG=4 FRED  
7 LETRIGHTLENG=3 500  
8 TEXT LENG=24 DATA A; X=1; RUN;  
9 PUT LENG=5 &FRED  
10 MEND
```

Notice that the macro SECOND is constant text since only one macro, in this case FIRST, at a time can compile. Everything inside the macro definition for SECOND would be constant text, even macro instructions that would have been compiled as macro instructions if they had not been inside a nested macro definition. During execution of FIRST, if the IF code instruction evaluates true then the macro SECOND is compiled. This suggests that nested macro definitions inside other macro definitions would be inefficient macro code.

Also notice that there are two positional parameters PARM1 and PARM 2. If the IF pseudo code instruction 3 evaluates false, we would jump to the pseudo code instruction 6, the left side or macro variable name side of the %LET pseudo code. The %ELSE pseudo code will not be hit if the %IF pseudo code is TRUE and executed because after the constant text in instruction 4 at instruction 5 we jump to instruction 10 the MEND pseudo code instruction.

MACRO EXECUTION – A SIMPLE CASE

Macro execution begins for name style macro when the word scanner detects a percent sign and calls the open code handler. This sets in motion the use of the macro interpreter to understand, interpret or execute the pseudo code macro instructions that only mean something to the macro interpreter. In the case,

```
%SIMPLE
```

the open code handler requests the next token from the word scanner, SIMPLE. It looks to see if this next token is a reserved macro keyword and since it is not, it tries to invoke a macro name SIMPLE. First it looks in the catalog WORK.SASMACR for a macro entry SIMPLE. If it is not found, it also looks for compiled stored macro named SIMPLE if there are compiled stored macros. Subsequently, it will search for an autocall macro named SIMPLE. In this case, SIMPLE is found as a session compiled macro in WORK.SASMACR.

The catalog entry SIMPLE is opened and the first instruction is read. It is the header pseudo instruction and there are checks for version, date, and so on. Remember the rest of SAS is waiting for a token while the macro facility is looking at the pseudo code instructions. If everything looks all right with the header instruction, the scanner tries to get another token, but it must call the macro interpreter for another instruction or for constant text because macro is executing. In the case of the macro SIMPLE defined earlier, the macro interpreter reads the next instruction, which is a constant text instruction.

```
1 TEXT LENG =20 1 TEXT LENG =20 DATA  
  A; X=1; RUN;
```

The constant text is pushed from the macro interpreter back into the word scanner as a stream of text:

```
DATA A; X=1; RUN;
```

The word scanner will break this stream of text into tokens following the rules for tokens and pass these tokens to rest of SAS, which is waiting for tokens. In this case, the tokens will leave the scanner, pass through the DATA step grammar, and pass through the tokenizer and on to the DATA step compiler, which will compile a DATA step until the token RUN is received.

The constant text that was pushed to the scanner has been consumed and the rest of SAS is waiting for the next token. The word scanner tries to get another token but must call the macro interpreter for another instruction or for more constant text. In this case, the instruction is for the MEND, the end of macro execution. The

macro entry SIMPLE will be closed in WORK.SASMACR and the macro catalog WORK.SASMACR will remain open on the theory that you will be looking for another macro soon.

A MORE COMPLEX MACRO EXECUTION EXAMPLE

Using the pseudo code from the compiled macro COMPLEX example, when the macro is invoked

```
%COMPLEX
```

The word scanner calls the open code handler when it detects the percent sign. The open code handler recursively calls the scanner for the next token, COMPLEX. It is not a macro keyword so it tries to open a macro COMPLEX. It reads the header and everything looks fine.

```
0 HDR COMPLEX key=0 pos=0 ver=8.00
  date=13998
```

Now the macro interpreter is going to be called as the scanner looks for new character strings to make into tokens or to execute instructions. The scanner calls the macro interpreter, which reads the next instruction.

```
1 ITERATIVE_DO ID I START 1 STOP 3 BY
  1 EXIT 6
```

This instruction is a count of where to start and stop a loop by a count of 1 and it increments a macro variable value I from the start to the stop when the condition is met. When the stop condition is met we will read the instruction 6.

The first time through the loop the iterative value is 1 and the macro variable I is created in the scope of this macro with a value of 1. Also, the first time through we find constant text,

```
DATA A; X=1; RUN;
```

which we send to the scanner to be broken into tokens. We also have pseudo code for the left and right side of a %LET instruction to execute. The left and right side of the LET instructions will pass the character strings Y&I and &I to the scanner. The ampersands trigger macro variable resolution and the value for the macro variable name I is 1. The left hand side will create a macro variable name Y1 and the right hand side will give it a value of 1.

```
2 TEXT LENG =18 DATA A; X=1; RUN;
3 LETLEFT LENG = 3 Y&I
4 LETRIGHT LENG=2 &I
```

The second and third time through the loop the value of I has been incremented by 1. New constant text is sent to the scanner and beyond and new macro variables Y2 with a value of 2 and Y3 with a value of 3 are created in the scope of the macro COMPLEX. The fourth time trying to do the iterative DO pseudo instruction the macro value I is 4 and the condition is not met. We exit to instruction 6.

```
5 IDOEND ID 1
6 PUT LENG=6 _LOCAL_
```

This instruction is a PUT which means to write a character string to the log. In this case the macro name, macro variable names, and macro variable values of the local macro variables will be written to the SAS log.

```
COMPLEX I 4
COMPLEX Y1 1
COMPLEX Y2 2
COMPLEX Y3 3
```

The word scanner is still waiting for a character stream to send up to SAS. It calls the macro interpreter for an instruction or for constant text. It reads the MEND instruction and terminates the macro execution. The local symbol table disengages.

```
7 MEND
```

Macro is finished executing and the word scanner needs tokens from some other source, perhaps by the user typing in more text, more code from a SYSIN file, or a file INCLUDED by DMS.

MACRO VARIABLES, THEIR CREATION, AND THEIR SCOPE

We have discussed earlier automatic macro variables that are provided by SAS. Also macro variables are created globally with %LET statements in open code or with a %GLOBAL statement. These open code %LET and %GLOBAL statements immediately create macro variables, which are global.

When a parameter is used, for example, if we invoked the macro FIRST above with

```
%FIRST(100, TEST)
```

```
0 HDR FIRST key=0 pos=2 ver=8.00
  date=13998
1 PARM pos=1 PARM1
2 PARM pos=2 PARM2
```

as the header is processed, the parameter values are processed also. A local symbol table for the scope of the FIRST macro is created. PARM1 is macro variable name put in that local symbol table with a value of 100. The macro variable name PARM2 is also created in that local symbol table with a value of TEST. When the MEND instruction terminates the executing macro, the local symbol table is also disengaged. This local symbol table will no longer be accessed.

If we had invoked this macro FIRST with no parameters, for example,

```
%FIRST( )
```

as the header is processed, the macro variables PARM1 and PARM2 would be created in the local symbol table with null values, that is values with no characters and a length of 0. But because the macro parameters are instructions the local symbol tables are created.

In other words, macro parameters are always local to the macro that defines them; you can not make macro parameters global. If you tried for example inside your macro definition for FIRST,

```
%GLOBAL PARM1;
```

you would get a message,

```
ERROR: Attempt to %GLOBAL a name
(PARM1) which exists in a local environment.
```

This is because as the header instruction is executing the macro sets up the local symbol table for the parameters.

However, later, with a pseudo code with a left and right hand instruction for %LET on a compiled macro the macro compiler searches the current scope first for a macro variable name and then all other existing scopes. If it finds a match it uses that macro variable name and changes the

value in that scope. If the macro processor does not find the variable in any existing scope, it creates the macro variable in the local scope of the macro's current environment.

So using the pseudo code above for the compiled macro FIRST, if the pseudo code instruction generated for the %ELSE path is taken, the macro compiler would search the macro local symbol table for FRED. If it were not found in the local table then the next higher table would be searched. Finally, the global symbol table would be searched for FRED. If the macro variable name were found at any level the value would be changed at that scope level. However, if the macro variable name is not found it finally creates a value for FRED in the FIRST macro's local symbol table.

```
6 LETLEFT LENG=4 FRED
7 LETRIGHT LENG=3 500
```

MACRO VARIABLE EXAMPLES WITH THE DATA STEP

Grasping this creation of local scoping symbol tables and the fact that a local table is disengaged when the MEND pseudo code is reached, makes the behavior of CALL SYMPUT in the DATA step a bit easier to understand. CALL SYMPUT is a DATA step CALL routine that is accessed in the run-time phase of the DATA step compiler. CALL SYMPUT will create a macro variable value in the most local symbol table.

In open code, the most local symbol table is the global symbol table. So the following code would create a macro variable value TOUR with a value SEATTLE UNDERGROUND in the global symbol table.

```
DATA _NULL_;
  CALL SYMPUT ('TOUR', 'SEATTLE
  UNDERGROUND');
RUN;
```

If you put this text inside a macro definition

```
%MACRO INSIDE;
  DATA _NULL_;
    CALL SYMPUT ('TOUR', 'SEATTLE
    UNDERGROUND');
  RUN;
%MEND INSIDE;
```

the pseudo code would look like:

```
0 HDR INSIDE key=0 pos=0 ver=8.00
  date=13998
1 TEXT LENG=64 DATA _NULL_; CALL
  SYMPUT('TOUR','SEATTLE
  UNDERGROUND'); RUN;
3 MEND
```

The tokens of the constant text would be pushed to the scanner. Since there are no parameter instructions there would be no local symbol table created. As the tokens left the scanner for the DATA step compiler the most local table would be the global table. After this code compiles in the DATA step compiler and the execution of the DATA step compiler calls the SYMPUT routine. The most local symbol table is the global one. Once again the code would create macro variable value TOUR with a value SEATTLE UNDERGROUND in the global symbol table. The macro scope would be in the next most local symbol table unless a symbol table for the local scope had been created.

If we modify our macro definition to have a parameter, the parameter instruction would create a local symbol table.

```
%MACRO APARM (KEY=WORD);
  DATA _NULL_;
  CALL SYMPUT ('TOUR','SEATTLE
  UNDERGROUND');
%MEND APARM;
```

The compiled macro definition for APARM would have the following instructions:

```
0 HDR APARM key=1 pos=0 ver=8.00
  date=13998
1 PARM pos=0 KEY LENG=4 INIT=WORD
2 TEXT LENG=64 DATA _NULL_; CALL
  SYMPUT ('TOUR','SEATTLE
  UNDERGROUND'); RUN;
3 MEND
```

The execution of these instructions

```
%APARM ()
```

would create a local macro symbol table for APARM when the parameter instruction was processed.

In this case the local macro symbol table for the executing macro APARM would contain KEY with a value of WORD and TOUR with a value of SEATTLE UNDERGROUND. But when the MEND instruction is executed the local symbol table would be disengaged and these macro variable values would go away. Users would be unable to reference the macro variables after the MEND instruction executed.

So what would happen if the RUN DATA step statement were outside the example?

```
%MACRO OTRUN (KEY=WORD);
  DATA _NULL_;
  CALL SYMPUT ('TOUR','SEATTLE
  UNDERGROUND');
%MEND OTRUN;
%OTRUN ()
RUN;
```

The compiled instruction would have a parameter, which would create a local symbol table with a macro variable KEY with a value of WORD.

```
0 HDR OTRUN key=1 pos=0 ver=8.00
  date=13998
1 PARM pos=0 KEY LENG=4 INIT=WORD
2 TEXT LENG=59 DATA _NULL_; CALL
  SYMPUT ('TOUR','SEATTLE
  UNDERGROUND');
3 MEND
```

Constant text would be pushed to the scanner and tokens would be sent out to the DATA step. But the RUN token which makes the DATA step compiler complete compilation would not be included in the constant text. After the constant text is seen the word scanner calls the macro interpreter to get the next instruction of to get more constant text. In this case the MEND instruction executes, removing the local symbol table. The DATA step is still waiting for the RUN statement. When that RUN token reaches the DATA step, the DATA step compiler will complete its compilation and then execute. The DATA step compiler calls the SYMPUT routine. The most local symbol table is the global one. Once again the code would create macro variable value TOUR with a value SEATTLE UNDERGROUND in the global symbol table.

So timing is everything here. For CALL SYMPUT there are two compilers, macro and DATA step. A local symbol table may be created

when the macro instructions are executing or not. When the DATA step compiler is executing it calls SYMPUT to locate the most local symbol table.

COMPILE AND EXECUTION TIME MACRO QUOTING FUNCTIONS

Macro quoting functions are complicated and beyond the scope of this paper. Essentially, they are used to mask special characters. One easy example is to mask a semicolon so that it is not the end of a macro statement, but rather it is disguised until the character is beyond the word scanner. When it passes beyond the word scanner the masking is removed. For example, if you wanted to create a macro variable value PROC PRINT; RUN; containing semicolons, how does SAS know if the semicolon is the end of the %LET statement or part of the PROC PRINT statement. You could use %STR to mask the semicolons.

With macro quoting each special character can be masked with unprintable ASCII or EBCDIC characters, called delta characters in much of macro literature. As these masked special delta characters leave the word scanner, the unprintable ASCII or EBCDIC characters are replaced with the printable special character. Special delta characters are also used as delimiters to preserve leading and trailing blanks. For example,

```
%LET MACVAR=%STR( PROC PRINT;  
RUN; );
```

would internally mask the first two semicolons so they would not be considered the end of the %LET statement. Internally the global symbol table would store a macro variable name MACVAR with a value substituting unprintable delta characters to disguise the semicolons. Just to illustrate by using happy faces to symbolize the delta characters, the value stored in the global symbol table for MACVAR could be represented by

```
☺ PROC PRINT☺ RUN☺ ☺
```

Also note that the representative ☺ delta character is used as a delimiter for the leading and trailing blanks. Please note that the use of the ☺ is symbolic, depending on your operating system and how the OEM character sets are defined these characters may look like garbage

characters, blanks, or any unprintable representation.

So what are the differences between the timing of the compile-time macro quoting functions %STR and %NRSTR and the run-time quoting functions %QUOTE, %BQUOTE, %NRQUOTE and %NRBQUOTE?

In addition to what special characters are masked as the quoting functions execute, when the quoting happens is the key. %STR and %NRSTR do the macro quoting while the macro definition is compiling and the delta characters are present in the constant text. The other macro quoting functions execute when the macro is executing and not when it is compiling. For example, while the following macro definition compiles instructions, the compile-time %STR does macro quoting. The run-time %QUOTE does not do the macro quoting, but is part of the constant text.

```
%MACRO TIMING;  
  %LET A = %STR( DATA A; X=1; RUN;);  
  %LET B = %QUOTE( DATA B; X=1;  
                  RUN;);  
%MEND;
```

Looking at the generated instructions, the %LET statement builds the macro variable value right hand pseudo code instruction until the first semicolon. In the case of the macro variable name A the %STR masks the semicolon at compile-time and the value is complete as the user intended. In the macro variable name instruction for B the quoting for %QUOTE will be done at run-time. The variable value will be truncated at the first semicolon.

```
0 HDR TIMING key=0 pos=2 ver=8.00  
  date=13998  
1 LETLEFT LENG=1 A  
2 LETRIGHTLENG=19 ☺ DATA A☺ X☺1☺  
  RUN☺  
3 LETLEFT LENG=1 B  
4 LETRIGHTLENG=15 %QUOTE( DATA B  
5 TEXT LENG=12 X=1; RUN;);  
6 MEND
```

When this macro executes the local symbol table will have macro variables and values:

```
A      ☺ DATA A☺ X☺1☺ RUN☺  
B      ☺ DATA B
```

and the constant text sent to SAS from the word scanner will probably be a surprise to the user:

```
X=1; RUN;);
```

This constant text will create 180 errors from the parser since no DATA step is compiling.

MACRO EXAMPLES WITH SAS/CONNECT

The word scanner processes all character streams. If a character stream on a local client is being processed for a buffer for RSUBMIT to a server, then the RSUBMIT code turns off the recognition of ampersands and percents as macro triggers.

Invoking the server side using an INITSTMT creates a global macro variable X with a macro variable value

```
HELLO FROM THE SERVER
```

on the server when SAS initializes.

```
SAS -DMR -INITSTMT "%LET X=HELLO  
FROM THE SERVER;" -COMMID TCP
```

Invoking the client on the local side:

```
SAS -COMMID TCP -SET #
```

where number is the TCP/IP remote port number and submitting the following statements.

```
OPTIONS REMOTE=MACHINE;  
SIGNON MACHINE;  
RSUBMIT;  
%LET SERVER = ON THE REMOTE  
MACHINE;  
ENDRSUBMIT;  
%LET CLIENT= ON THE LOCAL MACHINE
```

signs on to the server machine when the SIGNON token is seen. When the token RSUBMIT leaves the scanner on the client side it signals CONNECT to begin to build a buffer to send to the server side. The CONNECT code turns off the macro triggers ampersand and percent on the client side. So CONNECT builds a buffer without using macro. The buffer would be:

```
%LET SERVER = ON THE REMOTE  
MACHINE;
```

This buffer would be sent from the client to the server. On the server the buffer would be sent as source code to the word scanner on the server. The server scanner would see the percent trigger and process the open code %LET on the server creating a second macro variable.

On the server the INITSTMT made the global macro variable X and now the %LET will create the macro variable SERVER.

```
X          HELLO FROM THE SCANNER  
SERVER    ON THE REMOTE MACHINE
```

On the client side there would be a macro variable value created by the %LET open code:

```
CLIENT    ON THE LOCAL MACHINE
```

However, if on the client side we had put the above code into a macro definition and then invoked the macro, different behavior would happen. Consider the same code inside a macro definition on the client side:

```
%MACRO FORSERV;  
  OPTIONS REMOTE=MACHINE;  
  SIGNON MACHINE;  
  RSUBMIT;  
    %LET SERVER = ON THE REMOTE  
      MACHINE;  
  ENDRSUBMIT;  
  %LET CLIENT= ON THE LOCAL  
    MACHINE;  
%MEND FORSERV;
```

The compiled instructions are on the client side something like:

```
0 HDR FORSERV key=0 pos=0 ver=8.00  
  date=13998  
1 TEXT LENG =52 OPTIONS REMOTE  
  MACHINE; SIGNON MACHINE; RSUBMIT;  
2 LETLEFT LENG =6 SERVER  
3 LETRIGHT LENG=21 ON THE REMOTE  
  MACHINE  
4 TEXT LENG =12 ENDRSUBMIT;  
5 LETLEFT LENG =6 CLIENT  
6 LETRIGHT LENG=21 ON THE LOCAL  
  MACHINE  
7 MEND
```

When this macro is invoked where it is on the client side with

```
%FORSERV
```

the constant text for the `OPTION` statement and `SIGNON` would be sent up to the `CONNECT` code. The `RSUBMIT` would try to build a buffer to send to the server and would turn off the macro triggers in the scanner for percent and ampersand. The next thing the word scanner would see is the pseudo instruction to make the left and right hand side of the `%LET` on the client, which would create a local macro variable value on the client for macro name `SERVER`. This is probably not what the user intended. Then the constant text would send up the token `ENDRSUBMIT` which would turn off the flag to ignore the macro triggers. There would be no tokens in the buffer for `CONNECT` to send to the server. Next the client would build the local macro variable `CLIENT` in the local scope for the macro environment `FORSERV` on the client. The word scanner will next process the pseudo instruction `MEND` that will close the macro and disengage the local macro symbol table for `FORSERV`.

To a user who did not have a sense of macro internals it might look as if the code worked fine until it was put inside a macro definition. Once inside the macro definition, this case does not build a buffer, does not create macros on the client in open code and creates local macro variables, which are not available, when the macro finishes execution. With a little knowledge of macro internals, one can see what is happening inside the macro.

SESSION COMPILED, AUTOCALL, COMPILED-STORED

Once you understand that a compiled macro looks like instructions to be read by the macro compiler, it is easy to understand what the compiled-stored macro looks like.

All the macros we have discussed are session-compiled macros; that is they are compiled in the session, stored in `WORK.SASMACR` and deleted from `WORK` at the end of the SAS session.

Autocall macros are the same as session compiled macros except that the source code for autocall macros are stored in files, read in as

source, and compiled before being stored in `WORK.SASMACR`. Then the autocall macro is automatically invoked after the compilation. Inefficiently written autocall macros could contain several macro definitions which would all be compiled before the macro called is invoked. After the session, the compiled autocall macro is deleted.

So you could see that saving the compiled macro might save time in production code since time would not be spent making the pseudo instructions and constant text in the compilation phase. It is crucial to save the macro definitions that created your compiled macros.

```
LIBNAME THERE 'pathname';  
OPTION MSTORE SASMSTORE=THERE;  
%MACRO SAVE/STORE;  
  DATA A; X=1; RUN;  
%MEND SAVE;
```

would save a compiled macro `SAVE` in a catalog `THERE.SASMACR` and in a subsequent session could be invoked with

```
LIBNAME THERE 'pathname';  
OPTION MSTORE SASMSTORE=THERE;  
%SAVE
```

The decision to use session compiled, autocall or compiled stored macro will vary with your applications and needs, but understanding how they look internally may help you make that decision, along with convenience and benchmarks.

CONCLUSION

As you write or revise macros, it is useful to understand what is happening in open code and what tokens you want to generate in the word scanner for the rest of the SAS System to use. It may be useful to consider what is happening as a macro definition is compiling and to imagine the pseudo instructions. Knowledge about the difference between compiled instructions and open code could help you with timing issues when writing or debugging macros.

ACKNOWLEDGEMENTS

I would like to thank my current manager, Amitava Ghosh, Director of Advanced Server Development, for his support regarding my

previous commitment to this technical macro paper.

Thanks also go to my predecessor in SAS macro design and development who contributed (wittingly or not) to macro internals as reflected in this paper: Robert Cross, Doug Cockrell, and Bruce Tindall.

REFERENCES

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

SAS and all other SAS Institute Inc. product or service names are registered trademarks or trademarks of SAS Institute Inc. in the USA and other countries. ® indicates USA registration. All other brand names and product names are trademarks of their respective companies.

OS/2® is a registered trademark of International Business Machines Corporation. Other brand and product names are registered trademarks of their respective companies.

HP/2® is a registered trademark of Hewlett-Packard Company.