Secrets of Macro Quoting Functions – How and Why

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

ABSTRACT

How, why, when and where to use macro quoting functions are tricky questions for all users. Macro quoting is not, as some think, putting single and double quotation marks in or around SAS tokens. Macro quoting is actually masking special characters and mnemonic expressions in the SAS® System so that these characters are not confused or ambiguous in their context during tokenization and execution of SAS or SAS macro. SAS macro quoting functions also are used to mark starting and ending delimiters.

The terms compile-time and run-time should be understood as related to macro for mastery of macro quoting functions. The differences and timing of the compile-time macro quoting functions, %STR and %NRSTR, will be emphasized. The meaning of run-time macro quoting functions and the individual implications in your code will be covered and the differences between run-time and compile-time macro quoting functions will be illustrated.

Details and history of %QUOTE and %BQUOTE may help users remember the differences in what is masked in each function. Also, the implementation of macro quoting functions with the NR-prefixes will be detailed.

Macro masking of special characters is achieved by using "delta" characters, unprintable ASCII or EBCDIC characters. Just what these characters are and how to examine them and debug them will be covered. Mnemonic expressions are macro quoted during explicit or implicit evaluation.

For debugging purposes, suggestions will be made about when macro quoting is stripped.

INTRODUCTION

In this paper I will use the emoticon ☺ to symbolize macro delta characters in macro quoting. The use of ☺ is just a symbol of delta characters to illustrate the position of a macro delta character in macro quoting.

I want to briefly define the term compiler and relate it 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 code 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 because it is in the SAS macro facility that macro quoting occurs. For the purpose of illustration, I will create a simple macro pseudo code to suggest compiled macro instructions and constant text.

A macro definition is the code found between the %MACRO statement and the %MEND statement. In one compilation phase with two passes, the SAS macro facility translates all statements inside a SAS macro definition into compiled instructions or constant text. This is referred to as compile-time and is completed with the %MEND statement. Later, when the compiled macro is invoked, the macro facility executes or runs these instructions in another phase, the run-time phase.

Macro compilation creates instructions and constant text and this is a compile-time phase. Macro execution runs or executes the instructions in a run-time phase. Macro quoting functions perform the macro quoting at run-time,

except for the compile-time macro quoting functions %STR and %NRSTR.

Finally, the last term to clarify is tokenization. Tokenization is the breaking of a character string into the smallest independent unit, a token that has meaning to the programming language.

WHY WE QUOTE TOKENS

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 string literals. There are also subtypes of token types, for example, double or single quoted strings in string literals. Some examples of token types include:

- String literals - "O'Connor", 'This is easy', '54321'x, '010111010101'b
- 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 - =, *, !, %, &, /, ;, +, “, ‘

Basic SAS tokens are then processed by different parts of SAS with different compilers, different grammars, and parsers. Identical tokens may have different meanings in the macro compiler than in other parts of the SAS System.

The reason we may need macro quoting is to avoid confusion and ambiguity with what token types we intend to use. For example, do we want the token OR to mean an abbreviation for the state of Oregon or the mnemonic logical operator OR? Do we want a semicolon to end a macro statement or a PROC statement? Is the name O'Connor an unclosed single quoted string or the Irish surname?

One of the simplest examples used to explain macro quoting is:

PROC PRINT; RUN;

This example contains two SAS statements ending in semicolons. By definition SAS statements end in semicolons. Confusion comes in because macro statements end in semicolons also. If we want to make these tokens including the semicolons into a macro variable value with a %LET statement, we would have confusion. The %LET statement begins with %LET and ends with a semicolon. Let's try:

%LET EXAMPLE= PROC PRINT; RUN;;

This code would create a macro variable EXAMPLE with a value of PROC PRINT in the macro symbol table, the storage location of macro variables and their values:

EXAMPLE PROC PRINT

The first semicolon would be consumed by the %LET when it ends at the first semicolon. The tokens RUN and the other semicolons would be left over tokens that would be sent up to the rest of the SAS System from the word scanner where they could create a 180 ERROR.

To make this example work as we intended, we need to mask or macro quote the two semicolons.

%LET EXAMPLE=%STR(PROC PRINT; RUN;);

Internally, this would mask the two semicolons in question, while they are inside the macro facility. The macro name would be EXAMPLE and the value internally would look like:

☺ PROC PRINT☺RUN☺☺

So the macro symbol table would contain:

EXAMPLE ☺ PROC PRINT☺RUN☺☺

The ☺ represents macro delta characters. Notice that starting and ending delimiters are macro quoted as well as the two semicolons.

When the macro variable name was referenced later in the SAS session:

&EXAMPLE

the delimiters are removed and the printable special characters replace the unprintable delta characters as each token is sent out of the word scanner:

PROC PRINT;RUN;

One helpful hint to see the delimiters when debugging macro quoting is to place an asterisk adjacent to the macro variable name in a %PUT statement. For example,

%PUT *&EXAMPLE*;

will print the following to the SAS log so you can see the starting and ending delimiter marked by the asterisk:

* PROC PRINT;RUN;*

HOW IS MACRO QUOTING DONE?

Inside the macro facility we have a set of tables that allow the macro subsystem to translate special characters into delta characters and to translate delta characters into special characters. Changing a special character to a delta character is macro quoting. Changing a delta character back to its original character is unquoting.

The characters to be masked with delta characters may be either special characters, such as ampersand, percent, and semicolon, or delimiters surrounding a series of tokens, telling the macro processor to do something special with the delimited tokens.

Characters used in SAS come from ASCII or EBCDIC code tables with decimal (and corresponding hex) values. For example, the upper case A in ASCII is represented with the decimal value 65 (hex 41) and in EBCDIC is represented with the decimal value 193 (hex C1). In ASCII the semicolon is represented with the decimal value 59 (hex 3B) and in EBCDIC the semicolon is represented with the decimal value 94 (hex 5E). These are printable characters in the code tables, but there are unprintable characters too.

The delta characters are unprintable characters available for use in ASCII or EBCDIC character sets. The unprintable characters available for use as delta characters in ASCII are those with decimal representations 1-8, 11, 14-26, 28-31, and 127-255. On EBCDIC machines the characters that may be used as deltas are 1-7, 16-23, 28-36, 38-39, 43-46, and 50.

We define these available unprintable characters as delta characters. Some unprintable characters are mapped as delimiters: %STR/%NRSTR start and stop, %QUOTE start, %NRQUOTE start, %BQUOTE start, %NRBQUOTE start, %UNQUOTE stop, and stop macro function. We also map the unprintable characters as delta characters to represent special characters that could be ambiguous in the SAS System:

DELTA SEMICOLON	;
DELTA AMPERSAND	&
DELTA PERCENT	%
DELTA SINGLE QUOTATION	’
DELTA DOUBLE QUOTATION	"
DELTA OPEN PARENTHESIS	(
DELTA CLOSE PARENTHESIS	)
DELTA PLUS	+
DELTA MINUS	-
DELTA STAR	*
DELTA SLASH	/
DELTA LESS THAN	<
DELTA GREATER THAN	>
DELTA NOT	^
DELTA EQUAL	=
DELTA OR	|
DELTA COMMA	,
DELTA TILDE	~

For example, to represent a DELTA SEMICOLON we map the unprintable ASCII decimal character 14 (hex 0E) as a DELTA SEMICOLON and we map the unprintable EBCDIC decimal character 38 (hex 26) as a DELTA SEMICOLON.

On most operating systems these unprintable characters if reflected on the monitor will look like garbage characters. Sometimes, machine specific items, such as the glyphs for an OEM or how the monitor itself is set, may define unprintable special characters as unexpected surprises, for example, a happy face. Regardless, how these "unprintable" characters are reflected on your term monitor should not be a concern. Internally the macro processor knows that these unprintable delta characters are masking special characters.

For debugging purposes there are ways to see these delta characters which may look like garbage characters and ways to have them converted to the printable characters they represent, unquoting. One way to detect a delta character at your monitor is to use OPTION MLOGIC. On the other hand, OPTION SYMBOLGEN strips delta characters and converts them to the special characters. %PUT

does convert the delta characters to printable characters. However, %PUT _USER_ and the other underscore %PUT versions do not convert the delta characters when printing to the log. But enough of the tedious internal details. For simplicity's sake, I will use the emoticon ☺ to illustrate those tedious unprintable delta characters, instead of using decimal or hex values.

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.

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. The word scanner is also the main location in the system where the macro facility is called. The word scanner knows how to handle delta characters and treats them as tokens when appropriate.

TRIGGERS IN THE SCANNER

The macro facility is triggered in the word scanner as it is looking at each character in a string of incoming code and the scanner sees a percent sign or an ampersand.

Consider the following incoming code, which will be sent on as tokens from the word scanner to TITLE processing. The developer wants the TITLE to reflect the date with format ddmmmyyyy.

```
TITLE "&SYSDATE9";
```

As this string of code comes into the scanner each character is examined. The first token TITLE, a word token, is sent on up to the global statement handler of the SAS System.

The next character is examined. It is a double quotation mark and the scanner is going to build double quoted string until is locates a closing double quotation mark. The scanner also is going to honor the macro triggers, percent and ampersand, while building the tokens inside this double quoted string. If, instead of a double quoted string, a single quoted string was being built, the scanner would "turn off" the macro triggers, percent and ampersand.

In this case, after the double quotation mark, the next character 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 SYSDATE9. 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.

In this case, the date the session was initialized is substituted and passed to the scanner by the macro symbolic substitution.

```
31OCT1999
```

(The new macro variable SYSDATE9 has been back ported from 7.01 to version 6 releases of SAS at the request of users after SUGI 24.)

The scanner uses these new characters and continues building the token:

```
"31OCT1999
```

When it locates the closing double quotation mark it passes the token "31OCT1999" up to the TITLE processing. This token is a string literal type with a subtype of double quotation.

The scanner examines the next character, which is the special token semicolon, and it passes it up to the TITLE processor. The semicolon signals the end of the TITLE statement and the TITLE is created.

The SAS word scanner is recursive and repeatedly calls into itself to substitute strings of code and to 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 this happens is inside single quoted strings. For example, the macro triggers would be "turned off" and no resolution would take place inside:

TITLE '&SYSDATE9';

Another case where the macro triggers are "turned off", while not in the scope of this paper, is when SAS/CONNECT® is building a buffer on the client after it has seen RSUBMIT.

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.

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.

MACRO COMPILATION – A SIMPLE CASE

Macro compilation is the process 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. This is compile-time.

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

The open code handler calls the word scanner for the next token, SIMPLE, and makes sure it is a valid macro name. 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
1 TEXT leng=20 DATA A; X=1; RUN;
2 LETL leng=4 NAME
3 LETR leng=4 FRED
4 MEND
```

These instructions are written as five records in an object SIMPLE in the SAS macro catalog in WORK.SASMACR. This represents compiled code.

%STR COMPILE-TIME FUNCTION

The simplest macro quoting function to describe is %STR. Whenever %STR occurs, even inside macro compilation, it turns off all macro functions and uppercasing and collects everything between the parentheses, matching pairs of parentheses. During collection, nesting of parentheses is honored. The very first closing parenthesis seen is not necessarily the one that ends the function. For example,

```
%STR(((FOO)))
```

ends at the third close parenthesis.

There is no macro resolution yet, but we reduce the double special characters used to create single quotation marks, parentheses and percents. %STR reduce any double special characters such as %", %% and %) to the single special character ", % and) respectively. In other words with %STR or %NSTR if you have an unmatched quotation mark or parenthesis, place a percent sign before that character. Use double percent signs to produce a single percent sign. For example, the following %STR code:

```
%PUT %STR(1.%) O%'CONNOR->
MACGRORY->CARROLL);
```

would first internally mask the closing parenthesis and the single quotation mark that were indicated above by the percent signs.

```
1.☺ O☺CONNOR->MACGRORY
->CARROLL
```

The building of the string is complete when the matching closing parenthesis is found. The delta quoting is applied, in this case to the minus and greater than sign. Delimiter deltas marking %STR start and %STR stop are placed where the opening and closing parenthesis were to preserve leading and trailing blanks. Internally it looks like:

```
☺1.☺ O☺CONNOR☺☺MACGRORY
☺☺CARROLL☺
```

The macro quoted string is pushed back on the scanner and the macro functions are turned on. The following would be printed to the log when the delta characters are replaced with the printable characters:

```
1.) O'CONNOR->MACGRORY->CARROLL
```

For another example, without %STR this %LET statement would end with the first semicolon.

```
%LET X = %STR(THIS IS A ;  I LIKE IT);
```

Using the emoticon to illustrate %STR, it I stored in the macro symbol table:

```
X       ☺THIS IS A ☺ I LIKE IT☺
```

When debugging the %PUT statement:

```
%PUT *&X*;
```

would strip the macro delta characters masking the semicolons and print the following to the log:

```
*THIS IS A ; I LIKE IT*
```

When using OPTIONS SYMBOLGEN, stripping of the macro delta characters would produce the debugging SYMBOLGEN information and a second additional NOTE to the log that unquoting had occurred:

```
SYMBOLGEN:  Macro variable X resolves to  THIS IS A ; I
LIKE IT

SYMBOLGEN:  Some characters in the above value, which
were subject to macro quoting, have been unquoted for
printing.
```

One last %STR example is for comments of the /* comments */ type. The word scanner normally consumes these. So if you wanted to %PUT them to the log the following code:

```
%PUT OUT /* IN */ OUT;
```

would print to the log consuming the inside comment:

```
OUT  OUT
```

To mask the meaning of the /* comment */ style comments, you could mask the meaning in the scanner using %STRs around the special characters / and *, for example %STR(/)STR(*):

```
%PUT  OUT %STR(/)%STR(*) IN
%STR(*)%STR(/) OUT;
```

Internally to the scanner it would mask beginning and ending delimiters and the special characters, looking like:

```
OUT ☺☺☺☺☺☺ IN ☺☺☺☺☺☺ OUT
```

would print to the log:

```
    OUT /* IN */ OUT
```

since the meaning of the comments has been removed.

%NRSTR COMPILE-TIME FUNCTION

%NRSTR behaves the same way that %STR works except it does *not resolve* the % and & but "turns off" the macro triggers. In macro, most

quoting functions come in pairs. One pair has a prefix of NR which "turns off" the macro triggers, ampersand and percent, so that they will not resolve. Internally the percent and ampersand are replaced with delta characters. For example, you might want a TITLE statement to illustrate the behavior of SYSDATE9 and to include the tokens &SYSDATE9 themselves. The single quotation mark string literal example suggested earlier would not work because you want SYSDATE9 to both resolve and not resolve in one TITLE statement. You might use %NRSTR inside a double quotation string literal.

```
TITLE "USE %NRSTR(&SYSDATE9) TO
GET THIS FULL DATE &SYSDATE9";
```

This would create a string literal in the word scanner:

```
"USE ☺☺SYSDATE9☺ TO GET THIS FULL
DATE 31OCT1999"
```

and as the string literal leaves the word scanner the delta characters are replaced with the special characters they represent and the delimiters are honored.

```
“USE &SYSDATE9 TO GET THIS FULL
DATE 31OCT1999”
```

WHAT COMPILE-TIME MEANS

The macro quoting functions %STR and %NRSTR take affect during compile-time. All other macro quoting functions take effect during run-time. This is a distinction which is important and may best be illustrated with an example and some pseudo code of a compiled macro. Remember that the %LET macro statement ends in a semicolon as you examine the macro definition EXAMPLE. Also remember that macro quoting only occurs during compilation for %STR and %NRSTR.

During the compilation phase, instructions and constant text are written to WORK.SASMACR from the %MACRO statement to the %MEND.

```
%MACRO EXAMPLE;
%LET S=%STR(DO:+=;&SYSDAY );
%LET N =%NRSTR(DO:+=;&SYSDAY );
%LET Q=%QUOTE(DO:+=;&SYSDAY );
%LET NRQ=
%NRQUOTE(DO:+=;&SYSDAY);
%LET BQ= %BQUOTE(DO:+ =;&SYSDAY
);
%LET NRB=
%NRBQUOTE(DO:+=;&SYSDAY );
%PUT _LOCAL_;
%MEND;
```

The pseudo code can be represented as follows.

```
0 HDR EXAMPLE date=14195 ver=8.00 pos=0
kwd=0
1 LETL leng=1 S
2 LETR leng=15 ☺DO:☺☺☺&SYSDAY☺
3 LETL leng=1 N
4 LETR leng=15 ☺DO:☺☺☺☺SYSDAY☺
5 LETL leng=1 Q
6 LETR leng=12 %QUOTE(DO:+=
7 TEXT leng=9 &SYSDAY);
8 LETL leng=3 NRQ
9 LETR leng=14 %NRQUOTE(DO:+=
10 TEXT leng=9 &SYSDAY);
11 LETL leng=2 BQ
12 LETR leng=13 %BQUOTE(DO:+=
13 TEXT  leng=9 &SYSDAY );
14 LETL leng=3 NRB
15 LETR leng=15 %NRBQUOTE(DO:+=
16 TEXT leng=9  &SYSDAY);
17 PUT leng=7 _LOCAL_
18 MEND
```

When you execute this macro you would get four ERROR 180 messages. Why? Look at the pseudo code again. Because the %LET values ended with the first semicolon, there are four constant text statements that create errors:

```
&SYSDAY);
```

Let’s turn on OPTIONS MLOGIC and execute EXAMPLE to debug and illustrate:

```
OPTIONS MLOGIC;
%EXAMPLE
 MLOGIC(EXAMPLE):Beginning execution.
MLOGIC(EXAMPLE):%LET (variable name is S)
MLOGIC(EXAMPLE):%LET (variable name is N)
MLOGIC(EXAMPLE):%LET (variable name is Q)
NOTE: Line generated by the macro variable "SYSDAY".
13    Thursday
        --------
        180
ERROR 180-322: Statement is not valid or it is used out of
proper order.
MLOGIC(EXAMPLE):%LET (variable name is NRQ)
NOTE: Line generated by the macro variable "SYSDAY".
13    Thursday
         --------
         180
ERROR 180-322: Statement is not valid or it is used out of
proper order.
```

```
MLOGIC(EXAMPLE):  %LET (variable name is BQ)
ERROR: Expected close parenthesis after macro function
invocation not found.
NOTE: Line generated by the macro variable "SYSDAY".
13     Thursday
       --------
       180
ERROR 180-322: Statement is not valid or it is used out of
proper order.
MLOGIC(EXAMPLE):  %LET (variable name is NRB)
ERROR: Expected close parenthesis after macro function
invocation not found.
NOTE: Line generated by the macro variable "SYSDAY".
13     Thursday
       --------
       180
ERROR 180-322: Statement is not valid or it is used out of
proper order.
MLOGIC(EXAMPLE):  %PUT _LOCAL_
EXAMPLE S ☺DO:☺☺☺Thursday☺
EXAMPLE N ☺DO:☺☺☺☺SYSDAY☺
EXAMPLE Q ☺DO:☺☺
EXAMPLE NRQ ☺DO:☺☺
EXAMPLE BQ
EXAMPLE NRB
MLOGIC(EXAMPLE):  Ending execution.
```

Looking at the local variables in the %PUT _LOCAL_ statement indicates that there are no values for the %BQUOTE and NRBQUOTE. The log reflects that the closing parentheses are missing and, therefore, the values were not created. There are values for %QUOTE and %NRQUOTE but not what we expected because the semicolons truncated the %LET values.

%QUOTE AND %NRQUOTE RUN-TIME FUNCTIONS

%QUOTE and %NRQUOTE are run-time macro functions. This means nothing happens during macro compilation. During macro compilation the effect is to treat the function as text. Looking at the pseudo code for EXAMPLE you see that the values for the right hand side of the %LET statements are truncated at the first semicolon.

For these paired functions, behavior will be identical except that percent and ampersand will be NOT RESOLVED for %NRQUOTE. When the argument for %QUOTE (and %NRQUOTE) is executed at run-time (or in open code) the delimiter for %QUOTE start is set when the first parenthesis is seen. Each special token coming out of the scanner will be set to the delta character defined for it. When the matching closing parenthesis is found the stop macro function delimiter replaces it.

%QUOTE and %NRQUOTE, like %STR and %NRSTR, use the percent sign to mask the single percent, parenthesis, and quotation mark. So in open code, using %QUOTE instead of %STR, in the example from above the quoting internally appears the same:

```
%PUT %QUOTE(1.%) O%'CONNOR->
MACGRORY->CARROLL);
```

In open code the macro quoting internally looks the same:

```
☺1.☺ O☺CONNOR☺☺ MACGRORY
☺☺CARROLL☺
```

The result on the log looks identical :

```
1.)  O'CONNOR-> MACGRORY->CARROLL
```

However if this had been inside a macro definition:

```
%MACRO TEST;
  %PUT %QUOTE(1.%) O%'CONNOR->
MACGRORY->CARROLL);
%MEND TEST;
```

the macro would not compile. Since the %QUOTE does not happen until the run-time the word scanner would be trying to build a single quoted string literal when it sees the single quote.

COMPILE-TIME VERUS RUN-TIME

One way to illustrate compile-time versus run-time is to write a macro that will print a WARNING sad face if there is no parameter:

```
WARNING==> ;-( missing parameter
```

or a NOTE happy face with the parameter, here indicated with the blank, like the following:

```
NOTE==>;-) a parameter: ___
```

Since in this contrived example we have a semicolon as the wink, we will need to macro quote that semicolon at compile-time with %STR so that it does not end the %PUT statement earlier than intended. But we do not know the value of the parameter to macro quote until run-time so we can use a %QUOTE that will quote at run-time:

```
%MACRO EX (P);
%IF %QUOTE(&P)= %THEN
   %PUT %STR(WARNING ==>;-%(  missing
parameter);
%ELSE
   %PUT %STR(NOTE==>;-%) a parameter:
&P);
%MEND EX;
```

To see the pseudo code gives us the idea of run-time versus compile-time quoting.

```
0 HDR EX key=0 pos=1 date=14056 ver=8.00
1 PARMAC pos? 1 PARM initvallen=0 init=
2 IF     falseaddr=5 leng=11 %QUOTE(&P)=
3 PUT leng=35 ☺WARNING☺☺☺☺☺☺
missing parameter☺
4 JUMP   addr=6 leng=0
5 PUT leng=25☺NOTE☺☺☺☺☺☺a parameter:
&P☺
6 MEND
```

In this pseudo code you can see that the %STR function quoted the special characters as the macro compiled. But the %QUOTE function in the %IF instruction will delta quote during execution.

Executing this code with invocations and copies of the log show that these run as intended:

```
%EX( )
WARNING==> ;-( missing parameter

%EX(HIGH-LOW)
NOTE==> ;-) a parameter: HIGH-LOW

%EX(^)
NOTE==> ;-) a parameter: ^
```

This is because the delta character substitution with %QUOTE happens as the macro executes. If we had not macro quoted at run-time and had the following %IF statement:

```
%MACRO EXBAD (P);
%IF &P= %THEN
   %PUT %STR(WARNING ==>;-%(  missing
parameter);
%ELSE
   %PUT %STR(NOTE==>;-%) a parameter:
&P);
%MEND EXBAD;
```

Note the pseudo code change:

```
0 HDR EXBAD key=0 pos=1 date=14056
ver=8.00
1 PARMAC pos? 1 PARM initvallen=0 init=
2 IF     falseaddr=5 leng=3 &P)=
3 PUT leng=35 ☺WARNING☺☺☺☺☺☺
missing parameter☺
4 JUMP   addr=6 leng=0
5 PUT leng=41☺NOTE☺☺☺☺☺☺a parameter:
&P☺
6 MEND
```

The evaluation at run-time of value of the parameter P would cause the following examples to have ERRORS during implicit evaluation:

```
%EX(HIGH-LOW)

%EX(^)
```

ERROR: A character operand was found in the %EVAL function or %IF condition where a numeric operand is required. The condition was: &P =

ERROR: The macro EXBAD will stop executing.

This is because during evaluation the substitution for &P is either HIGH-LOW or ^ which are not quoted to prevent unintended evaluations at run-time.

%BQUOTE AND %NRBQUOTE RUN-TIME FUNCTIONS

%BQUOTE and %NRBQUOTE are run-time macro functions. This means nothing happens during macro compilation. During macro compilation the effect is to treat the function like text. Like all paired macro quoting functions with the NR prefix, %NRBQUOTE does NOT RESOLVE percents and ampersands.

%BQUOTE and %NRBQUOTE are also more mature and robust since these macro functions treat parentheses and quotation marks as single characters during run-time.

How could it be especially useful to allow a single unmatched quotation? Consider a DATA step that creates a macro variable value with CALL SYMPUT:

```
DATA A;
WORD="O'CONNOR";
CALL SYMPUT('VAR', WORD);
RUN;
```

Executing this DATA step creates following macro variables in the symbol table:

VAR O'CONNOR

This macro variable value with the single quotation mark might cause problems with an unclosed single quote. But you want to use such a macro variable value with an unclosed quotation mark.

```
%MACRO ONE;
%IF %BQUOTE(&VAR) =  %THEN %PUT
THERE IS NO VALUE ;
%ELSE %PUT HELLO, %BQUOTE(&VAR);
%MEND;
```

The pseudo code for this looks like:

```
0 HDR  ONE  pos=0 key=0 date=14062
ver=8.00
1 IF     falseaddr=4 leng=15
%BQUOTE(&VAR) =
2 PUT  leng=17 THERE IS NO VALUE
3 JUMP addr=5 leng=0
4 PUT leng=26 Hello, %BQUOTE(&VAR)
5 MEND
```

Invoking this macro:

%ONE

would cause the macro variable VAR to use the value O'CONNOR. The use of the %BQUOTE run-time macro quoting function would allow the macro quoting of O☺CONNOR during run-time. The single quotation mark would not appear as an unclosed quote and would allow the macro to execute and print the name to the log when it unquotes the token for printing. If %QUOTE had been used, we would have a string literal with unmatched quotes and the macro would have stopped executing.

%UNQUOTE RUN-TIME FUNCTION

The %UNQUOTE run-time function during compile-time treats the arguments like text. During run-time it collects the tokens between the parentheses, letting macro items resolve. It throws away the non-translatable deltas such the delimiters. It replaces the other delta characters with the original special characters.

%UNQUOTE is most useful if something is macro quoted inside an executing macro and you want to use it in an unquoted manner. Another use is when the text "looks" right in the log but the generated text will not work as expected.

%SUPERQ RUN-TIME FUNCTION

%SUPERQ takes effect during run-time.
In macro compilation it is treated as text.

To use %SUPERQ reference the macro variable name with no leading ampersand. It immediately gets the macro variable value from the macro symbol table with a straight copy and not scanning in the scanner. %SUPERQ applies %NRBQUOTE behavior to the value. Since %SUPERQ does not attempt resolution of its argument there is no warning if the macro variable does not resolve. If you do not want the warning with %NRBQUOTE if something does not resolve then %SUPERQ may be the ticket.

For example consider:

```
DATA _NULL_;
CALL SYMPUT('WORD', 'BEN&JERRY');
RUN;
```

which would create a macro variable in the symbol table:

WORD BEN&JERRY

With the TITLE statement below:

```
TITLE "SEE YOU &SYSDAY AT &WORD";
```

You would get the warning:

WARNING: Apparent symbolic reference JERRY not resolved.

There would be no such warning with:

```
TITLE "SEE YOU &SYSDAY AT
%SUPERQ(WORD)";
```

Notice that no ampersand is used with %SUPERQ function.

%QUPCASE, %QSUBSTR, AND %QSAN RUN-TIME FUNCTIONS

%QUPCASE, %QSUBSTR and %QSCAN take place during macro execution and they are treated as text during compile-time. During run-time the functions are performed like uppercase,

substring or scan. Then, if this is the %Q version, the leading and trailing delimiters are preserved.

The key to remember here is that %UPCASE, %SUBSTR and %SCAN returns the unquoted results, even if the argument was quoted.

Let's look at a few examples. This:

%LET X=%NRSTR(%eval(1+2));

looks like the following internally when stored in the macro symbol table:

X ☺☺eval(1☺2)☺

Since the %PUT removes macro quoting:

%PUT &X;

would print to the log:

%eval(1+2)

If the debugging developer had turned on OPTION SYMBOLGEN, the debugging information would indicate that the macro quoting had been removed for printing:

SYMBOLGEN: Macro variable X resolves to %eval(1+2)

SYMBOLGEN: Some characters in the above value, which were subject to macro quoting, have been unquoted for printing.

Remember that %UPCASE macro function will return the unquoted result even if the value was macro quoted. So the following %UPCASE would print 3 to the log:

%PUT %UPCASE(&X);

While the %QUPCASE version would print the unquoted value to the log, %EVAL(1+2):

%PUT %QUPCASE(&X);

Consider another example:

%LET A=1;
%LET ABC=5;
%LET DEF=%NRSTR(&ABC);

Internally these would be stored in the macro symbol table as:

A 1
ABC 5
DEF ☺☺ABC☺

Since the %PUT removes macro quoting the %PUT statement below would print &ABC to the log but OPTION SYMBOLGEN would have indicated that macro quoting had been removed:

%PUT &DEF;

Remember that %SUBSTR macro function will return the unquoted result even if the value was macro quoted. So the following %SUBSTR would print 1 to the log:

%PUT %SUBSTR(&DEF,1,2);

This is because the %SUBSTR would substring &A which has the macro variable of 1 in the symbol table.

However, in the %QSUBSTR version the quoted results are returned and &A would be printed to the log since that is the quoted result:

%PUT %QSUBSTR(&DEF,1,2);

QUOTING MNEMONIC OPERATORS

The macro facility has mnemonic operators:
AND, EQ, GE, GT, LE, LT, NE, NOT, and OR.

These mnemonic operators can be used in explicit or implicit macro evaluation. %EVAL is the explicit macro evaluation for integer arithmetic and logical expressions. Implicit macro evaluation occurs in the following macro statements: iterative %DO, %DO %UNTIL, %DO %WHILE, and %IF %THEN.

If a macro quoting function is active during macro evaluation and a mnemonic operator is detected inside the macro quoting delimiters, the quoted mnemonic will not be treated as a mnemonic operator. Consider

%MACRO TEST(PARM);
%IF &PARM EQ %THEN %PUT MISSING PARAMETER;
%MEND TEST;

Invoking this with a mnemonic parameter such as OR:

%TEST(OR)

would confuse implicit evaluation in %IF. The ERROR messages would print to the log.

ERROR: A character operand was found in the %EVAL function or %IF condition where a numeric operand is required. The condition was: &VAL EQ.

ERROR: The macro will stop executing.

This is because internally it appears the implicit evaluation is comparing two mnemonic operators:

OR EQ

If you quoted PARM in the implicit %IF evaluation, at invocation a mnemonic like OR would be masked in the evaluation:

```
%MACRO TEST(PARM);
%IF  %QUOTE(&PARM) EQ %THEN
    %PUT MISSING PAREMATER;
%MEND TEST;
```

Internally the implicit evaluation would look like:

☺OR☺ EQ

which would mask the OR token so it would not look like a character mnemonic operator.

All macro quoting functions mask mnemonic operators only during explicit or implicit evaluation.

CONCLUSION

As you write or revise macros and as you use macro quoting functions, it is useful to understand what is happening internally. Unprintable delta characters replace special characters during macro quoting in open code, and at compile-time and run-time. These characters are restored to the special printable characters during unquoting. 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 compile-time and run-time 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.