Paper SM04

SAS Macro Quoting - A Look Behind the Scenes

Tim Lepp, Bayer AG, Berlin, Germany

ABSTRACT

Have you been cheating your way through Macro Quoting over the years? Didn't feel like really dealing with it? And let's be honest, somehow you always managed to do it in the end.

Even if this is somewhat unsatisfactory - for the pure user this approach is probably even sufficient. But at the latest, if you want to program really robust macros, you have to deal with the topic more intensively.

This paper is for those who really want to know what happens behind the scenes. The macro quoting functions which you will find well explained in most of the papers, will be touched only very briefly. The focus will be put more on the general interaction between macro processor and other functional units. And with that it is explained also the timing aspect of macro quoting. That is, when does happen what in compilation and execution phases.

INTRODUCTION

What is SAS® Macro Quoting and why do we need it? When you hear quoting, you might think it has something to do with quotation marks. And it does, but only figuratively. With "normal" programming languages, quotation marks are used to distinguish "text" from keywords, operators, etc. The SAS Macro Language, which is mainly a code generator/structuring tool, on the other hand, regards everything as text (also quotation marks itself).

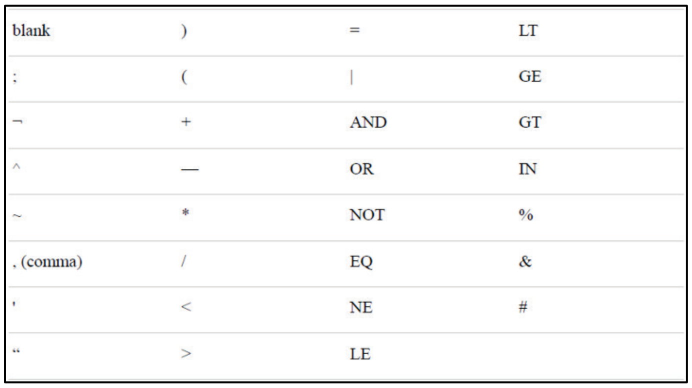
Example:

```
H&M, %Percent, g=9.8
```

Does this contain...

(a) A list of fashion labels?
(b) A set of macro parameters?
(c) The macro variable M, the macro call Percent()?

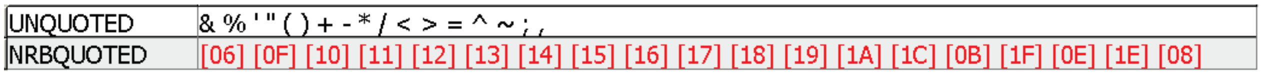
This is not easy even for us to recognize, if we do not know the context. That is why we need a more comprehensive approach for distinction. And there are of course other characters and mnemonics than those given in the example above, which could be misinterpreted by the macro processor. A complete list is given in Figure 1.



blank	)	=	LT
;	(	|	GE
¬	+	AND	GT
^	—	OR	IN
~	*	NOT	%
, (comma)	/	EQ	&
'	<	NE	#
"	>	LE	

Figure 1: Signs and mnemonics with operational meaning for the macro processor [1]

The principle approach of SAS Macro Quoting is now as follows: The characters listed in Figure 1 are temporarily hidden (masked) from the *Macro Processor* by replacing them with non-printable characters in the range 0x01 to 0x1F. This character area is located in the control characters area and can therefore vary a bit between operating systems. For a better understanding an example of two macro variables is given in Figure 2. The former contains a selection of special characters unmasked. The second masks the same special characters using the %NRBQUOTE function. Start and end bytes (here 0x06 and 0x08) denote the start and end ID bytes of the respective quoting function. Values in square brackets represent hexadecimal values of the respective non-printable characters.



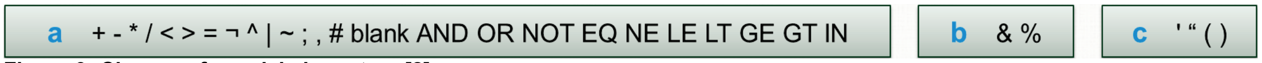
UNQUOTED	& % ' " ( ) + - * / < > = ^ ~ ; ,
NRBQUOTED	[06] [0F] [10] [11] [12] [13] [14] [15] [16] [17] [18] [19] [1A] [1C] [0B] [1F] [0E] [1E] [08]

Figure 2: Example of storage for unquoted and quoted values

MACRO QUOTING FUNCTIONS

This section is intended to give an overview of the commonly used quoting functions with their respective special features.

In general, the following three classes of characters are distinguished, which can be misinterpreted by the macro processor and must therefore be masked, if they are text.



a + - * / < > = ¬ ^ | ~ ; , # blank AND OR NOT EQ NE LE LT GE GT IN

b & %

c ' " ( )

Figure 3: Classes of special characters [2]

(a) operators, mnemonics, miscellaneous
(b) macro triggers
(c) unmatched characters

Let us stick to the example from the introduction and see what SAS makes out of it for the various quoting functions. We assume the variable `VAR0` is defined as follows:

```
DATA _NULL_; CALL symput('var0','H&M, O''Neill, %Percent, g=9.8'); RUN;
```

And for illustration purposes the macro variable `M` and the macro `%Percent` are defined as follows:

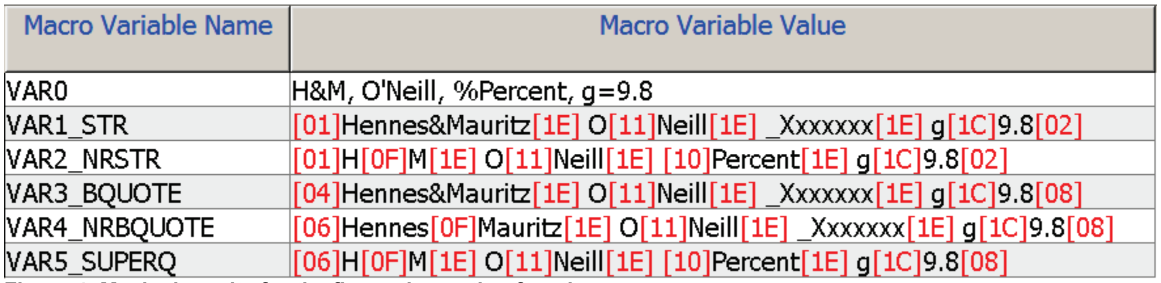
```
%LET M=ennes&Mauritz;
%MACRO Percent(); _Xxxxxxx %MEND;
```

Now we apply the five most important quoting functions to the variable `VAR0`[1]:

```
%LET var1_STR      =      %str(H&M, O%'Neill, %Percent, g=9.8); 2
%LET var2_NRSTR    =    %nrstr(H&M, O%'Neill, %Percent, g=9.8);
%LET var3_BQUOTE   =   %bquote(&var0.); 2
%LET var4_NRBQUOTE = %nrbquote(&var0.); 2
%LET var5_SUPERQ   =   %superq(var0);
```

[1] The % in front of the single quote in %STR and %NRSTR functions is needed, as it won't be quoted otherwise and lead to unbalanced strings. %QUOTE and %NRQUOTE cannot be used due to the unbalanced single quote.
[2] Warning messages are generated due to unknown variable references and macro invocations.

Following result is generated:



Macro Variable Name	Macro Variable Value
VAR0	H&M, O'Neill, %Percent, g=9.8
VAR1_STR	[01]Hennes&Mauritz[1E] O[11]Neill[1E] _Xxxxxxx[1E] g[1C]9.8[02]
VAR2_NRSTR	[01]H[0F]M[1E] O[11]Neill[1E] [10]Percent[1E] g[1C]9.8[02]
VAR3_BQUOTE	[04]Hennes&Mauritz[1E] O[11]Neill[1E] _Xxxxxxx[1E] g[1C]9.8[08]
VAR4_NRBQUOTE	[06]Hennes[0F]Mauritz[1E] O[11]Neill[1E] _Xxxxxxx[1E] g[1C]9.8[08]
VAR5_SUPERQ	[06]H[0F]M[1E] O[11]Neill[1E] [10]Percent[1E] g[1C]9.8[08]

Figure 4: Masked results for the five main quoting functions

In the left column you can see the macro variable name and in the right column the content as it is stored in the *Global Symbol Table*. The values in square brackets represent the non-printable hexadecimal characters mentioned in the introduction. Take a time looking at this figure, it already gives you a couple of insights.

Following a brief summary of the quoting functions with their properties:

%STR
Masks class **a** (see Figure 3) and class **c** if the character to be masked is preceded by a % as kind of escape character. In the above example the % in O%'Neill is placed before the single quote. The function has difficulties if there are unpaired parentheses or quotation marks in the argument. Often the session can only be restored by a session reset. It is a compilation phase function.

%NRSTR
Masks classes **a** and **b**. Class **c** is subject to the same restrictions as for the %STR function. The function also has difficulties if there are unpaired parentheses or quotation marks in the argument. It is a compilation phase function.

%BQUOTE
Masks classes **a** and **c** without restrictions. Also unpaired brackets or quotation marks are handled. The function works in execution phase.

%NRBQUOTE
Masks classes **a**, **b** and **c**. However, it should be noted that for the macro triggers (class b) the function does not immediately mask these characters like %NRSTR, but the macro triggers are resolved as far as possible and only masked at the very end when no resolution is possible anymore. This is shown in Figure 4, VAR4_NRBQUOTE - &M is resolved to Hennes&Mauritz and as &Mauritz does not exist, the & is masked with [0F]. The function works in execution phase.

%SUPERQ
Masks classes **a**, **b** and **c**. A special feature here is that the function expects the name of a macro variable as argument, without the preceding &. The content of this macro variable, which must exist, is then masked without further resolution. This is where the function differs from %NRBQUOTE. Comparing the results in Figure 4, one can see that %SUPERQ is the actual sister function to %NRSTR at the time of execution (apart from the behavior concerning the paired single quotes which is explained below).

%UNQUOTE

Unmask classes **a**, **b** and **c**. It can be helpful if, for example, you want to hide something from the macro processor in the compilation phase, but want to make it visible again in the execution phase.

%QUOTE / %NRQUOTE

The functions mask equivalent to %STR and %NRSTR but at the time of execution. I.e. they have the same restrictions concerning the class **c** characters and difficulties with unpaired brackets and quotation marks. This means there are no advantages compared to using %BQUOTE and %NRBQUOTE. The functions actually mainly exist for compatibility reasons and do not require further attention. Apart from the very theoretical scenario when we want to keep a part of a string inside single quotes unresolved but the rest of the string quoted (see below).

' ' (SINGLE QUOTES)

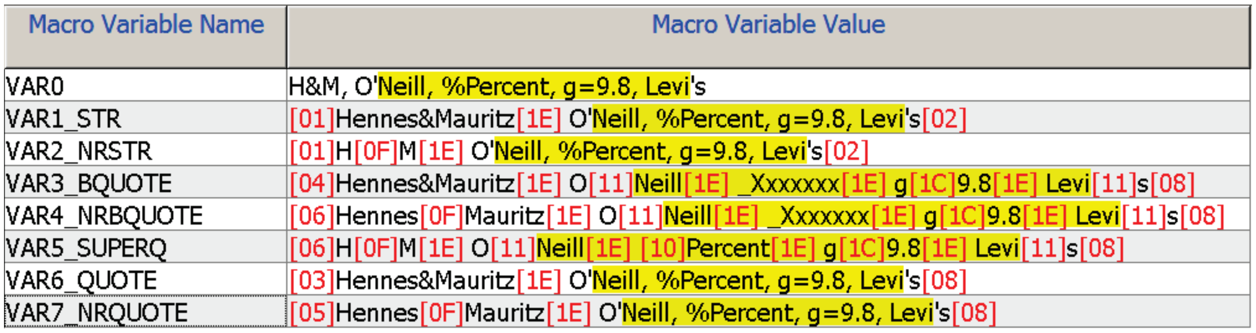
As the subject does not seem complicated enough yet, we also need to know about the effect of the single quote in that context.

The single quote is actually not a quoting function, but rather a kind of compile time directive (when it comes in pairs). It is listed here because it works similar to the %NRSTR function in the way that it prevents resolution of macro triggers (class **b**) but without actually quoting something in the sense of replacing characters. Let us add another label to the above example to illustrate the effect:

```
DATA _NULL_; CALL symput('var0','H&M, O''Neill, %Percent, g=9.8, Levi''s'); RUN; 3

%LET var1_STR       =      %str(H&M, O'Neill, %Percent, g=9.8, Levi's); 4
%LET var2_NRSTR     =    %nrstr(H&M, O'Neill, %Percent, g=9.8, Levi's);
%LET var3_BQUOTE    =   %bquote(&var0.); 4
%LET var4_NRBQUOTE = %nrbquote(&var0.); 4
%LET var5_SUPERQ    =   %superq(var0);
%LET var6_QUOTE     =    %quote(&var0.); 4
%LET var7_NRQUOTE   =  %nrquote(&var0.); 4
```

The resulting output is:



Macro Variable Name	Macro Variable Value
VAR0	H&M, O'Neill, %Percent, g=9.8, Levi's
VAR1_STR	[01]Hennes&Mauritz[1E] O'Neill, %Percent, g=9.8, Levi's[02]
VAR2_NRSTR	[01]H[0F]M[1E] O'Neill, %Percent, g=9.8, Levi's[02]
VAR3_BQUOTE	[04]Hennes&Mauritz[1E] O[11]Neill[1E] _Xxxxxxx[1E] g[1C]9.8[1E] Levi[11]s[08]
VAR4_NRBQUOTE	[06]Hennes[0F]Mauritz[1E] O[11]Neill[1E] _Xxxxxxx[1E] g[1C]9.8[1E] Levi[11]s[08]
VAR5_SUPERQ	[06]H[0F]M[1E] O[11]Neill[1E] [10]Percent[1E] g[1C]9.8[1E] Levi[11]s[08]
VAR6_QUOTE	[03]Hennes&Mauritz[1E] O'Neill, %Percent, g=9.8, Levi's[08]
VAR7_NRQUOTE	[05]Hennes[0F]Mauritz[1E] O'Neill, %Percent, g=9.8, Levi's[08]

Figure 5: Masked results showing the effect of single quotes for the different quoting functions

One can see that only the functions %BQUOTE, %NRBQUOTE and %SUPERQ are actually masking content inside the single quotes (yellow highlighted), because the quotes themselves are masked and therefore lose their special meaning. All others do not mask anything inside the single quotes, but also do not resolve the %Percent macro call.

[3] H&M™, O'Neill™, %Percent™, g=9.8™, Levi's™

[4] Warning messages are generated due to unknown variable reference.

This functionality may solve certain quoting problems for instance when we deal with `CALL EXECUTE`, but may also lead to unexpected results and make the whole topic even more difficult. Please see also two interesting papers about the behavior and usage together with the dequote function [3], [4].

%QSCAN / %QSUBSTR / %QUPCASE / %QSYSFUNC / %QCMPRES / %QLOWCASE / %QLEFT / %QTRIM
For the sake of completeness these macro functions and autocall macros should also be mentioned. These primarily of course have a different purpose, but mask the result in the same way as %NRBQUOTE.

PROGRAM FLOW

The following section is supposed to serve as general introduction to the program flow and in particular explain the relationship between the *Macro Processor* and the *Data Step Compiler*.

Figure 6 contains a general representation of the functional units involved in the program flow. On the left side are the input blocks. These can be normal SAS - or SCL programs, all kinds of batch or interactive submits and the like. The code from these input blocks lands successively on the *Input Stack*. This in turn is read word by word or token by token, which is the smallest piece of meaning within the code, by the *Word Scanner*. The *Word Scanner* in turn analyzes the code and forwards it to the appropriate functional unit for compilation and execution.

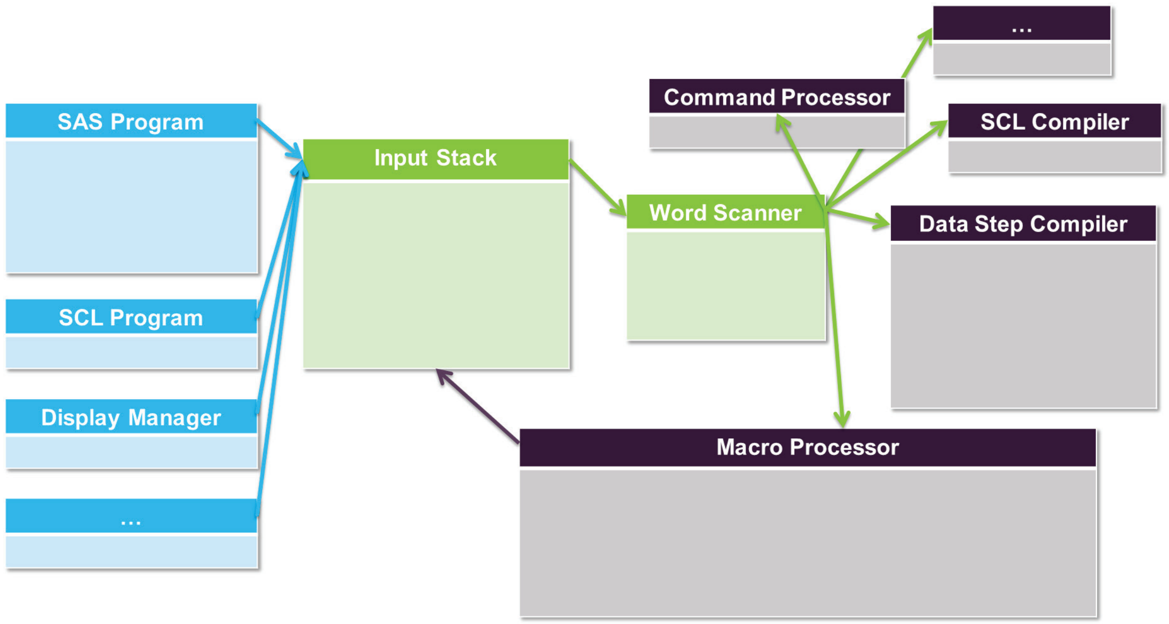


Figure 6: Schematic diagram of functional units

The figure shows that the *Macro Processor* is the only functional unit back-connected to the *Input Stack*. Let us take a closer look at the program flow using the following small example program.

```
DATA today;
    date = "&sysdate9";
RUN;
```

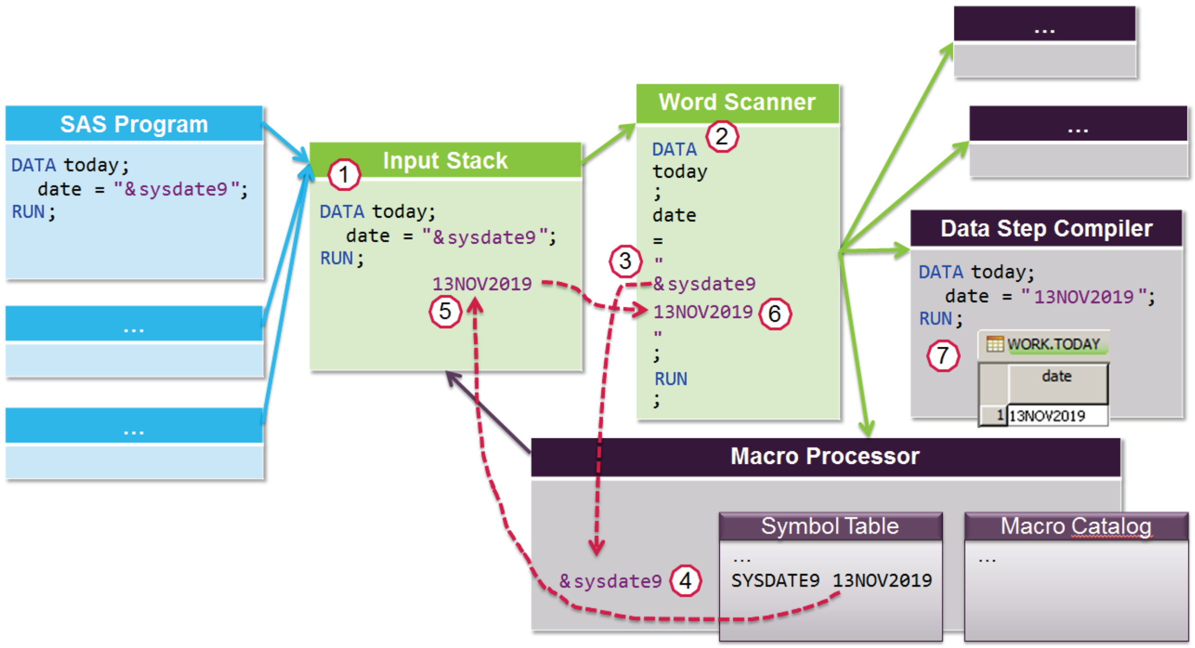


Figure 7: Program flow using the example program

In the first step (**1**) our example program is loaded onto the *Input Stack*. Then the *Word Scanner* starts its work and reads the *Input Stack* token by token (**2**). Read tokens are removed from the *Input Stack*. The *Word Scanner* now recognizes the code pieces as data step code and forwards them to the *Data Step Compiler* for compilation. It does this until it encounters the `& sysdate9`. This token is recognized as a macro variable reference and passed on to the *Macro Processor* for processing (**3**). As long as the macro processor is working, the *Word Scanner* pauses. Since our small test program was sent in open code, the macro processor now checks whether it finds the macro variable in the *Global Symbol Table* (**4**). It does, because it is an automatic system variable. The *Macro Processor* now sends the contents of this variable back to the *Input Stack*. It is not written at the end as usual, but at the beginning of the stack (**5**). I.e. what is still on the input stack is the following: `13NOV2019"; RUN;`
Now the *Word Scanner* resumes its work (**6**), recognizes the remaining code as data step code and forwards it to the *Data Step Compiler*. As soon as it receives the `RUN;` statement, a so-called data step boundary, it means that the code can now be executed. The dataset `WORK.TODAY` is generated (**7**).

The macro processor cycle has been explained using a macro variable reference, but of course this applies equally to all Macro Calls / Statements / Functions / Variables and also to the Macro Quoting Functions, whose timing we will take a closer look at in the following section.

MACRO QUOTING TIMING

As we learned in the overview section, the Macro Quoting functions are divided into two categories regarding timing. The *Compilation Time* functions and the *Execution Time* functions. In most papers this is mentioned as a property of the respective function, but beyond that it remains uncommented, as if it were completely clear what we are talking about here. So what exactly does this mean?
Compilation and execution phases are available for the *Macro Processor* as well as for the *Data Step Compiler* and other functional units. There might be the first confusion what the property of the quoting functions actually refers to. Since these are macro functions, this property refers exclusively to the compilation and execution phase of the macro processor. As we have seen in the previous section, however, this is again in a somewhat strange relationship to the

other functional units. I.e. what should be remembered at this point is that all compilation and execution time quoting functions are already processed in the compilation phase of the data step (or other functional units).

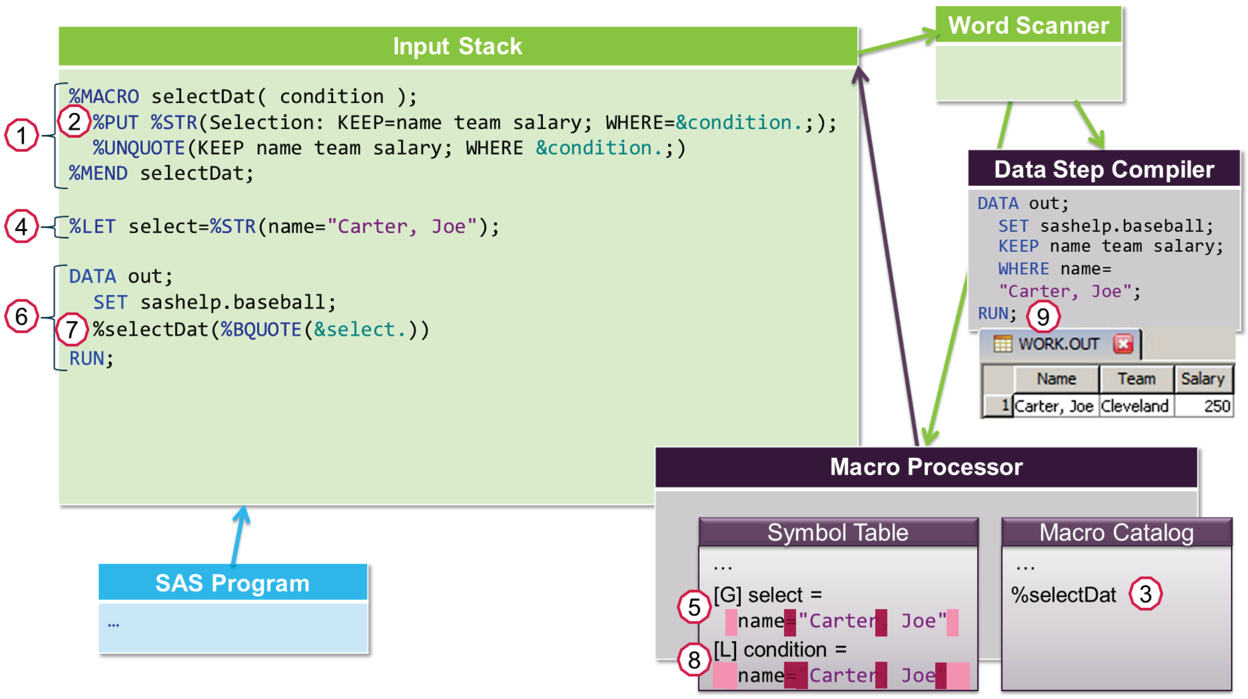
Let us have a look at the following example program:

```
%MACRO selectDat(condition);
    %PUT %STR(Selection: KEEP=name team salary; WHERE=&condition.;);
    %UNQUOTE(KEEP name team salary; WHERE &condition.;)
%MEND selectDat;

%LET select=%STR(name="Carter, Joe");

DATA out;
    SET sashelp.baseball;
    %selectDat(%BQUOTE(&select.))
RUN;
```

We do not need to talk about the meaningfulness of this piece of code, it should only serve illustration purposes. In the following figure, the scheme from Figure 7 is in principle reproduced once again. The visualization via the *Word Scanner* is omitted for the sake of clarity. Dark red boxes in the code represent masked characters and pink boxes represent ID bytes of the quoting functions.



Input Stack:

```
%MACRO selectDat( condition );
%PUT %STR(Selection: KEEP=name team salary; WHERE=&condition.;);
  %UNQUOTE(KEEP name team salary; WHERE &condition.;)
%MEND selectDat;

%LET select=%STR(name="Carter, Joe");

DATA out;
  SET sashelp.baseball;
%selectDat(%BQUOTE(&select.))
RUN;
```

Data Step Compiler:

```
DATA out;
  SET sashelp.baseball;
  KEEP name team salary;
  WHERE name=
  "Carter, Joe";
RUN;
```

WORK.OUT

	Name	Team	Salary
1	Carter, Joe	Cleveland	250

Symbol Table:

```
…
[G] select =
  name="Carter, Joe"
[L] condition =
  name="Carter, Joe"
```

Macro Catalog:

```
…
%selectDat
```

Figure 8: Timing of quoting

We have learned that the *Input Stack* is processed from top to bottom. In the first step the macro `%selectDat` is defined (**1**). Macro definitions clearly take place at compile time. This means that `%STR` and `%NRSTR` functions are being used. The expression (**2**) is masked to:

```
%PUT  Selection: KEEP=name team salary; WHERE=&condition.; ;
```

The macro itself is then compiled and stored in the Macro Catalog **(3)**.

Let us look at the `%LET` statement **(4)**. Macro statements in Open Code are actually executed immediately. But even these expressions are compiled before execution. In this case the `%STR` function is used again. The expression is masked to:

```
%LET select= name="Carter  Joe" ;
```

Then the statement is executed and the macro variable `select` is stored in the *Global Symbol Table* **(5)**.

Now we turn to the data step **(6)**. The first two lines of the data step are clearly recognized by the *Word Scanner* as pure data step code and forwarded to the *Data Step Compiler* for compilation. The macro call **(7)** becomes a bit more interesting. The macro `%selectDat` is already compiled and stored in the *Macro Catalog*. Nevertheless, this expression also goes through the compilation and execution phases. As usual, the `%STR` and `%NRSTR` functions would be used in the compilation phase. We don't have either here, so we move on to the execution phase. Here the expression is resolved from the inside to the outside. I.e. first the macro variable reference `&select` is resolved. This is taken from the *Global Symbol Table*:

```
%selectDat(%BQUOTE( name="Carter, Joe" ))
```

Now the `%BQUOTE` function is applied to the already masked content as follows:

```
%selectDat(  name= Carter, Joe   )
```

In addition to the ID bytes of the `%STR` function, the ID bytes of the %BQUOTE function are now added. And as we know from the overview section, the `%BQUOTE` function additionally masks the class **c** characters, i.e. the quotation marks are now also masked.

Then the macro itself is executed. The macro variable `condition` is created in the *Local Symbol Table* of the macro **(8)**. There is an output in the log generated and what is actually left from the macro call at the end as code is the expression within the `%UNQUOTE` function[5], which is then written back to the input stack.

```
KEEP name team salary; WHERE name="Carter, Joe";
```

This expression together with the `RUN;` statement is now identified by the *Word Scanner* as data step code and passed to the *Data Step Compiler*. It recognizes the data step boundary and executes the compiled code **(9)**.

CONCLUSION

We have seen that macro quoting is a complex topic. That is why the try and error approach will most likely continue to be used. The rules - which function masks what? - are quickly understood, and even better if the concept of character replacement is clear. The whole topic is getting complicated mainly by the timing component and here in particular by the interaction of macro processor and data step compiler (or other functional units). Once you have understood the cycle of macro code resolution, you can also better understand the timing of the quoting functions. I hope with this paper I was able to bring a little more light into this subject.

Finally, a few key points to remember:

- %STR / %NRSTR work **before** the resolution of the macro triggers **&** and **%**. %BQUOTE / %NRBQUOTE / %SUPERQ and all other quoting functions work **after** their resolution.

[5] It is not self-explanatory why we need the %UNQUOTE function here. Normally everything that leaves the macro facility is unquoted automatically. But there are a few cases where this does not work properly, especially when we deal with masked quotation marks. Without the %UNQUOTE SAS issues a syntax error. What is printed to the log looks ok though. Because masked characters are automatically unquoted when printed to the log, it is not easy to see where the syntax is erroneous.

- To mask **“static text”** input use %STR or %NRSTR.
- To mask the content of **&macro** variables or **%macro** calls use %BQUOTE, %NRBQUOTE or %SUPERQ.
- %SUPERQ is the actual **sister function** to %NRSTR at execution time.
- There is hardly a situation in which %BQUOTE would have to be preferred to the %NRBQUOTE function. Both work almost identically and warnings due to unresolved macro triggers are also issued with the %NRBQUOTE function.
  Hence the main quoting functions to remember are **%STR**, **%NRSTR**, **%NRBQUOTE** and **%SUPERQ**.
- Be clear about the effect of **single quotes** in the quoting context.
- The compilation and execution phase of the data step has nothing to do with the compilation / execution time of the quoting functions. **All masking already takes place during the compilation phase of the data step**.
- If a syntax error in the log is generated, but the expression in the log could be sent in the Data Step without problems - often an **%UNQUOTE over the entire expression helps**. It seems that especially masked quotation marks are not automatically unquoted correctly when they leave the Macro Facility.

REFERENCES

[1] SAS Institute Inc. 2016. SAS® 9.4 Macro Language: Reference, Fifth Edition. Cary, NC: SAS Institute Inc.

[2] Patterson, Brian; Remigio, Mylene. 2007, “Don’t %QUOTE() Me on This: A Practical Guide to Macro Quoting Functions”, SAS Global Forum 2007

[3] Lee, Shan. 2007, “Quoting Macro Variable References”. PhUSE 2007

[4] Hendrickx, John. 2014, “Dequote me on that: Using the dequote function to add some friendliness to SAS macros”. Phuse 2014

[5] Whitlock, Ian. 2003, “A Serious Look at Macro Quoting”. Proceedings of the 28th SAS Users Group International (SUGI 28)

CONTACT INFORMATION

Your comments and questions are valued and encouraged. Contact the author at:

Tim Lepp
Bayer AG
Email: tim.lepp(at)bayer.com