Underutilized Features in the SAS® Macro Language

George J. Hurley, The Hershey Company

ABSTRACT

From conversations in the SAS Community, it seems that there are key features of the macro language that are underutilized at best and misunderstood at worse. This talk will discuss some of these underutilized or misused SAS Macro features and when to use them. These features include when to use %sysfunc, %sysevalf, quoting functions, the parmbuff option, and other interesting features.

INTRODUCTION

The SAS Macro language allows for the use of repeatable code blocks within the SAS language. While the basics of getting a SAS Macro up and running are quite simple, there are some very useful aspects of this language that require a more thorough discussion. When used correctly, these features greatly increase the power of SAS Macro language; however, when used incorrectly, results can be puzzling.

UNDERUTILIZED FEATURES

%EVAL AND %SYSEVALF

Consider the following SAS Macro code:

```
%let x=3;
%let y=4;
%let z=&x+&y;

%put x=&x;
%put y=&y;
%put z=&z;
```

The %put statement will print the values of the macro variables x, y, and z into the log. Will the log then print x=3, y=4, z=7?

```
1    %let x=3;
2    %let y=4;
3    %let z=&x+&y;
4
5    %put x=&x;
x=3
6    %put y=&y;
y=4
7    %put z=&z;
z=3+4
```

As can be seen from the log above, the value of x is 3, the value of y is 4, but the value of z is not 7, but rather 3+4. The reason for this is that SAS Macro values are stored as text literals, rather than numbers. So, the value of x is not the number 3, but rather the text string, “3”. Consequently, the value of z is the text string, “3+4”. This makes sense when you are using macros to dynamically write SAS Code, because SAS Code itself is text. Consider the following code:

```
data one;
x=&x;
y=&y;
z=&x+&y;
run;

proc print noobs;
run;
```

As expected, the proc print produces the following output:

x	y	z
3	4	7

This is because the macro variables are resolved to text prior to the data step being executed, so what really happens is SAS transforms the data step to the following:

```
data one;
x=3;
y=4;
z=3+4;
run;
```

Since 3 and 4 here are not in quotes, SAS knows that x and y are intended as numeric variables, and consequently, so is z, which is the addition operator performed on two numeric variables.

But, what if we want to perform the addition operation within the Macro language and not within a data step? This is when %eval and %sysevalf come in useful. %eval is used to wrap around an integer operation to allow evaluation of the operation. For example, consider the following:

```
%let x=3;
%let y=4;
%let z=%eval(&x+&y);

%put x=&x;
%put y=&y;
%put z=&z;
```

Now, as expected, z holds the value 7. However, note this is still a text value.

```
17
18   %let x=3;
19   %let y=4;
20   %let z=%eval(&x+&y);
21
22   %put x=&x;
x=3
23   %put y=&y;
y=4
24   %put z=&z;
z=7
```

One might note the following code…

```
%let x=3;
%let y=4;
%let z=&x+&y;

%put x=&x;
%put y=&y;
%put z=%eval(&z);
```

produces the same results:

```
25
26   %let x=3;
27   %let y=4;
28   %let z=&x+&y;
29
30   %put x=&x;
```

```
x=3
31   %put y=&y;
y=4
32   %put z=%eval(&z);
z=7
```

Up to this point, the focus has been on %eval. %eval, as mentioned performs integer operations. %sysevalf, its sister function, performs floating point operations. Consider the following code:

```
%let w=3;
%let x=3.5;
%let y=4;
%let z1=%eval(&x+&y);
%let z2=%sysevalf(&x+&y);
%let d1=%eval(&y/&w);
%let d2=%sysevalf(&y/&w);

%put x=&x;
%put y=&y;
%put z1=&z1;
%put z2=&z2;
%put d1=&d1;
%put d2=&d2;
```

Also consider the log:

```
157  %let w=3;
158  %let x=3.5;
159  %let y=4;
160  %let z1=%eval(&x+&y);
ERROR: A character operand was found in the %EVAL function or %IF condition where a numeric
       operand is required. The condition was: 3.5+4
161  %let z2=%sysevalf(&x+&y);
162  %let d1=%eval(&y/&w);
163  %let d2=%sysevalf(&y/&w);
164
165  %put x=&x;
x=3.5
166  %put y=&y;
y=4
167  %put z1=&z1;
z1=
168  %put z2=&z2;
z2=7.5
169  %put d1=&d1;
d1=1
170  %put d2=&d2;
d2=1.33333333333333
```

The following should be noted from the log. First, when an addition problem with a decimal is placed in %eval, an error is generated. This is because %eval interprets "." as a character value, since it only expects integer operands. Likewise, %eval discards any fractional part of a division problem (via truncation) in order to return an integer. It can also be seen here that %sysevalf, using floating point arithmetic, generates behavior that would be expected.

In addition to arithmetic expressions, %eval and %sysevalf also accept logical arguments, such as the following:

```
%let w=3;
%let y=4;
%let z=%eval(&w>&y);
```

```
%put &z;
```

As can be seen in the log, the value of z is 0, since 3 is not larger than 4.

```
178  %let w=3;
179  %let y=4;
180  %let z=%eval(&w>&y);
181  %put &z;
0
```

Any part of the macro language that performs evaluations implicitly uses %eval. This can lead to some strange results. Consider the following macro, which is given in example 3 in the documentation for %eval (in SAS 9.3).

```
%macro compare(first,second);
   %if &first>&second %then %put &first > &second;
   %else %if &first=&second %then %put &first = &second;
   %else %put &first<&second;
%mend compare;
%compare(1,2)
%compare(-1,0)
%compare(12.8,2.2)
%compare(1.8,2.2)
```

It can be seen that the macro breaks down when decimal values are used. Specifically, 12.8 is given as being smaller than 2.2.

```
184  %macro compare(first,second);
185     %if &first>&second %then %put &first > &second;
186     %else %if &first=&second %then %put &first = &second;
187     %else %put &first<&second;
188  %mend compare;
189  %compare(1,2)
1<2
190  %compare(-1,0)
-1<0
191  %compare(12.8,2.2)
12.8<2.2
192  %compare(1.8,2.2)
1.8<2.2
```

This is because since, %eval is implicitly used in the %if statement, any values with a decimal are treated as a text value, and %if uses a text comparison instead. Since the first character “1” of “12.8” comes before the first character, “2” of “2.2”, “12.8” is hence smaller than “2.2”. As should be obvious, this can lead to odd behavior in a macro that relies on this comparison. Luckly, the macro can be re-written as follows if we anticipate it being given decimal values:

```
%macro compare(first,second);
   %if %sysevalf(&first>&second) %then %put &first > &second;
   %else %if %sysevalf(&first=&second) %then %put &first = &second;
   %else %if %sysevalf(&first<&second) %then %put &first<&second;
%mend compare;
%compare(1,2)
%compare(-1,0)
%compare(12.8,2.2)
%compare(1.8,2.2)
```

It can now be seen that results generated are as expected for all comparisons:

```
214  %macro compare(first,second);
215     %if %sysevalf(&first>&second) %then %put &first > &second;
216     %else %if %sysevalf(&first=&second) %then %put &first = &second;
217     %else %if %sysevalf(&first<&second) %then %put &first<&second;
218  %mend compare;
219  %compare(1,2)
1<2
220  %compare(-1,0)
-1<0
221  %compare(12.8,2.2)
12.8 > 2.2
222  %compare(1.8,2.2)
1.8<2.2
```

In addition to performing floating point arithmetic, %sysevalf has one other useful feature. %sysevalf has an optional conversion-type parameter, which accepts one of the following values (if specified): "BOOLEAN", "CEIL", "FLOOR", "INTEGER". These are fairly straight forward in their behavior. Consider the following example:

```
%let w=3;
%let x=5.6;
%let y=4;
%let z=6.1;

%let bool1=%sysevalf(&w-&w,boolean);
%let bool2=%sysevalf(&w-1000,boolean);
%let ceil1=%sysevalf(&z-&x,ceil);
%let ceil2=%sysevalf(&w-&x,ceil);
%let floor1=%sysevalf(&z-&x,floor);
%let floor2=%sysevalf(&w-&x,floor);
%let integer1=%sysevalf(&z-&x,integer);
%let integer2=%sysevalf(&w-&x,integer);

%put bool1=&bool1;
%put bool2=&bool2;
%put ceil1=&ceil1;
%put ceil2=&ceil2;
%put floor1=&floor1;
%put floor2=&floor2;
%put integer1=&integer1;
%put integer2=&integer2;
```

It can be seen that "BOOLEAN" produces a value that is 0 if the answer is 0 (or missing) and 1 otherwise. "CEIL" and "FLOOR" behavior exactly like ceiling and floor functions applied to the result. "INTEGER" truncates the result (e.g. it behaves identical to "FLOOR" for positive results and "CEIL" for negative results).

%SYSFUNC

The %sysfunc function is one of the most powerful functions in the SAS Macro Language. %sysfunc allows the user to execute most SAS datastep functions from within the macro. This has many uses in SAS. The following is presented as a example using the date function, intnx, which increments dates, but generally, the syntax is similar regardless of the function chosen.

In this example, a macro variable has a date stored in it. The date is this author's 8th wedding anniversary. The following code will allow the author's 10th wedding anniversary date to be stored into a macro variable.

```
data _null_;
dt='20may13'd;
call symput ("dt",dt);
```

```
run;

%let ann10=%sysfunc(intnx(YEAR, &dt, 2, S));
%let ann10f=%sysfunc(intnx(YEAR, &dt, 2, S),mmddyy10.);

%put George and Debbie - 8 Years, &dt;
%put George and Debbie - 10 Years, &ann10;
%put George and Debbie - 10 Years, &ann10f;
```

The log displays the following:

```
438  %let ann10=%sysfunc(intnx(YEAR, &dt, 2, S));
439  %let ann10f=%sysfunc(intnx(YEAR, &dt, 2, S),mmddyy10.);
440
441  %put George and Debbie - 8 Years, &dt;
George and Debbie - 8 Years,        19498
442  %put George and Debbie - 10 Years, &ann10;
George and Debbie - 10 Years, 20228
443  %put George and Debbie - 10 Years, &ann10f;
George and Debbie - 10 Years, 05/20/2015
```

Several things should be noted here. First, the intnx function correctly incremented the SAS date function stored in the macro variable dt by two years. However, intnx returned a SAS date two years later. It may not be meaningful if the author was using this macro to automate a title in a macro to use the SAS date. For example, having a gift engraved, "George and Debbie – 10 Years, 20228," would be at best not interpretable, and at worst may lead to a really bad evening. Luckily, SAS allows a format be applied to the macro variable output using %sysfunc. This, obviously, improves the interpretability of the output here, and clearly has many other applications.

Now, suppose the author wanted the output to read, "The author's 10th anniversary is 05/20/2015". Consider this code:

```
%put The author's 10th anniversary is &ann10f;
```

And the resulting output:

```
444  %put The author's 10th anniversary is &ann10f;
```

Note that nothing was written to the log by the %put statement. This is due to the fact that SAS read the single quote as the opening of a quotation and has not yet encountered an ending quote. SAS will continue looking for this second quote, and as a program runs, the following warning will be encountered if it does not encounter one.

```
WARNING: The quoted string currently being processed has become more than 262 characters long.
         You might have unbalanced quotation marks.
```

If it does encounter a second single quote, it will end the quotation and continue processing, which can lead to a host of other issues, which at best will generate this note:

```
NOTE 49-169: The meaning of an identifier after a quoted string might change in a future SAS
             release.  Inserting white space between a quoted string and the succeeding
             identifier is recommended.
```

In either case, the program will not run in the way intended. The solution is to utilize quoting functions, which is the topic of the next section.

QUOTING FUNCTIONS

There are several quoting functions in SAS, each with somewhat different properties. In general quoting functions mask the values of special characters within a macro variable.

The example from the previous section can be solved by utilizing the %bquote function to mask the single quote. Here,

```
%put The author%bquote(')s 10th anniversary is &ann10f;
```

will result in the following log:

```
733  %put The author%bquote(')s 10th anniversary is &ann10f;
The author's 10th anniversary is 05/20/2015
```

This is as expected. Each quoting function masks a different range of characters. Table 1 below lists the functions and what they mask.

Table 1. Macro Quoting Functions

%str	+ − * / < > = ¬ ^ ~ ; , # blank AND OR NOT EQ NE LE LT GE GT IN; as well as matched single quotes, double quote, or parenthesis
%nrstr	Same as %str, but also includes & and %
%quote	+ − * / < > = ¬ ^ ~ ; , # blank AND OR NOT EQ NE LE LT GE GT IN; as well as matched single quotes, double quote, or parenthesis
%nrquote	Same as %str, but also includes & and %
%bquote	' " ( ) + − * / < > = ¬ ^ ~ ; , # blank AND OR NOT EQ NE LE LT GE GT IN
%nrbquote	Same as %bquote, but also includes & and %
%superq	& % ' " ( ) + − * / < > = ¬ ^ ~ ; , # blank AND OR NOT EQ NE LE LT GE GT IN; however, it operates somewhat differently (see discussion)

%str can also be used to mask unmatched quotes, but the syntax is somewhat different. For the example above,

```
%put The author%str(%')s 10th anniversary is &ann10f;
```

will also work, yielding:

```
737  %put The author%str(%')s 10th anniversary is &ann10f;

THE AUTHOR'S 10TH ANNIVERSARY IS 05/20/2015
```

Note the addition of the "%" prior to the single quote allows this to work. This will also work for unmatched parentheses and unmatched double quotes.

It can be seen that %str and %quote mask the same characters, just as %nrstr and %nrquote do. Just like the %str family, %quote can also handle unmatched quotes and parentheses via a leading percent sign.

There is a subtle difference between these. The %str family marks constant text, and is known as compilation functions. The %quote family acts at execution time and are called execution functions. The %bquote family is an extension of %quote, acting at execution time as well, but not requiring the same treatment of unmatched quotes or parentheses that %quote does.

In many cases, this may make no difference. However, here is an example to illustrate the difference:

```
%let vars=My 10th anniversary is %nrstr(&ann10f);
%let varb=My 10th anniversary is %nrbquote(&ann10f);
%put &vars;
%put &varb;
%put %nrstr(&vars);
%put %nrbquote(&vars);
```

The log returns the following:

```
79   %let vars=My 10th anniversary is %nrstr(&ann10f);
80   %let varb=My 10th anniversary is %nrbquote(&ann10f);
81   %put &vars;
My 10th anniversary is &ann10f
82   %put &varb;
```

```
My 10th anniversary is 05/20/2015
83   %put %nrstr(&vars);
&vars
84   %put %nrbquote(&vars);
My 10th anniversary is &ann10f
```

Here, %nrstr, acting at compile time, presents any resolution of the macro variables it masks. So, %put %nrstr(&vars) yields "&vars" being written to the log, and %put &vars generates the string with the macro variable "ann10f" unresolved. Alternatively, %nrbquote acts at execution time, and we see that %put &varb allows the resolution of "ann10f". In %put %nrbquote(&vars), it is seen that "vars" is resolved and written to the log, but since "ann10f" is quoted with %nrstr, the macro variable "ann10f" is not resolved.

Note that the quoting functions beginning with %nr all add "&" and "%" to the characters masked by the parent quoting function; a trick to remember this is that "NR" stands for "Not Resolved". That is, they prevent the resolution of what would appear to be a macro contained therein.

%superq acts somewhat differently than any of these macro functions. %superq is useful when you need to mask references that may arise within a macro variable. Consider this example:

```
data storelist;
store='A&P';
output;
run;

data one; set storelist;
       call symput("store1", store);
run;

%let sq1=%superq(store1);
%let nb1=%nrbquote(&store1);
%let ns1=%nrstr(&store1);

%put &sq1;
%put &nb1;
%put &ns1;
```

The log now reads:

```
358  data storelist;
359  store='A&P';
360  output;
361  run;

NOTE: The data set WORK.STORELIST has 1 observations and 1 variables.
NOTE: DATA statement used (Total process time):
      real time           0.07 seconds
      cpu time            0.03 seconds


362
363  data one; set storelist;
364      call symput("store1", store);
365  run;

NOTE: There were 1 observations read from the data set WORK.STORELIST.
NOTE: The data set WORK.ONE has 1 observations and 1 variables.
NOTE: DATA statement used (Total process time):
      real time           0.04 seconds
```

```
      cpu time            0.04 seconds


366
367  %let sq1=%superq(store1);
368  %let nb1=%nrbquote(&store1);
WARNING: Apparent symbolic reference P not resolved.
369  %let ns1=%nrstr(&store1);
370
371  %put &sq1;
A&P
372  %put &nb1;
A&P
373  %put &ns1;
&store1
```

As can be seen %superq prevented the resultion of “&P” within the text of the macro variable. %nrbquote generated a warning message when it encountered “&P” and attempted to resolve it. %nrstr, as expected prevents the resolution of the macro variable “store”. Hence, only %superq returns the desired result. It should be noted %superq does not accept the ampersand associated with the macro variable when it is called.

Like everything in SAS, there is a work around here, but it is more complicated. In the code below, the tranwrd function is used to embed %nrstr into the macro text to prevent the resolution of “&P” when the variable is %put to the log.

```
data sl2; set storelist;
y=tranwrd(store,'&','%nrstr(&)');
run;


data two; set sl2;
       call symput("store2", y);
run;

%put &store2;
```

The log yields:

```
374  data sl2; set storelist;
375  y=tranwrd(store,'&','%nrstr(&)');
376  run;

NOTE: There were 1 observations read from the data set WORK.STORELIST.
NOTE: The data set WORK.SL2 has 1 observations and 2 variables.
NOTE: DATA statement used (Total process time):
      real time           0.06 seconds
      cpu time            0.04 seconds


377
378
379  data two; set sl2;
380      call symput("store2", y);
381  run;

NOTE: There were 1 observations read from the data set WORK.SL2.
NOTE: The data set WORK.TWO has 1 observations and 2 variables.
```

```
NOTE: DATA statement used (Total process time):
      real time           0.03 seconds
      cpu time            0.03 seconds


382
383  %put &store2;
A&P
```

In the code above, the tranwrd function is used to embed %nrstr into the macro text to prevent the resolution of "&P" when the variable is %put to the log. As can be seen, this works, yet, is more cumbersome. Also, in this case, there was knowledge that "&" was the issue in the store list file. In a real example, it is often the case that this file may contain any number of strings that would cause issue. Since %superq masks a wide range of characters, %superq is the better approach.

Many macro functions, such as %sysfunc, include a "q-version", such as %qsysfunc. The "q-versions" allow for masking of special characters in its result.

PARMBUFF

The parmbuff option is available on the %macro statement. Parmbuff allows for any number of optional macro parameters to be entered upon a call to the macro and stores the values including the opening and closing parentheses in the marcro variable "syspbuff".

Consider:

```
%macro runme / parmbuff;
%put syspbuff=&syspbuff;
       %if %length(&syspbuff)<=2 %then %put I have no friends;
       %else %do;
              %let num=1;
              %let dsname=%scan(&syspbuff,&num);
              %do %while(&dsname ne);
                     %put &dsname is my friend;
                     %let num=%eval(&num+1);
              %let dsname=%scan(&syspbuff,&num);
              %end;
       %end;
%mend runme;


%runme ;
%runme ();
%runme (Charlie);
%runme (Bob, Charlie);
%runme (Mary, Jonah, Marie);
```

This generates the following log:

```
304  %macro runme / parmbuff;
305  %put syspbuff=&syspbuff;
306      %if %length(&syspbuff)<=2 %then %put I have no friends;
307      %else %do;
308          %let num=1;
309          %let dsname=%scan(&syspbuff,&num);
310          %do %while(&dsname ne);
311              %put &dsname is my friend;
312              %let num=%eval(&num+1);
```

```
313             %let dsname=%scan(&syspbuff,&num);
314          %end;
315      %end;
316  %mend runme;
317
318
319  %runme ;
syspbuff=
I have no friends
320  %runme ();
syspbuff=()
I have no friends
321  %runme (Charlie);
syspbuff=(Charlie)
Charlie is my friend
322  %runme (Bob, Charlie);
syspbuff=(Bob, Charlie)
Bob is my friend
Charlie is my friend
323  %runme (Mary, Jonah, Marie);
syspbuff=(Mary, Jonah, Marie)
Mary is my friend
Jonah is my friend
Marie is my friend
```

As can be seen, the listed parameters are stored in “syspbuff” with all parentheses and commas.

Parmbuff can also be used with named parameters, as seen in the following example:

```
%macro runme (temp=3)/ parmbuff;
%put temp=&temp;
%put syspbuff=&syspbuff;
       %if %length(&syspbuff)<=2 %then %put I have no friends;
       %else %do;
              %let num=1;
              %let dsname=%scan(&syspbuff,&num);
              %do %while(&dsname ne);
                     %put &dsname is my friend;
                     %let num=%eval(&num+1);
              %let dsname=%scan(&syspbuff,&num);
              %end;
       %end;
%mend runme;


%runme ;
%runme (temp=5);
%runme (Charlie);
%runme (temp=5,Charlie);
%runme (Charlie,temp=5);
```

The log generated is as follows:

```
346  %macro runme (temp=3)/ parmbuff;
347  %put temp=&temp;
348  %put syspbuff=&syspbuff;
349      %if %length(&syspbuff)<=2 %then %put I have no friends;
350      %else %do;
```

```
351          %let num=1;
352          %let dsname=%scan(&syspbuff,&num);
353          %do %while(&dsname ne);
354              %put &dsname is my friend;
355              %let num=%eval(&num+1);
356              %let dsname=%scan(&syspbuff,&num);
357          %end;
358      %end;
359  %mend runme;
360
361
362  %runme ;
temp=3
syspbuff=
I have no friends
363  %runme (temp=5);
temp=5
syspbuff=(temp=5)
temp=5 is my friend
364  %runme (Charlie);
temp=3
syspbuff=(Charlie)
Charlie is my friend
365  %runme (temp=5,Charlie);
ERROR: All positional parameters must precede keyword parameters.
366  %runme (Charlie,temp=5);
temp=5
syspbuff=(Charlie,temp=5)
Charlie is my friend
temp=5 is my friend
```

It is of interest here to note that first, when the keyword parameter temp is specified in the macro call, it is included in the macro variable “syspbuff”. However, when not included and the default value is used, it is not included. It is also noteworthy that positional parameters still must precede keyword parameters as always.

CONCLUSION

The macro language has many interesting and underutilized components. The components discussed here are just a small sampling of the vast number of available functions and options in the macro language.

CONTACT INFORMATION

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

George J. Hurley
The Hershey Company
19 E Chocolate Ave.
Hershey, PA 17033
717.534.5337
717.534.6991
ghurley@hersheys.com