Paper B2B-12

Submitting SAS® Code On The Side

Rick Langston, SAS Institute Inc., Cary NC

Presented by Vince DelGobbo

ABSTRACT

This paper explains the new DOSUBL function and how it can submit SAS code to run "on the side" while your DATA step is still running. It also explains how this function differs from invoking CALL EXECUTE or invoking the RUN_COMPILE function of FCMP. Several examples are shown that introduce new ways of writing SAS code.

INTRODUCTION

The DOSUBL function is new in the second maintenance release of SAS 9.3. The function submits code to SAS and waits for that code to complete submission.

The DOSUBL function has a single argument, which is a string value. For example:

```
data _null_;
   rc = dosubl('data x; y=1; run; %let abc=1;');
   abc=symget('abc'); put abc=;
run;
data _null_;
   set x;
   put _all_;
run;
```

Although this code is not of much value, it shows the syntax of the function. The small DATA step that creates Work.X is submitted and run to completion before the outer DATA step is completed.

All macro variables, macro definitions, and Work data sets are available to the code that is to be executed. All macro variables, macro definitions, and Work data sets created by the executed code are then subsequently available to your DATA step and any SAS code that follows. For example, in the SAS code above, the data set Work.X and the macro variable &abc are available after the DOSUBL function completes.

THE DIFFERENCE BETWEEN DOSUBL AND CALL EXECUTE

DOSUBL differs from CALL EXECUTE in that CALL EXECUTE performs only immediate execution on macro code. If that macro code expands to DATA and PROC steps, that code is stacked to execute after the current DATA step completes. Consider this code:

```
%macro doit(value);
data _null_;
   call symput('xyz',"&value.");
run;
%mend doit;

%let xyz=q;
data _null_;
   call execute('%doit(a)');
   xyz_value = symget('xyz');
   put xyz_value=;
run;
```

When this code runs, the DATA step gives us this output:

```
xyz_value=q
```

The output shows that the DATA step with the SYMPUT call in it has been stacked to run later, so the value of &xyz is still q.

But, if we run this code instead, we get the desired result:

```
%let xyz=q;
data _null_;
```

```
      rc = dosubl('%doit(b)');
      xyz_value = symget('xyz');
      put xyz_value=;
      run;
```

We get this result because the %doit macro has run to completion before returning from the DOSUBL function:

```
xyz_value=b
```

THE DIFFERENCE BETWEEN DOSUBL AND RUN_MACRO

Another function that is similar to DOSUBL is RUN_MACRO. The RUN_MACRO function runs within the compiler (FCMP) environment, and cannot run in any other environment. Also, RUN_MACRO is intended specifically for invoking macros (hence its name), but the DOSUBL function can execute any code provided to it.

It is more straightforward to invoke macros using DOSUBL. Consider this example. The macro %testmacro expects to use two macro variables, &a and &b, and the result of subtracting them is saved in macro variable &p. To invoke this macro using PROC FCMP, you call RUN_MACRO and pass the name of the macro and the arguments whose names are used as the names of the macros containing the values.

```
   /* Create a macro called TESTMACRO. */
%macro testmacro;
   %let p = %sysevalf(&a - &b);
%mend testmacro;

   /* Use TESTMACRO within a function in PROC FCMP to subtract two numbers. */
proc fcmp outlib = sasuser.ds.functions;
   function subtract_macro(a, b);
      rc = run_macro('testmacro', a, b, p);
      if rc eq 0 then return(p);
      else return(.);
   endsub;
run;
```

To call the macro, you do so via the function that has been defined:

```
   /* Make a call from the DATA step. */
option cmplib = (sasuser.ds);
data _null_;
   a = 5.3;
   b = 0.7;
   p = .;
   p = subtract_macro(a, b);
   put p=;
run;
```

Resulting in this output:

```
p=4.6
```

Compare using the FCMP environment with using DOSUBL, where you can call the macro directly:

```
%global a b p;
%put p=&p; /* should not yet be known */

%let a=5.3; %let b=0.7;
data _null_;
     rc = dosubl('%testmacro');
     run;
%put p=&p;
```

Also resulting in the same output:

```
p=4.6
```

USING DOSUBL WITHIN PROC STREAM

DOSUBL can be very handy when it is used in the new STREAM procedure, which is experimental in SAS 9.3 and production in SAS 9.4. The STREAM procedure enables you to process an input stream that consists of arbitrary text that can contain SAS macro specifications.

First consider this macro:

```
%macro setval(x);
   %global y;
   data _null_;
      y = &x + 2;
      call symputx('y',y);
      run;
%mend setval;

%let given=3;
```

We want to generate output from PROC STREAM, where we substitute macro variables and computed values:

```
filename mytext temp;
proc stream outfile=mytext; begin
I was given &given apples and I added
 2 more of them and my new amount was %setval(&given) &y.
;;;;
```

When this code runs, we get the following in the mytext file:

```
I was given 3 apples and I added 2 more of them and my new amount was
data _null_;  y = 3 + 2; call symputx('y',y); run;
```

We did not want to see the generated SAS code, just the result of the computation in the macro variable. Using DOSUBL with %SYSFUNC, we can do this successfully:

```
%let given=3;
filename mytext temp;
proc stream outfile=mytext; begin
I was given &given apples and I added
 2 more of them and my new amount was
%let x=%sysfunc(dosubl(%setval(&given))); &y..
;;;;
```

This results in this text in the mytext file:

```
I was given 3 apples and I added 2 more of them and my new amount was 5.
```

The executed code is not part of the output stream because it is being executed "on the side".

USING DOSUBL WITH PROC SQL

Here is another example to compare CALL EXECUTE with DOSUBL. This example invokes PROC SQL and its ability to create macro variables. We would like to use those macro variables within the same DATA step. The very same PROC SQL code is passed both to CALL EXECUTE and to DOSUBL.

First we see the SAS log where we create data set Work.Temp with two observations each for the values of 1 through 5, for the variable i:

```
1          data temp;
2               do i=1 to 5;
3                  output; output;
4                  end;
5               run;

NOTE: The data set WORK.TEMP has 10 observations and 1 variables.
```

We can run PROC SQL and use the COUNT function with the INTO clause to create macro variables &n and &dist, which hold the observation count and the distinct value count respectively:

```
6          proc sql;
7               select
8                 count(i) into :n from temp;
9               select
10                count(distinct i) into :dist from temp;
11              quit;
12
13         %put count is &n with &dist distinct values ;
count is       10 with        5 distinct values
14
```

We try invoking the same code with CALL EXECUTE, changing the names of the macro variables to avoid confusion. We attempt, within the same DATA step, to access the &n and &distmacro variables via SYMGET. But, we find they are not yet available. The PROC SQL step executes after our current DATA step completes, as we see in the SAS log:

```
15         data _null_;
16              call execute('
17         proc sql;
18              select
19                count(i) into :n2 from temp;
20              select
21                count(distinct i) into :dist2 from temp;
22              quit;
23              ');
24              n2 = input(symget('n2'),best12.);
25              dist2 = input(symget('dist2'),best12.);
26              put n2= dist2=;
27              run;

NOTE: Invalid argument to function SYMGET at line 24 column 17.
NOTE: Invalid argument to function SYMGET at line 25 column 20.
n2=. dist2=.
n2=. dist2=. _ERROR_=1 _N_=1

NOTE: CALL EXECUTE generated line.
1          + proc sql;
1          +                select        count(i) into :n2 from temp;
1          +                                                                  select
count(distinct i) into :dist2 from temp;
1          +
     quit;
```

Now, we try the invoking the same code using DOSUBL , changing the names of the macro variables to avoid confusion. This time we are able to obtain the macro variables because the PROC SQL code executed “on the side” while our DATA step awaited the return from DOSUBL.

```
28
29         data _null_;
30              rc = dosubl('
31         proc sql;
32              select
33                count(i) into :n3 from temp;
34              select
35                count(distinct i) into :dist3 from temp;
36              quit;
37              ');
38              n3 = input(symget('n3'),best12.);
```

```
39              dist3 = input(symget('dist3'),best12.);
40              put n3= dist3=;
41              run;


n3=10 dist3=5
```

CODE SIMPLIFICATION

In the paper “Use the Full Power of SAS in Your Function-Style Macros” (Rhoads 2012), there are several examples using the RUN_MACRO function. Michael Rhoads describes the three pieces of SAS code - the outer macro, the user-written function, and the inner macro - as a way to harness the power of function-style macros.

However, using DOSUBL, this becomes much simpler with only one section of SAS code needed. In Michael’s first example that is described in the paper, the three sections of code can be reduced to one section using DOSUBL, as seen below:

```
%MACRO ExpandVarList(data=_LAST_, var=_ALL_);
   %if %upcase(%superq(data)) = _LAST_
      %then %let data = &SYSLAST;
   %let rc = %sysfunc(dosubl(%str(
      proc transpose data=&DATA(obs=0) out=ExpandVarList_temp;
      var &VAR;
      run;
      proc sql noprint;
         select _name_ into :temp_varnames separated by ' '
         from ExpandVarList_temp
      ;
      drop table ExpandVarList_temp;
      quit
   )));

&temp_varnames
%MEND ExpandVarList;
```

CONCLUSION

The DOSUBL function is a very powerful addition to your arsenal of SAS functions. Being able to execute SAS code “on the side” allows for you to perform arbitrary functionality and react to results without having to resort to more complex SAS coding.

REFERENCES

Rhoads, Michael. 2012. “Use the Full Power of SAS in Your Function-Style Macros”. NorthEast SAS Users Group 2012, Baltimore, Maryland. Available at www.**nesug**.org/Proceedings/**nesug**12/bb/bb14.pdf.

CONTACT INFORMATION

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

Rick Langston
SAS Campus Drive
SAS Institute Inc.
Cary, NC 27513
E-mail: Rick.Langston@sas.com