CALL EXECUTE: How and Why

H. Ian Whitlock, Westat Inc.

Abstract

CALL EXECUTE is a relatively new DATA step function which interacts with the SAS Macro Facility. It allows one to send character data to the macro facility for immediate macro execution during the execution of the DATA step. It was introduced in SAS® 6.07 and documented in *Technical Report P-222 Changes and Enhancements to Base SAS Software*.

This paper gives examples showing how to use this routine and why you might want to use it. It also explains how macros invoked by CALL EXECUTE behave differently from those invoked via the standard macro call.

Introduction

You are to organize a program to produce PROC PRINTs for many different SAS data sets. How should the program be organized? One interesting way is to treat the list of data sets as data and write a program to use this data.

```
data _null_ ;
  input dsn $char50. ;
  call execute
    ( "proc print"                ||
      "  data = " || dsn || ";" ||
      "run ;" ) ;
cards ;
nat.rep
st.akrep
st.alrep
run ;
```

One might think we are trying to execute PROC PRINT in a DATA step, but this is not what is happening. After each record is read, CALL EXECUTE sends the character string between the parentheses to the macro facility. The macro facility sees no macro instructions so it dumps the code into the input stack for processing when the DATA step is finished. Thus the print steps are generated during the execution of the DATA step, but they are not executed until after the DATA step completes.

In pre-macro times one used PUT statements to write SAS code to a file and then used %INCLUDE to execute it. We achieved essentially the same result by using the input stack instead of a new file. The main advantage is that we don't have to set up the file or delete. (In SAS 6.11 one can write code to a work catalog, hence managing the file is now easier, and makes this old method more attractive in some cases. In some systems one can write external files to the SAS work directory, and leave it to the system to clean them up.)

Example Without Macro Code

Consider writing a program to read a flat file containing both data and specifications for reading the data. For example:

```
* specifications variable format
V NMVAR    3.
V DTVAR    DATE7.
V CHVAR    $CHAR4.
* data follows
D 10015JUL94ABCD
D 20012DEC93WXYZ
```

In this example we would like to have the code:

```
data temp ;
  infile in firstobs = 6 ;
  input @3
    nmvar  3.
    dtvar  date7.
    chvar  $char4.
  ;
run ;
```

Since the details change with the input data we want to generate this code automatically. Except for the '6' the first three lines are fixed. The block of code just after 'input @3' is a partial copy of what we read from the top of the file. The last two lines are again fixed. We should be able to structure the program as three CALL EXECUTES:

1. at _N_ = 1 to write the first block
2. on each observation of type V to handle the varying input details
3. on the first observation of type D to write the last block.

The only problem is that we won't know the starting observation until after we have read all the variables and formats. We can move this option to an INFILE statement in the last block, since a buffer accumulates options given in INFILE statements.

```
data _null_ ;
  length var $ 9 fmt $ 15 ;

  if _n_ = 1 then call execute
    ('data temp; infile in;' ||
     '  input @3 ' ) ;

  infile in ;
  input rectype $char1. @ ;

  if rectype = 'V' then
  do ;
    input var $ fmt $ ;
    call execute ( var || fmt ) ;
  end ;
  else
```

```
    if rectype = 'D' then
    do ;
      call execute
        ( '; infile in firstobs = ' ||
              put ( _n_ , 4. )      ||
          ' ; run ; '
        ) ;
      stop ;
    end ;
  run ;
```

Both this example and the one given in the introduction could have been solved using CALL SYMPUT to generate arrays of macro variables and using macro code to generate the required SAS code. What are the disadvantages of using CALL SYMPUT for these problems? The code would have been longer and more complex, and it would have required sophisticated macro code with a longer execution time. It is the direct simplicity of the above code that makes CALL EXECUTE appealing.

With the above examples in mind one should be able to find many simple examples where CALL EXECUTE can provide a simple solution to repetitive code problems. We turn instead to the problem of how CALL EXECUTE works when macro code is involved.

Example With Macro Code

CALL SYMPUT chooses the environment for the macro variables that it creates. If the variable already exists, then the environment of that variable is chosen. If the environment of the "SYMPUT" contains local variables, then the new "SYMPUT" variable is also local to this environment; otherwise the variable will be put in the first local environment encountered with local variables, or in the global environment when no such local environment is encountered. One often finds the need to force the variable to be global. This means declaring the variable ahead of time. How can this be done when the name of the variable is data? CALL EXECUTE comes to the rescue.

```
  data _null_ ;
    input name $ value $ ;
    call execute
      ('%global ' || name || ';' ||
       '%let ' || name || ' = ' ||
            value || ';'
      ) ;
  cards ;
  abc xyz
  ;
```

This example is different from the previous one in that we are now sending open macro code instructions to the macro facility. It is important that the %GLOBAL and %LET tokens be enclosed in single quotes. At compile time the word scanner looks inside double quotes and sends tokens beginning with a %-sign to the macro facility. We want the tokens sent during execution time, not compile time; hence it is important to hide the %-signs in single quotes.

When the DATA step containing the CALL SYMPUT statement is housed in a macro, it would often be nice to guarantee that the created variable will be local. You might expect to replace the %GLOBAL with a %LOCAL, but this cannot be done. %GLOBAL statements are open code macro statements; %LOCAL statements must be macro compiled when the macro is read and we would be trying to send it during macro execution time. CALL EXECUTE can handle macro instructions, but only those that do not involve macro compilation.

Example Invoking a Macro

Very quickly one learns that it is a pain to spell out a lot of SAS code using CALL EXECUTE. Is there a better way? Yes, macros generate SAS code, hence it is better to use CALL EXECUTE to send the short message, "invoke a macro" rather than give the SAS code directly. Remember that the invocation of a macro can appear in open code.

For example, suppose we have the error handling code below to be executed whenever the data set ERRS has observations.

```
    data w ;
      merge
        errs ( in = w )
        original ;
      by baseid ;
    run ;

    proc print data = w ( obs = 500 ) ;
    run ;

    %put Serious errors - halting ;
    endsas ;
```

We could house the code in a macro FATALERR and then use the following code to invoke the macro.

```
      data _null_ ;
        set errs ( obs = 1 ) ;
        call execute ('%fatalerr' ) ;
      run ;
```

What happens? If the step does not stop with the first execution of the SET statement because ERRS is empty, then the string '%FATALERR' is sent to the macro facility for execution. In effect, we have written

```
      %IF ERRS is not empty %then %FATALERR;
```

When the macro executes, it first generates the code for the DATA step, and then code for the PROC PRINT, both of which are dumped into the input stack for processing. Now the %PUT message is written to the log and finally the ENDSAS statement is dumped into the input stack. Note that the %PUT message is written to the log before the merge step is compiled, but after it is generated. When the DATA step containing the CALL EXECUTE finishes, the merge and print steps are compiled and executed, followed by the ENDSAS request. In general, as a result of executing a macro via CALL EXECUTE, the macro facility sends constant text to the input stack and processes macro instructions sequentially.

If the macro FATALERR had been invoked directly each step would have been generated, compiled, and then executed, before the next step. Hence there is a subtle, but important, difference here. With the CALL EXECUTE invocation each step is generated sequentially, but none are compiled or executed until after the DATA step containing the CALL EXECUTE is finished. This means that the generated steps are executed outside the macro environment that generated them. Under direct macro invocation the steps would sequentially compile and execute within the macro environment instead of outside it. Consequently, one must be very careful when writing macros to be invoked through CALL EXECUTE. Fortunately, a lot of the time it doesn't matter.

A Bad Example Using a %INCLUDE

Consider a simpler approach using %INCLUDE. Suppose we have the code

```
proc print data = &data ;
run ;
```

in a file REP.SAS, and we used CALL EXECUTE in the following way:

```
data _null_ ;
  input dsn $char50. ;
  call execute
    ('%let data = '|| dsn ||';' ||
     '%include "rep.sas" ; ' ) ;
cards ;
nat.rep
st.akrep
st.alrep
run ;
```

During the execution of this step the macro variable is changed three times ending up with the value ST.ALREP. Three copies of the %INCLUDE instruction are sent to the input stack. Remember that %INCLUDE is not a macro instruction! But now they all execute giving PROC PRINTs of ST.ALREP. The variable DATA changed during execution of the CALL EXECUTE DATA step; hence at the end of execution the value of &DATA is the final value ST.ALREP. So each time the include file is subsequently compiled &DATA resolves to ST.ALREP.

Contrast this with invoking the macro PRINTIT

```
%macro printit ( data = ) ;
    proc print data = &data ;
    run ;
%mend  printit ;
```

using

```
data _null_ ;
  input dsn $char50. ;
  call execute
    ('%printit ( data = '|| dsn ||')'
) ;
cards ;
nat.rep
st.akrep
st.alrep
run ;
```

Here the parameter DATA is resolved during execution of the macro, which is during the DATA step execution. Hence the correct value is used.

The DATA Step Function Reference Problem

Consider the macro:

```
%macro m1 ( p ) ;
  data _null_ ;
    p = &p ;
    q = symget ('p' ) ;
    put '%M1:' p= q= ;
  run ;
%mend  m1 ;
```

Direct invocation %M1 (1) produces the message

```
%M1: p=1 q=1
```

as one would expect. But the code:

```
data _null_ ;
   call execute ('%M1(1)') ;
run ;
```

either produces the error message, “NOTE: Invalid argument to function SYMGET at line 1 column 39.”, or it works and most likely produces an unexpected result. Using the system option MPRINT doesn't help because it shows that exactly the same code is produced whether CALL EXECUTE or direct invocation is used.

What happened? The line “P = &P ;” generated the line “P = 1 ;” showing that the parameter P existed and had the value 1 when the line was generated. The line “Q = SYMGET (‘P’) ;’ is constant text and generates itself. So what went wrong? The data step executed after the DATA step containing the CALL EXECUTE; hence it was after the macro M1 completed execution (during the invoking DATA step). But P was a parameter of M1; hence it no longer existed when the line “Q = SYMGET (‘P’) ;” was compiled. Thus either the compiler complains that the macro variable P doesn't exist. It's life ended when the macro M1 finished generating code, but the generated DATA step executed after the CALL EXECUTE step finished, i.e. long after M1 finished.

On the other hand, the code would work if there happened to be a macro variable P in the execution environment. Of course, this would probably produce the wrong value, since it is the wrong variable. Worse yet, it might accidentally produce the correct result.

Are there other ways to achieve this bad result? Sure. One could use the RESOLVE function or CALL EXECUTE. In each case a DATA step function interfacing with the macro facility is used to delay access to the variable until the containing DATA step executes in the wrong macro environment. Can one produce an error with a direct macro reference to the variable? I don't think so, because any direct reference would have to be resolved in the correct environment.

The DATA Step Function Assignment Problem

In the last example the reference to a macro variable was bad because of the delayed execution making use of the variable. In this example the problem is that the assignment of the variable is delayed until after its use.

Consider the two step macro M2:

```
%macro m2 ( data = ) ;

   %local commvar ;

   data _null_ ;
      call symput
         ( 'commvar' , nobs ) ;
      stop ;
      set &data nobs = nobs ;

   run ;

   %if &commvar = 0 %then
   %do ;
      /* error processing */
   %end ;
   %else
   %do ;
      /* normal processing */
   %end ;

%mend m2 ;
```

If M2 is invoked directly everything is fine. In fact the code illustrates a standard macro technique for communicating between steps. In this case the first step communicates the size of a SAS data set in order to make a decision about which branch of two possibilities to execute.

How would the invocation of M2 via CALL EXECUTE work? By the first line of the macro, COMMVAR is a local variable. But, it is assigned by CALL SYMPUT in an executing DATA step that will not execute until after the macro has finished. On the other hand the reference &COMMVAR in the %IF statement will resolve during the macro's execution to the null value. Thus, the %ELSE block of code will always be generated.

Again it is a timing issue created by a DATA step function interface to the macro facility. This time, it is the assignment that is delayed, so that the assignment is made in the wrong environment. In the previous example it was access after assignment that was delayed and took place in the wrong macro environment.

Should one conclude that the presence of DATA step function interface to the macro facility is necessary to cause problems? No, there are other ways to assign macro variables which require SAS execution (as opposed to macro execution). Consider

```
%macro m3 ( data = , var = ) ;

   proc sql ;
      select &var from &data ;
      %put sqlobs = &sqlobs ;
      quit ;

   %mend  m3 ;
```

Again one must remember that when M3 is invoked via CALL EXECUTE the execution of the PROC SQL code will be delayed until after the DATA step making the CALL EXECUTE finishes, but that the %PUT will execute during this step; hence before the generated PROC SQL code executes and assigns the correct value to SQLOBS. There will be no error messages when there has been a prior execution of PROC SQL.

Conclusion

Examples have been given to show that the CALL EXECUTE subroutine is a valuable addition to the language. It is particularly powerful when used to invoke macros. On the other hand, one must really understand how SAS code and macro code are processed in order to avoid the pitfalls. Hopefully the examples given here are enough to give one a good intuition about the power and the problems of this technique. Like many powerful features in SAS, CALL EXECUTE is too good to ignore and yet may be dangerous for the unwary.

CALL EXECUTE can dramatically simplify many problems previously handled by CALL SYMPUT and arrays of macro variables. Although somewhat messier, PUT statements to write SAS code and then the use of %INCLUDE to execute the code can often be used in place of CALL EXECUTE. If PUT statements are preferred, then remember the advantages of putting the code in the work directory either directly or via a source catalog.

When using CALL EXECUTE, one should distinguish two kinds of character data

- constant text which is dumped into the input stack for later execution
- macro instructions which are executed immediately

I have had a hidden agenda. In addition to looking at CALL EXECUTE, I wanted to bring out the importance of recognizing and understanding the four crucial times in the execution of a SAS program:

1. Macro compile time
2. Macro execution time
3. SAS compile time
4. SAS execution time

SAS is a rich and different language from all the more traditional programming languages because of the intertwining of these times in a SAS program.

The author can be contacted by mail

H. Ian Whitlock
Westat
1650 Research Boulevard
Rockville, MD 20850-3129

or e-mail

Whitloi1%westat@mcimail.com

References

Riba, S. David (1996). SELF-MODIFYING SAS® PROGRAMS: A DATA STEP INTERFACE. *Proceedings of the Fourth Annual SoutEast SAS Users Group Conference.* SESUG Atlanta, GA. pp. 42-49.

Riba, S. David (1996). SELF-MODIFYING SAS® PROGRAMS: A DATA STEP INTERFACE. *Proceedings of the Ninth Annual NorthEast SAS Users Group Conference.* NESUG Boston, MA. pp. 188-195.

SAS Institute, Inc., *SAS® Technical Report P-222, Changes and Enhancements to Base SAS® Software, Release 6.07,* Cary, NC. SAS Institute Inc., 1991. pp 311-312.

Whitlock, H. Ian (1994). CALL EXECUTE Versus CALL SYMPUT. *Proceedings of the Seventh Annual NorthEast SAS Users Group Conference.* NESUG, Philadelphia, PA. pp. 254-255.