

SELF-MODIFYING SAS® PROGRAMS: A DATA STEP INTERFACE

S. David Riba, JADE Tech, Inc., Clearwater, FL

ABSTRACT

There are many situations where it is useful to have a SAS® program modify itself during execution. Some examples include:

- ♦ conditionally execute a macro based on the data in a SAS dataset
- ♦ conditionally execute a procedure when there is no data in a SAS dataset
- ♦ generate in-line SAS program code based on the contents in a SAS dataset
- ♦ write SQL statements for the SQL Pass-Through Facility "on-the-fly"

This paper explores a little-known routine in the SAS System that allows a SAS program to write SAS code "on-the-fly" during execution. Since CALL EXECUTE is a DATA step CALL routine, the full capabilities of the DATA step can be applied to conditionally generate SAS program statements while the program is executing. If you have ever written program code to a temporary file and then %INCLUDEd it back into your program, you need to take a closer look at CALL EXECUTE. The paper will discuss the uses of CALL EXECUTE with specific examples based on real world situations.

BACKGROUND

SAS programs typically are processed sequentially, one step following another. With the exception of the Macro facility, there is no other method available to alter the execution of the sequential steps in a SAS program. The SAS programmer who has tried to change the flow of a SAS program based on conditions encountered while executing the program has had few options other than coding a macro to execute conditionally.

The limitation to this approach becomes apparent when the following code sample is examined:

```
IF (some-condition-is-true) THEN
    %a_macro ;
ELSE %b_macro ;
```

The intent is to conditionally invoke either macro *a_macro* or macro *b_macro* based on conditions encountered in the DATA step. Unfortunately, the SAS macro processor resolves the macro invocations long before the DATA step executes the conditional test. In this example, the result is that BOTH macro *a_macro* AND macro *b_macro* are resolved BEFORE the start of the DATA step.

Another common example that illustrates the problem with step-boundary processing is the situation that occurs when a SAS program is required to execute a PROCEDURE on a SAS dataset that has no observations. Each step in a SAS program compiles and executes independently of every other step. Normally, there is no option to prevent PROC steps such as PROC SORT or PROC PRINT from compiling and executing on a null dataset other than enclosing the program code within a macro that conditionally executes them.

A third example is the use of a DATA step to conditionally generate in-line SAS program code. Traditionally, this problem has been resolved by writing to an external file and then %INCLUDEing the contents of the external file into the program stream. The limitations to this method include:

- ♦ the need to allocate an external file with read/write access
- ♦ additional I/O required to write to, and read from, an external file
- ♦ the need to perform a cleanup operation to delete the external file

Finally, there is the need to generate SQL statements for the SQL Pass-Through Facility. Typically, variable information can not be included in these statements. The program must contain the exact query statement needed to "pass-through" the SQL gateway. Variable substitution, such as the values in a SAS dataset, can not be used when creating the SQL query.

These are some of the types of problems that CALL EXECUTE can be useful in addressing.

ABOUT CALL EXECUTE

Documentation about CALL EXECUTE appeared in SAS Technical Report P-222 where it was included as a DATA step interface to the Macro facility. CALL EXECUTE was actually available in Version 5 of the SAS System, but disappeared when Version 6 was released. It was re-introduced in Release 6.07 "for backward compatibility". Other than the very brief discussion in P-222, there does not appear to be much information about this CALL routine or its use in any of the SAS literature.

The purpose of CALL EXECUTE is to place program instructions on the SAS program stack. The program stack is then read and executed before SAS compiles the next step in the SAS program. Macro instructions execute immediately and are then placed on the program stack, while non-macro instructions execute as soon as the current DATA step finishes.

Since CALL EXECUTE is a DATA step routine, the full functionality of the DATA step interface is available. CALL EXECUTE arguments are not seen by the SAS supervisor until they are conditionally executed.

The syntax to use CALL EXECUTE is simply

```
CALL EXECUTE ( argument ) ;
```

To place text or an unresolved macro expression on the program stack, enclose the argument in single quotes:

```
CALL EXECUTE ( 'argument' ) ;
```

Since CALL EXECUTE is a DATA step call routine, it can be conditionally invoked in a DO loop or as part of an IF.. THEN.. ELSE construction.

USING CALL EXECUTE TO CONTROL MACRO RESOLUTION

CALL EXECUTE can be used to conditionally place the result of a macro invocation on the SAS program stack. To properly implement the example on the prior page using CALL EXECUTE, the example could have been re-written as:

```
IF (some-condition-is-true) THEN
    CALL EXECUTE('%a_macro') ;
ELSE CALL EXECUTE('%b_macro') ;
```

In the example above, either macro A_MACRO or macro B_MACRO will be executed immediately based on evaluation of the conditional logic. It is important to note in the above example that some conditional test should be added to limit implementation of the CALL EXECUTE to a single occurrence. Otherwise, every observation in the dataset will invoke either macro A_MACRO or macro B_MACRO consecutively.

Arguments to CALL EXECUTE are not resolved immediately. Instead, all of the arguments to CALL EXECUTE are only acted upon conditionally, if the condition evaluates as true. If the argument to CALL EXECUTE is a macro, the macro executes immediately. If the argument to CALL EXECUTE is SAS code, the code is placed on the program stack and executes immediately after termination of the current DATA step. It is also very important to note that you can not use CALL EXECUTE to invoke macros that contain CALL SYMPUTs. The macro will resolve immediately, while the reference created by CALL SYMPUT will queue on the stack for execution after the DATA step has ended.

USING CALL EXECUTE TO GENERATE IN-LINE SAS PROGRAM CODE

Another problem that can occur is the need to change the order of a SAS program based on conditions in the data. For example, it is often useful to be able to bypass sorting and printing null datasets. The typical manner to deal with this problem is to enclose the PROC steps within a macro, and then conditionally invoke the macro.

A sample session might look like this:

```
DATA _null_ ;
    CALL SYMPUT('numobs',put(nobs,5.));
    STOP ;
    SET sasdsn nobs=nobs ;
RUN ;

%MACRO hidethis ;
    %if &numobs %then %do ;
        PROC SORT data=sasdsn ;
            BY varlist ;
        RUN ;
        PROC PRINT data=sasdsn ;
            BY varlist ;
        RUN ;
    %end ;
%MEND hidethis ;

%hidethis ;
```

This sample code first determines if there are observations in a SAS dataset. It then invokes a macro named HIDE THIS which then issues a PROC SORT and PROC PRINT conditionally.

Contrast the above with the use of CALL EXECUTE to accomplish the same task:

```
DATA _null_ ;
  IF (nobs) THEN DO ;
    CALL EXECUTE
      ('PROC SORT data=sasdsn;');
    CALL EXECUTE(' BY varlist ;' ) ;
    CALL EXECUTE('RUN ;' ) ;
    CALL EXECUTE
      ('PROC PRINT data=sasdsn;');
    CALL EXECUTE(' BY varlist ;' ) ;
    CALL EXECUTE('RUN ;' ) ;
  END ;
  STOP ;
  SET sasdsn nobs=nobs ;
RUN ;
```

The above example accomplished the same results as the prior example using only a single DATA step. When there are observations in the SAS dataset to be acted upon, CALL EXECUTE places both PROC steps on the program stack to execute immediately after the DATA step. Thus eliminating the need to clutter the SAS macro environment with small, limited purpose macros and additional processing steps.

It is important to note that all of the above separate CALL EXECUTE statements could have been included in a single CALL EXECUTE statement, as long as the length of the text did not exceed 256 characters. The arguments to CALL EXECUTE have been presented individually in each of the examples in this paper for clarity.

In fact, since regular DATA step conditional logic is evaluated before the CALL EXECUTE statement is acted upon, this example could have been extended one step further with an optional report to print if there are no observations in the dataset to process.

```
DATA _null_ ;
  IF (nobs) THEN DO ;
    CALL EXECUTE
      ('PROC SORT data=sasdsn;');
    CALL EXECUTE(' BY varlist ;' ) ;
    CALL EXECUTE('RUN ;' ) ;
    CALL EXECUTE
      ('PROC PRINT data=sasdsn;');
    CALL EXECUTE(' BY varlist ;' ) ;
    CALL EXECUTE('RUN ;' ) ;
```

```
END ;
ELSE DO ;
  FILE print;
  PUT @10
    "No observations available";
END ;
STOP ;
SET sasdsn nobs=nobs ;
RUN ;
```

How does CALL EXECUTE compare with writing an external file and then %INCLUDEing it into the program stream? Traditionally, the above program code could also have been written as follows:

```
FILENAME external
  'some-external-file-reference' ;
DATA _null_ ;
  FILE external ;
  IF (nobs) THEN DO ;
    PUT 'PROC SORT data=sasdsn;';
    PUT ' BY varlist ;' ;
    PUT 'RUN ;' ;
    PUT 'PROC PRINT data=sasdsn;';
    PUT ' BY varlist ;' ;
    PUT 'RUN ;' ;
  END ;
  ELSE DO ;
    FILE print;
    PUT @10
      "No observations available";
  END ;
  STOP ;
  SET sasdsn nobs=nobs ;
RUN ;

%INCLUDE external ;
```

This method of generating a SAS program requires the following additional steps:

- allocating an external file reference
- writing to the external file
- reading the external file back into the SAS program stream with %INCLUDE
- deleting the external file, or leaving it permanently in place

CALL EXECUTE, on the other hand, writes directly to the program stack without the need to allocate files, the I/O to write or read external files, or the processing to clean up temporary files afterwards.

With CALL EXECUTE you can also pass the values of variables into the code you write to the stack by removing the apostrophe delimiters.

For example, to conditionally define a Title based on the values of a character variable:

```
header1 = 'title "Report Title1";' ;
header2 = 'title "Report Title2";' ;

IF (some-condition-is-true) THEN
  CALL EXECUTE ( header1 ) ;
ELSE
  CALL EXECUTE ( header2 ) ;
```

USING CALL EXECUTE TO GENERATE IN-LINE SQL PROGRAM CODE FOR SAS/ACCESS®

One final example of the utility of CALL EXECUTE to generate in-line program code is to bypass a limitation of SAS/ACCESS®. SQL queries using the SQL Procedure Pass-Through Facility of SAS/ACCESS® are limited to hard coded values and can not include the contents of SAS variables.

The following is an example of the more traditional approach to dealing with this limitation, and is reprinted from *SQL Joins - The Long and The Short of It*, presented by Paul Kent at SUGI 20. This example writes the body of an SQL query to a temporary file defined by fileref TEMP. It inserts the contents of the variable KEY from SAS dataset UNIQ into the query. It then copies the contents of the file that was written into the body of an SQL query with a %INCLUDE.

```
%let chunk=105 ;

proc sql;
  create view uniq as
  select unique key
  from small
  order by key ;

data _null_ ;
  file temp ;
  set uniq end=end ;
  if _n_ = 1 then do ;
    put "create table result as"
      / " select key,data"
      / " from connection to dbms"
      / " (select key,data"
      / "from large where key in ( "
      / key ;
  end ;
  else if mod(_n_,&chunk) = 0
    and not end then do ;
    put "));" // ;
    put "insert into result"
      / " select key,data"
      / " from connection to dbms"
      / " (select key,data"
```

```
      / " from large where key in("
      / key ;
  end ;
  else if end then do ;
    put key "));" // ;
  end ;
  else put key "," ;
run ;

proc sql ;
  connect to <DBMS> as dbms ;
  %inc temp ;
```

This example can be re-written to eliminate the file I/O using CALL EXECUTE as follows:

```
%let chunk=105 ;

proc sql;
  create view uniq as
  select unique key
  from small
  order by key ;

data _null_ ;
  set uniq end=end ;
  if _n_ = 1 then do ;
    CALL EXECUTE( 'proc sql ; ' ) ;
    CALL EXECUTE(
      'connect to <DBMS> as dbms ;' ) ;
    CALL EXECUTE(
      'create table result as ' ) ;
    CALL EXECUTE(
      'select key,data ' ) ;
    CALL EXECUTE(
      'from connection to dbms ' ) ;
    CALL EXECUTE(
      '(select key,data ' ) ;
    CALL EXECUTE(
      'from large where key in( ' ) ;
    CALL EXECUTE( key ) ;
  end ;
  else if mod(_n_,&chunk) = 0
    and not end then do ;
    CALL EXECUTE( ')); ' ) ;
    CALL EXECUTE(
      'insert into result' ) ;
    CALL EXECUTE(
      'select key,data' ) ;
    CALL EXECUTE(
      ' from connection to dbms' ) ;
    CALL EXECUTE(
      '(select key,data' ) ;
    CALL EXECUTE(
      'from large where key in(' ) ;
    CALL EXECUTE( key ) ;
  end ;
  else if end then
    CALL EXECUTE(
      ',' || key || '));' ) ;
  else
```

```
CALL EXECUTE( key || ' , ' ) ;
run ;
```

As noted above, the individual CALL EXECUTE statements could have been combined into several much larger concatenated calls, but are presented individually for clarity.

USING CALL EXECUTE TO LOAD AN EXCEL SPREADSHEET FROM A SAS DATASET

The above example actually illustrates a solution to a very real problem with the SQL Pass-Through Facility in SAS. For example, SAS/Access to ODBC® uses the SQL Pass-Through Facility to issue SQL commands to the ODBC Driver Manager. However, these SQL commands can not contain variable information. Thus, it is not possible to directly load an Excel spreadsheet with the contents of a SAS dataset.

The following sample code illustrates the use of CALL EXECUTE to build a spreadsheet in Microsoft Excel from a SAS dataset:

The initial section creates the SQL code to define the Excel spreadsheet. It places the PROC SQL code on the program stack to create and define a spreadsheet using ODBC. This section executes only once at the beginning of the DATA step.

```
data _null_ ;
  set report end=eof ;
  if (_N_ eq 1) then do ;
    call execute('proc sql feedback ; ');
    call execute('connect to odbc as cls
      (prompt) ' ) ;
    call execute('execute (drop table
      [Sheet1] ) by cls ; ' ) ;
    call execute('execute (create table
      [Sheet1] ' ) ;
    call execute(' (PROTOCOL int, SITE
      int, PATIENT int, ID char(15), ');
    call execute(' SC_NO char(15),
      STUDYWEEK char(15) , ' ) ;
    call execute(' BLEEDDATE char(15),
      DATESHIPPED char(15), ' ) ;
    call execute(' ACCESSNO int, ACCESSDT
      char(15), PLATEID char(15), ' ) ;
    call execute(' RES VALU char(15),
      MRL_UNIT char(15) ' ) ;
    call execute(' ) ) by cls ; ' ) ;
  END ;
```

The middle section loads the values from a SAS dataset into the spreadsheet. Note that it combines the values of variables with character formatting information. This section executes once for each

observation in the dataset. The resolved values of the variables in the dataset are placed on the stack. Thus, the problem with the SQL Pass-Through Facility is resolved before the code is placed on the stack.

```
call execute(' execute (insert into
  [Sheet1] values ( ' ) ;
call execute(PROTOCOL) ;
call execute(' , ' ) ;
call execute(SITE) ;
call execute(' , ' ) ;
call execute(PATIENT) ;
call execute(' , ' ) ;
call execute("'" || trim(ID) || "'" ) ;
call execute(' , ' ) ;
call execute("'" || trim(SCN) || "'" ) ;
call execute(' , ' ) ;
call execute("'" || trim(S_WEEK) || "'" ) ;
call execute(' , ' ) ;
call execute(
  "'" || put(B_DT,mmddyy8.) || "'" ) ;
call execute(' , ' ) ;
call execute(
  "'" || put(S_DT,mmddyy8.) || "'" ) ;
call execute(' , ' ) ;
call execute(ACCESSNO) ;
call execute(' , ' ) ;
call execute("'" ||
  put(A_DT,datetime16.) || "'" ) ;
call execute(' , ' ) ;
call execute("'" || trim(P_ID) || "'" ) ;
call execute(' , ' ) ;
call execute(
  "'" || trim(RES_VALU) || "'" ) ;
call execute(' , ' ) ;
call execute("'" || trim(UNIT) || "'" ) ;
call execute(' ) ) by cls ; ' ) ;
```

This final section executes only once and performs the cleanup necessary to complete the PROC SQL step.

```
if (eof) then do ;
  call execute(
    'disconnect from cls ; ' ) ;
  call execute('quit ; ' ) ;
end ;
run ;
```

COMBINING CALL EXECUTE WITH SCL FOR A HYBRID SOLUTION

Starting with Release 6.11 of the SAS System, it is now possible (on an experimental basis) to use some of the Screen Control Language functions in Base SAS programming. The example below has been extracted from a larger program. It uses CALL

EXECUTE to generate SAS program code to create a SAS dataset and define several variables using some of the attributes from a second SAS dataset.

```
DATA _null_ ;
  call execute(
    'data sasdsn1; set sasdsn; label ');
  dsid = OPEN('sasdsn') ;
  nvars = ATTRN(dsid,'nvars') ;
  DO i = 1 to nvars ;
    name = varname(dsid,i) ;
    label = varlabel(dsid,i) ;
    call execute (name|| '=' ||label);
  END ;
  call execute ('; RUN ;') ;
  STOP ;
RUN;
```

CAVEATS WHEN USING CALL EXECUTE

Since CALL EXECUTE can be used to provide a direct interface with the macro facility, a caution must be raised about the results of improper use.

Excessive interaction between the DATA step and the macro facility will slow down execution of your program code. For example, the following code sample takes much longer to process using CALL EXECUTE than through alternate means of coding:

```
%LET namelist = ;

DATA _null_ ;
  INPUT dsname memtype ;
  CALL EXECUTE ('%LET namelist =
    &namelist'||trim(left(dsname)) ) ;
```

which builds a macro variable NAMELIST containing the names of all datasets in a particular library. However, the repetitive interaction between CALL EXECUTE and the macro facility (redefining a macro variable for every pass through the DATA step) cause the DATA step to take an exceptionally long time to process. A better solution would have been to develop the value for NAMELIST during each pass of the DATA step, and invoke CALL EXECUTE when the last record was being processed.

Another critical issue can be defined as "timing". Since CALL EXECUTE can interact directly with both the macro processor and the SAS program stack, the resolution of the arguments to CALL EXECUTE may yield unexpected results. If the arguments are valid SAS code, they will be placed on the SAS program stack for execution at the close of the current DATA step. If the arguments are

macro statements, they may cause an error condition if they:

- ♦ attempt to reference an unresolved macro variable
- ♦ attempt to modify the currently executing DATA step which has already been compiled
- ♦ contain SAS program code that unexpectedly terminates the currently executing step

For example:

```
%MACRO addobs (name = ) ;
  %do i = 1 %to 3 ;
    name2 = "&name" ;
    pass = &i ;
    output ;
  %end ;
%MEND addobs ;

DATA sasdsn1 ;
  SET sasdsn ;
  call execute (
    '%addobs (name=' || name || ')' ) ;
  RUN ;
```

The intent in the example above is to invoke a macro named ADDOBS in the middle of a DATA step to perform additional DATA step processing. However, since the SAS code to be invoked is DATA step code, the SAS supervisor will place it on the stack for execution at the end of the currently executing DATA step. This will create an error condition, since the next "step" that will be submitted for execution is DATA step code outside of a DATA step.

To resolve problems like this, the example could have been recoded as:

```
DATA _null_ ;
  SET sasdsn end=eof;
  IF (_n_ eq 1) THEN do ;
    call execute('DATA sasdsn1;');
    call execute('SET sasdsn ');
  end ;
  call execute (
    '%addobs (name=' || name || ')' ) ;
  IF (eof) THEN
    call execute('RUN ;') ;
  RUN;
```

Note the submission of a RUN statement as part of the CALL EXECUTE instructions. It is prudent to terminate the DATA step to define the step boundary explicitly.

Below are several additional examples based on the above which illustrate the interaction between CALL EXECUTE and the macro processor.

Assume that dataset SASDSN1 contains three records each with values of Fred, Tom, and Judy for the variable NAME:

Example 1

```
DATA _null_ ;
  SET sasdsn1 end=eof;
  IF ( _n_ eq 1) THEN do ;
    call execute('DATA sasdsn ;') ;
    call execute(' SET sasdsn1 ;') ;
  END ;
  call symput('tname',name) ;
  call execute (
    'newname = symget("tname") ;');
  call execute (
    'put name= newname= ;');
  IF (eof) THEN call execute('RUN;');
RUN ;
```

Example 1 uses the resolved value of the variable TNAME at execution. The result is:

```
NAME=Fred NEWNAME=Judy
NAME=Fred NEWNAME=Judy
NAME=Fred NEWNAME=Judy
NAME=Tom NEWNAME=Judy
NAME=Tom NEWNAME=Judy
NAME=Tom NEWNAME=Judy
NAME=Judy NEWNAME=Judy
NAME=Judy NEWNAME=Judy
NAME=Judy NEWNAME=Judy
```

Example 2

```
DATA _null_ ;
  SET sasdsn1 end=eof;
  IF ( _n_ eq 1) THEN do ;
    call execute('DATA sasdsn ;') ;
    call execute(' SET sasdsn1 ;') ;
  END ;
  call symput('tname',name) ;
  call execute (
    'newname = "&tname" ;');
  call execute (
    'put name= newname= ;');
  IF (eof) THEN call execute('RUN;');
RUN ;
```

Example 2 uses the current value of the macro variable TNAME when placing SAS code on the program stack. The result is:

```
NAME=Fred NEWNAME=Fred
NAME=Fred NEWNAME=Tom
```

```
NAME=Fred NEWNAME=Judy
NAME=Tom NEWNAME=Fred
NAME=Tom NEWNAME=Tom
NAME=Tom NEWNAME=Judy
NAME=Judy NEWNAME=Fred
NAME=Judy NEWNAME=Tom
NAME=Judy NEWNAME=Judy
```

In these examples, the issue of when the macro variable resolves is crucial in obtaining accurate results.

Below is an example of how the SAS code can be modified to process correctly.

Example 3

```
DATA _null_ ;
  SET sasdsn1 end=eof;
  IF ( _n_ eq 1) THEN do ;
    call execute('DATA sasdsn ;') ;
    call execute('SET sasdsn1 ;') ;
  END ;
  call execute (
    'call symput("tname",name); ');
  call execute (
    'newname = symget("tname") ;');
  call execute (
    'put name= newname= ; ');
  IF (eof) THEN call execute('RUN;');
RUN ;
```

The result of the above is:

```
NAME=Fred NEWNAME=Fred
NAME=Fred NEWNAME=Fred
NAME=Fred NEWNAME=Fred
NAME=Tom NEWNAME=Tom
NAME=Tom NEWNAME=Tom
NAME=Tom NEWNAME=Tom
NAME=Judy NEWNAME=Judy
NAME=Judy NEWNAME=Judy
NAME=Judy NEWNAME=Judy
```

In this example, the macro values for variables are defined and resolved in the same data stream and yield the correct results.

CONCLUSION

CALL EXECUTE is a versatile addition to the repertoire of any SAS programmer who requires the ability to modify a SAS program based on conditions evaluated while the program is executing. It can be useful in selecting between different macros in a DATA step, in conditionally executing procedures, and in writing SQL statements for the SQL Procedure Pass-Through Facility. To avoid

unexpected results, it is important to have an understanding of the interaction between the Macro Processor and the SAS Supervisor when using CALL EXECUTE.

REFERENCES

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 71, 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.

Musterer, Robert and Jiang, Owen (1995). Use of the Call Execute Routine to Pass a List of Variables From a DATA Step to a PROC Step. *Proceedings of the Eighth Annual NorthEast SAS Users Group Conference*. NESUG, Washington, DC. pp. 552-554.

Kent, Paul (1995). SQL Joins - The Long and The Short of It. *Proceedings of the Twentieth Annual SAS Users Group International Conference*. SAS Institute, Inc, Cary, NC. pp. 206-215.

TRADEMARK INFORMATION

SAS is a registered trademark of SAS Institute Inc. in the USA and other countries. ® indicates USA registration.

AUTHOR

The author welcomes comments, suggestions, and questions by phone (813-726-6099) or by e-mail JADETECH@MCIMAIL.COM.

The author may be contacted at:

S. David Riba
JADE Tech, Inc.
P O Box 4517
Clearwater, FL 34618
(813) 726-6099
INTERNET: jadetech@mcimail.com

