Paper 027-30

CALL EXECUTE: A Powerful Data Management Tool

Denis Michel, Johnson & Johnson Pharmaceutical Research and Development, L.L.C.

ABSTRACT

There are often data management challenges that require automation of code generation as the optimal solution. The SAS® System is rich in tools that provide for automated code generation, including the macro facility and the CALL EXECUTE routine. After using CALL EXECUTE to solve some specific tasks, the routine becomes an indispensable tool to the programmer for solving general problems.

This paper describes how to use CALL EXECUTE, presents some caveats about the use of the routine, and provides programs that perform specific data management tasks by using CALL EXECUTE to generate SAS statements.

INTRODUCTION

The CALL EXECUTE routine resolves an argument and executes the resolved value at the next step boundary. The syntax is simple:

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

There are three types of arguments that can be used. First, the value of argument can be a text string enclosed in quotes, as in the following examples.

```
CALL EXECUTE (‘proc print data=sugi30.demog; run;’);
CALL EXECUTE (‘%print (dsn=sugi30.demog) ’);
```

Note that the call to the %print macro is enclosed in single quotes. This is important because CALL EXECUTE resolves arguments enclosed in single quotes during program execution. Arguments within double quotes resolve while the data step is being constructed, which will often cause syntax errors.

Second, argument can be the name of a SAS data step character variable containing SAS statements. In this case, we do not enclose the argument in quotes. For example, data set sugi30.prtsamp may contain a character variable prtcode with many values as partially shown below.

```
Obs          prtcode

 1    proc print data=sugi30.ae; run;
 2     proc print data=sugi30.demog; run;
 3     ….
```

We could execute all of the SAS statements contained in variable prtcode in every observation by using the following code.

```
data _null_;
 set sugi30.prtsamp;
 call execute (prtcode);
run;
```

The third type of argument is a character expression that is resolved by the DATA step to macro text or SAS statements. This type of argument allows us to generate SAS code within data steps. This is powerful and flexible, and allows us to solve data management tasks with straightforward, understandable programs. Here is a simple example. We want to print the first 50 observations of all data sets in a SAS data library. The program below uses CALL EXECUTE to generate the SAS code needed.

```
***Read SAS data sets in library;
proc sql;
  create table _dsets_ as
    select memname
      from dictionary.tables
      where libname='SUGI30';
quit;

***Print the first 50 observations of the data sets;
***Include titles identifying data set - watch your quotes!!!;
data _null_;
 set _dsets_;
 call execute
 (
  'proc print data=sugi30.' ||
  trim(memname) ||
  '(obs=50); title ''First 50 observations of data set ' ||
   trim(memname) || '''; run;'
 );
run;
```

We read the data set names in data library SUGI30 using PROC SQL dictionary tables to create temporary data set _DSETS_. In this case, we are using a single CALL EXECUTE routine to generate the PROC PRINT and TITLE statements, as well as the RUN statement. We could have used separate CALL EXECUTE statements because the routine is very flexible. CALL EXECUTE simply sends the generated code to the SAS input stack for execution after the data step executes.

We use the concatenate operator to combine character strings with the data set names read from _DSETS_ during the execution of the data step (data _NULL_ step). All single quotes are used in this code. Since we quote the text strings sent to CALL EXECUTE, we use two single quotes after the title statement to resolve to one single quote sent for SAS execution. After concatenating the data set name from variable MEMNAME, we use three single quotes. The first quote is to quote the text string for CALL EXECUTE and the next two single quotes resolve to one single quote ending the title statement for execution by SAS. With OPTIONS SOURCE, the SAS log would contain the following lines.

```
NOTE: CALL EXECUTE generated line.

proc print data=sugi30.AE(obs=50); title 'First 50 observations of data set AE'; run;

proc print data=sugi30.DEMOG(obs=50); title 'First 50 observations of data set DEMOG';
run;
```

CAVEATS

Like most powerful tools, CALL EXECUTE has features that require understanding for proper usage. Some of the known caveats are presented here in an effort to learn from previous mistakes and avoid tedious program debugging and frustration.

MACRO TIMING

The most common problem causing SAS errors and unexpected results involves the timing of macro references and SAS statements. It is important to know that macro references within CALL EXECUTE are executed immediately, but SAS language statements within CALL EXECUTE do not execute until after the data step containing CALL EXECUTE is executed. SAS statements generated by macros also execute after the data step has executed. That is why you cannot use CALL EXECUTE to invoke a macro that contains references to macro variables created by CALL SYMPUT in that macro. One of our examples later will use both CALL SYMPUT and CALL EXECUTE properly.

QUOTING

Since the arguments passed to CALL EXECUTE are usually quoted, we have to use care when generating code. Single quotes are used to prevent macro references from executing immediately. When generating TITLE and LABEL statements, we have to ensure we do not have unbalanced quotes.

/* PL/I STYLE COMMENTS MUST BEGIN AND END IN THE SAME CALL EXECUTE */

CALL EXECUTE is very flexible. The routine allows you to submit partial SAS statements to the input stack. The following code works (but is not a good practice).

```
data _null_;
    call execute('data ');
    call execute('test; x=');
    call execute('1; output; run; proc');
    call execute(' print; run;');
   run;
```

The exception is for PL/I style comments (/*   */), which must be in the same CALL EXECUTE statement. The following code generates a SAS error.

```
data _null_;
    call execute('/* Start comment');
    call execute(' More comment text.');
    call execute(' end comment */');
   run;
```

CALL EXECUTE MAY TRUNCATE OR REPEAT GENERATED CODE IN RELEASE 8.2

A documented problem exists in SAS release 8.2. Using CALL EXECUTE to build large amounts of code (generally more than a thousand lines of generated code) may cause the code that is generated to be truncated or unexpectedly repeated. A Technical Support hot fix for Release 8.2 TSLEVEL TS2M0 for this issue is available. See SAS documentation SN-005243. The problem is resolved in SAS release 9.

EXAMPLES

Specific data management tasks will be solved using CALL EXECUTE. It will be evident that utility macros containing the routine can be written to solve various challenges.

1. FILTER ALL DATASETS BASED ON VALUES IN ONE DATASET

In this example of a table lookup, we will filter a SAS data library containing many datasets with information about unique study subjects (variable USUBJID). Our task is to filter the data from permanent data library SUGI30 to WORK library datasets including subjects that have any serious adverse events, contained in dataset AE, variable AESER, with values of “Y”.

```
*** Select the subjects with serious adverse events;
*** Generate a control input dataset for PROC FORMAT;
proc sort data=sugi30.ae(keep=usubjid aeser)
          out=cntlin(rename=(usubjid=start))
          nodupkey;
 where aeser=’Y’;
 by usubjid;
run;
*** Control input dataset requires variables fmtname and label;
data cntlin;
 set cntlin;
 retain fmtname ‘$SAE_SEL’ label ‘SAE_YES’;
run;
*** Generate format $SAE_SEL;
proc format cntlin=cntlin;
run;

*** Read datasets with subject data;
proc sql;
  create table _dsets_ as
    select memname
      from dictionary.columns
      where libname=’SUGI30’ and name='USUBJID';
quit;

*** Now use CALL EXECUTE to subset all datasets;
data _null_;
  set _dsets_;
  call execute(
               ‘data ‘
               || trim(memname)
               || ‘; set sugi30.‘
               || trim(memname)
               || ‘; where put(usubjid, $SAE_SEL.) = ‘’SAE_YES’’ ; run; ’
              );
run;
```

The table lookup is performed using the FORMAT procedure and PUT function, which is more efficient than using sort and merge. The CALL EXECUTE routine is used in the execution of the filtering code. The SAS log would show the generated code.

```
NOTE: CALL EXECUTE generated line.
data AE; set sugi30.AE; where put(usubjid, $SAE_SEL.) = 'SAE_YES' ; run;
data CONMED; set sugi30.CONMED; where put(usubjid, $SAE_SEL.) = 'SAE_YES' ; run;
data DEMOG; set sugi30.DEMOG; where put(usubjid, $SAE_SEL.) = 'SAE_YES' ; run;
data EXPOSURE; set sugi30.EXPOSURE; where put(usubjid, $SAE_SEL.) = 'SAE_YES' ; run;
```

This example uses all single quotes. We use two single quotes around SAE_YES to resolve to a single quote. Alternatively, we could have used double quotes in the CALL EXECUTE argument as follows, to generate the same SAS code. Here is the revised line.

```
                    || “; where put(usubjid, $SAE_SEL.) = ‘SAE_YES’ ; run; “
```

Note that the program code above can be modified for different table lookups. In the example, we read unique subject numbers that have serious adverse events. For example, if 10 subjects have 20 serious adverse events, and these 10 subjects have a total of 100 adverse events (20 serious and 80 non-serious), the output dataset has all 100 observations. Similarly, we have output datasets for these 10 subjects containing all data available in all datasets. We can print the datasets, export them to MS Excel, or perform any other requirement. This is powerful stuff!

2. COMPARE DATASETS ACROSS DATA LIBRARIES

In cases where we have two versions of a data library, we need an automated way of comparing datasets to ensure that there are no unexpected changes. We may work on a data migration, where we need to compare the data across data library locations, or we may receive a new data library for a project that was previously “final” and “locked”. We may also change a macro that outputs data and want to see the results of the change. The following macro compares two data libraries (&old and &new) and reports differences.

```
%macro mcomp(old=,
             new=);

title3 "Old data library=&old ==== New data library=&new";
***Read SAS data sets in both libraries;
proc sql;
  ***Datasets only in old data library;
  create table _onlyold as
     select memname
      from dictionary.tables
      where libname=%upcase("&old")
      and memname ^in(
                     select memname
                       from dictionary.tables
                       where libname=%upcase("&new")
                    );

  ***Datasets only in new data library;
  create table _onlynew as
     select memname
      from dictionary.tables
      where libname=%upcase("&new")
      and memname ^in(
                     select memname
                       from dictionary.tables
                       where libname=%upcase("&old")
                    );

  ****Datasets in common to compare;
  create table _dsets_ as
    select memname
      from dictionary.tables
      where libname=%upcase("&old")
      and memname in(
                     select memname
                       from dictionary.tables
```

```
                      where libname=%upcase("&new")
                   );
quit;

title4 'Datasets in Old not in New';
proc print data=_onlyold;
run;
title4 'Datasets in New not in Old';
proc print data=_onlynew;
run;
title4;

***Compare each data set;
data _null_;
 set _dsets_;
 call execute
 (
  "proc compare data=&old.." ||
   trim(memname) ||
  " compare= &new.." ||
   trim(memname) || " ;run;"
 );
run;
%mend mcomp;
```

If we define libnames BEFORE and AFTER, and use the following macro call, the output will show the differences across the data libraries, first printing datasets that are not in both data libraries.

```
%mcomp(old=before,
       new=after);
```

The datasets in common are then compared with the code generated by CALL EXECUTE.

```
NOTE: CALL EXECUTE generated line.
proc compare data=before.AE compare= after.AE ;run;
proc compare data=before.DEMOG compare= after.DEMOG ;run;
```

3. GENERATE AND VALIDATE SAS TRANSPORT FILES

Customers often request that all SAS datasets in a data library be provided as SAS transport files that can be browsed with SAS Viewer. This macro generates SAS transport files, and optionally validates the files by comparing them to the original datasets.

```
%*********************************************************************;
%*** MXCOPY.SAS - Copy SAS data library to transport files;
%*** SUGI 30 April 2005;
%*** Specify:   libref : SAS data library reference;
%***            xref   : Directory location for xport files;
%***            Validate : Yes (default) or No validation of output;
%*** NOTE: A libname statement must have previously defined the libref;
%*** Example %mxcopy(libref=rawdata, xref=h:\sugi30\xptraw);
%*********************************************************************;
```

```
%macro mxcopy (libref=, xref=, validate=Yes);

%***Case insensitive;
%let libref=%upcase(&libref);
%let xref=%upcase(&xref);
%let validate=%upcase(&validate);

%***Get the data set names;
proc sql;
  create table _dsets_ as
      select memname
      from dictionary.tables
      where libname="&libref";
quit;

%***Write the transport files;
data _null_;
  set _dsets_;
  call execute(
               "libname " || trim(memname) || " xport  ' "
                || "&xref.\" || trim(memname) || ".XPT'; "
                || "proc copy in=&libref out= "
                || trim(memname) || "; select "
                || trim(memname) || "; run; "
              );
run;

%***Validation;
%***Read transport files to work library and compare to input files;
%if &validate ^= NO %then %do;
 data _null_;
   set _dsets_;
   call execute(
                "proc copy in= "
                || memname || " out=work; run; "
                || "proc compare data=&libref.."
                || trim(memname) || " compare=work."
                || trim(memname) || "; run;"
               );
 run;
%end; %***Validation;

%mend mxcopy;
```

In the macro above, we use double quotes for the text in CALL EXECUTE routine that contain single quotes in the generated LIBNAME statements. If we define libname SUGI30, and use the following macro call, all SAS data sets in data library SUGI30 are copied to transport files in the folder h:\sugi and the transport files are validated.

```
%mxcopy(libref=sugi30,
        xref=h:\sugi);
```

The SAS log shows the generation of the transport files, as in the sample below.

```
NOTE: CALL EXECUTE generated line.
```

```
   libname AE xport  'H:\SUGI\AE.XPT';
   proc copy in=SUGI30 out= AE;
   select AE;
   run;
```

The SAS log also displays the validation of the transport files, accomplished by copying the transport files to temporary data sets and comparing to the original data sets.

```
   NOTE: CALL EXECUTE generated line.
   proc copy in= AE out=work;
   run;
   proc compare data=SUGI30.AE compare=work.AE;
   run;
```

The SAS output should be checked to ensure that PROC COMPARE found no differences.

4. APPLY ATTRIBUTES FROM METADATA

There are cases when SAS datasets are provided without some of the required attributes, which are stored in separate locations. In this example, there is a separate metadata file containing dataset labels and variable labels. The metadata dataset METADATA.ITEMLBL contains variables TBLNAME (dataset name), TBLLABEL (dataset label), VARNAME (variable name) and VARLABEL (variable label) and looks like the following partial print.

```
   Dataset: METADATA.ITEMLBL

   tblname   tbllabel           varname   varlabel

   AE        Adverse Events     STUDYID   Study ID
   AE        Adverse Events     USUBJID   Unique subject number
   AE        Adverse Events     AESEQ     AE Sequence Number
   AE        Adverse Events     AETERM    Reported Term
   CONMED    Concomitant Meds   STUDYID   Study ID
   CONMED    Concomitant Meds   USUBJID   Unique subject number
   CONMED    Concomitant Meds   CMSEQ     Conmed Sequence Number
   CONMED    Concomitant Meds   CMREPRT   Were Any Meds Reported Administered
   DEMOG     Demographics       STUDYID   Study ID
   DEMOG     Demographics       USUBJID   Unique subject number
   DEMOG     Demographics       BIRTHDT   Date of Birth
```

Our task is to apply these dataset labels and variable labels to the datasets in SAS data library SUGI30. Here is the code.

```
data _null_;
 set metadata.itemlbl end=eof;
 by tblname;

 ***Generate the datasets procedure code once at the beginning of the dataset;
 if _n_=1 then call execute('proc datasets library=sugi30 memtype=data nolist;');

 ***Apply dataset labels once per dataset;
 if first.tblname
   then call execute('modify '  || trim(tblname)
                               || ' (label = "'
                               || trim(tbllabel)
                      || '");'
                    );

 ***Apply variable labels;
 call execute('label '
               || trim(varname) || ' = "'
               || trim(varlabel) || '";'
             );

 ***Generate the QUIT statement to terminate PROC DATASETS once;
 *** at the end of the dataset;
 if eof then call execute('quit;');
run;
```

The generated SAS code follows.

```
   NOTE: CALL EXECUTE generated line.
   proc datasets library=sugi30 memtype=data nolist;
   modify AE (label = "Adverse Events");
   label STUDYID = "Study ID";
   label USUBJID = "Unique subject number";
   label AESEQ = "AE Sequence Number";
   label AETERM = "Reported Term";

   modify CONMED (label = "Concomitant Meds");
   label STUDYID = "Study ID";
   label USUBJID = "Unique subject number";
   label CMSEQ = "Conmed Sequence Number";
   label CMREPRT = "Were Any Meds Reported Administered";

   modify DEMOG (label = "Demographics");
   label STUDYID = "Study ID";
   label USUBJID = "Unique subject number";
   label BIRTHDT = "Date of Birth";

   quit;
```

We generated the PROC DATASETS statement once at the beginning of the metadata dataset. For each dataset, we wrote a modify statement to label the dataset. Then we generated label statements for every variable. At the end of the metadata file, we generated a QUIT statement to end PROC DATASETS.

In this example, single quotes are used to enclose the arguments passed to the input stack by CALL EXECUTE. The labels use double quotes. This prevents mismatched quotes from being generated if a dataset or variable label includes a single quote. For example, if a variable INVRESP has the label “Investigator’s assessment of response”, the code above would work fine, generating the line below.

```
   label INVRESP = "Investigator’s assessment of response";
```

5. PRINT DATA BY KEY VARIABLES

In clinical research, we sometimes need to print data by subject, instead of by dataset. Quality control audit listings are printed by subject, so that the data for each subject can be compared to the case report forms from which they were entered. Subject profiles are generated in cases where reviewers want to see data by subject, instead of by dataset. Here is an example of printing all data by subject for a random audit sample size of the square root of the subject population plus one. We require a dataset that contains all subjects (DEMOG) and we'll select a random sample and print the datasets by subject.

```
***=========================================================================;
***QCAUDIT.SAS - Select random sample and print data by unique subject ID;
***SUGI 30 Apr2005;
***=========================================================================;

proc sql;
 ***Get SAS data sets in library that have patient information;
 create table _dsets_ as
  select distinct memname
    from dictionary.columns
    where libname='SUGI30' and name='USUBJID'
    order by memname;

 ***All subjects are in DEMOG dataset;
 create table _allsub_ as
  select distinct usubjid
   from sugi30.demog
   order by usubjid;
quit;

***Assign a random number to each subject;
data _allsub_;
 set _allsub_ end=eof;
 ***Use the current date as the seed and assign random number;
 _ranuni=ranuni(today());
 if eof then do;
    ***Store total number of subjects as a global macro variable;
    call symput('totsubj', left( trim( (put(_n_, 8.)) ) ) );
    ***Sample size is SQRT(_n_) +1 and ensure sample size is not > total size;
    sampsize=min( ceil( sqrt(_n_) +1), _n_);
    call symput('sampsize', left( trim( (put(sampsize, 8.)) ) ) );
 end;
run;
%put =======>Total number of subjects is =====> &totsubj;
%put =======>Random sample size is =====> &sampsize;

***Subset the random sample;
proc sort data=_allsub_ out=_random_;
 by _ranuni;
run;
data _random_;
 set _random_(obs=&sampsize);
run;
***Resort by subject;
proc sort data=_random_;
 by usubjid;
run;
title3 'All subjects';
proc print data=_allsub_;
run;
title3 'Random sample subjects';
```

```
proc print data=_random_;
run;
title3;

***Global macro variables with all unique subject numbers to list;
data _null_;
     set _random_;
     i+1;
     call symput('sn' || left(put(i,4.)), trim(usubjid) );
run;

***QC listing macro;
%macro _qc;
%do i=1 %to &sampsize;
%***Start each subject at page 1;
options pageno=1;

data _null_;
 set _dsets_;
   by memname;

***Titles;
call execute
        (
        'title3 "Subject ID: &&sn&i  Dataset: '||
         trim(memname) ||'"; run;'
        );

***Print the datasets;
call execute
       (
        'proc print data=sugi30.' ||
         trim(memname) ||
        '; where usubjid = "&&sn&i"; run;'
        );

run;
%end;
%mend _qc;

***Call the macro to print the listings;
%_qc
```

We used CALL EXECUTE inside a macro loop to title and print data by subject. The code works because the macro variable containing the sample size (&sampsize) and macro variables containing subject identifiers (&sn1, &sn2, &sn3, etc.) are global macro variables generated previously in open code using CALL SYMPUT. A sample of the generated code in the SAS log follows.

```
NOTE: CALL EXECUTE generated line.
title3 "Subject ID: SUGI-30-001-800007  Dataset: AE"
proc print data=sugi30.AE;
where usubjid = "SUGI-30-001-800007";
run;
title3 "Subject ID: SUGI-30-001-800007  Dataset: CONMED";
```

```
run;
proc print data=sugi30.CONMED;
where usubjid = "SUGI-30-001-800007";
run;
```

In case we have a data set with long text variables that require PROC REPORT instead of PROC PRINT, we can easily modify the macro to accommodate the data. For example, a dataset COMMENTS has a long variable CTTERM (length=$200), which would be truncated if printed with PROC PRINT. We modify the print section of the code above as follows.

```
***Print the datasets;
if memname^=’COMMENTS’ then call execute
       (
        'proc print data=sugi30.' ||
         trim(memname) ||
        '; where usubjid = "&&sn&i"; run;'
        );
  else call execute
       (
        'proc report data=sugi30.’ ||
         memname ||
        '; where usubjid = "&&sn&i";' ||
        ‘column usubjid domain ctseq ctterm;’ ||
        ‘define ctterm / left width=100 flow; run;’
        );
```

With the code above, all datasets other than COMMENTS are printed using PROC PRINT and the COMMENTS dataset is printed with PROC REPORT. Other modifications can be made due to the flexibility of the CALL EXECUTE routine.

CONCLUSION

The CALL EXECUTE routine is a powerful tool, which, when used properly, can solve a multitude of data management tasks. Other techniques can be used to solve these problems, but CALL EXECUTE is often easier to use and modify. Writing SAS code to external files, then using %INCLUDE is an alternative. Another technique is the use of CALL SYMPUT to generate a series of macro variables, which requires macros with a minimum of double ampersands. Different tools can be used to solve the same problems. It is up to the SAS programmer to determine the appropriate tool. CALL EXECUTE and the macro facility provide the flexibility and ease of use to make them worth considering when the next task challenges you.

REFERENCES

SAS Institute Inc., *SAS Macro Language: Reference Version 8*, Cary NC: SAS Institute Inc., 1999.

Croonen, Nancy and Theuwissen, Henri, “Table Lookup: Techniques Beyond the Obvious”, *Proceedings of the Twenty-Seventh Annual SAS Users Group International Conference*, 2002.

Vergile, Bob, “Magic with CALL EXECUTE”, *Proceedings of the Twenty-Second Annual SAS Users Group International Conference*, 1997.

Whitlock, H. Ian, “CALL EXECUTE: How and Why”, *Proceedings of the Twenty-Second Annual SAS Users Group International Conference*, 1997.

Whitlock, H. Ian, “CALL EXECUTE Versus CALL SYMPUT”, *Proceedings of the Seventh Annual NorthEast SAS Users Group Conference*, 1994.

CONTACT INFORMATION

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

Denis Michel
Johnson & Johnson Pharmaceutical Research and Development, L.L.C.
1125 Trenton-Harbourton Road
PO Box 200
Titusville, NJ 08560
Email: dmichel@prdus.jnj.com