Paper 243-2011

SAS® Code and Macros: How They Interact

Bruce Gilsen, Federal Reserve Board

INTRODUCTION

SAS users at every level of experience seem to struggle to understand the interaction between SAS code (DATA and PROC steps) and macros. Understanding this interaction enables SAS users to understand existing applications and to take better advantage of the tools offered by SAS software and develop code much more effectively. This important concept is first reviewed in a somewhat informal way, similar to the one-on-one discussions I have had many times with Federal Reserve Board SAS users, and then with a series of small examples.

Macros Generate Code

The following informal, over-simplified explanation echoes many one-on-one discussions I have had with Federal Reserve Board SAS users. It is intended to provide an intuitive understanding of macro and SAS code processing. For more information, see the examples below. For a more formal and detailed explanation, see the chapters *SAS Programs and Macro Processing* and *Macro Processing* in the *SAS 9.1.3 Macro Language Reference*.

Over 20 years ago, a long-departed SAS expert explained macro execution with a simple mantra: "Macros generate code." I still consider this an extremely helpful way to start thinking about macro execution. I think of macros as containing two things.

- *Macro language elements (%DO, %IF, %PUT, %STR, etc.)*
- *SAS (non-macro) code that the macro processor treats as literal text*

During macro execution, the macro processor executes macro language elements and generates zero or more lines of SAS code. When macro processing completes, or at any step boundary, if any lines of SAS code were generated, they are processed as if the user typed them at that location.

Understanding the interaction of SAS (DATA and PROC step) code and macros enables users to understand existing applications and easily code applications that might otherwise be much more difficult. For example, macros can be used to easily generate SAS code that can be

- a large block of code such as a DATA or PROC step or multiple steps
- a single SAS statement
- part of a SAS statement

Let's consider some examples. Examples 1 - 3 are fairly simple. Examples 4 and 5 are slightly less simple, but show an important coding concept: how to iteratively generate SAS code in a macro. Example 6 is more complex and illustrates a more automated application. Examples 7 and 8 illustrate more advanced topics: retrieving macro variables in a DATA step and generating SAS code in a DATA step with CALL EXECUTE.

In the examples below, SAS code being "generated" means that the SAS code is treated as constant text by the macro processor, and is placed on the input stack to be processed when the macro finishes executing or at a step boundary, just as if the user had typed the code at that location in the program.

Example 1. SAS code but no macro language elements.

Macro MAC1 contains a DATA step but no macro language elements. While not a very interesting macro application in real life, this macro illustrates at the simplest level the relationship between SAS code and macros.

```
%macro mac1;                       (1)
  data one;                        (2)
    x1=11;                         (3)
    x2=22;                         (4)
    x3=33;                         (5)
  run;                             (6)
%mend mac1;                        (7)
```

```
  %mac1                              (8)
```

(1) - (7) define macro MAC1. After the macro processor processes (7), MAC1 is compiled. (8) invokes MAC1, which executes as follows.

- (2) - (6) are SAS code and are generated, just as if the user had typed them at that location in the program. The following SAS code is generated.

  ```
  data one;
    x1=11;
    x2=22;
    x3=33;
  run;
  ```

- (6) is SAS code that specifies a step boundary. It causes DATA step ONE to be compiled and executed.
- (7) is a %MEND statement that ends macro execution.

A macro of this type could be useful if the same DATA step was needed in more than one location in an application. You could just invoke the macro with %MAC1 wherever the DATA step was needed.

Example 2. SAS code and macro language elements with a step boundary.

Macro MAC2 contains a DATA step and some %LET and %PUT statements, which are macro language elements. As noted above, any SAS code generated by a macro is processed (compiled and executed) when macro processing completes or at any step boundary. This example illustrates the effect of step boundaries.

```
  %macro mac2;                                           (1)
    %local blah;                                         (2)
    data one;                                            (3)
      x=1;                                               (4)
      %let blah=11;                                      (5)
      %put in DATA step before PUT statement blah=&blah; (6)
      put x=;                                            (7)
      %let blah=22;                                      (8)
      %put in DATA step after PUT statement blah=&blah;  (9)
    run;                                                 (10)

    %put after DATA step;                                (11)

  %mend mac2;                                            (12)
  %mac2                                                  (13)
```

(1) - (12) define macro MAC2. After the macro processor processes (12), MAC2 is compiled. (13) invokes MAC2, which executes as follows. Text written to the SAS log when MAC2 executes is included at the end of this section.

- (2) is a macro language element and is immediately executed. Macro variable BLAH is created and is local to this macro.
- (3) and (4) are SAS code and are generated.
- (5) is a macro language element and is immediately executed. Macro variable BLAH is assigned the value 11.
- (6) is a macro language element and is immediately executed. The text in this statement is written to the SAS log.
- (7) is SAS code and is generated.
- (8) is a macro language element and is immediately executed. Macro variable BLAH is assigned the value 22.
- (9) is a macro language element and is immediately executed. The text in this statement is written to the SAS log.
- (10) is SAS code and is generated. Since it specifies a step boundary, DATA step ONE comprising SAS statements (3), (4), (7), and (10) is compiled and executed. The PUT statement, (7), writes x=1 to the SAS log. Note in the SAS

log shown below that (6) and (9) are written to the SAS log before (7).

- (11) is a macro language element and is immediately executed. The text in this statement is written to the SAS log.
- (12) is a %MEND statement that ends macro execution.

The following is written to the SAS log when MAC2 executes.

```
in DATA step before PUT statement blah=11                    <==== from (6)
in DATA step after PUT statement blah=22                     <==== from (9)

x=1                                                          <==== from (7)
NOTE: The data set WORK.ONE has 1 observations and 1 variables.
NOTE: DATA statement used (Total process time):
      real time           0.00 seconds
      cpu time            0.01 seconds

after DATA step                                              <==== from (11)
```

Example 3. Use macro logic to conditionally generate a SAS statement.

In this example, we want a DATA step to include the statement X2=22; only when the value of macro variable INCLUDEX2 is 1.

DATA step when INCLUDEX2 is 1:

```
data one;
  x1=11;
  x2=22;
  x3=33;
run;
```

DATA step when INCLUDEX2 is not 1:

```
data one;
  x1=11;
  x3=33;
run;
```

Let's review the execution of macro MAC3 to see how it differs when INCLUDEX2 is or is not equal to 1. In a real application, INCLUDEX2's value could be generated within the application, but for simplicity, we just assign its value with a %LET statement.

Example 3.a. When INCLUDEX2 is equal to 1.

```
%let includex2=1;

%macro mac3(macvar=);              (1)
  data one;                        (2)
    x1=11;                         (3)
    %if &macvar=1 %then %do;       (4)
      x2=22;                       (5)
    %end;                          (6)
    x3=33;                         (7)
  run;                             (8)
%mend mac3;                        (9)

%mac3(macvar=&includex2)           (10)
```

(10) invokes macro MAC3. Macro variable MACVAR, which is local to this macro, has the value 1. MAC3 executes as follows.

- (2) and (3) are SAS code and are generated.

- In (4), the macro processor replaces the macro variable MACVAR with its value, 1.

  ```
  %if &macvar=1 %then %do;          (original statement)
  %if 1=1 %then %do;                (after macro variable replaced with its value)
  ```

  1=1 is true, so the macro processor processes the code between %DO and %END. (5) is SAS code and is generated.

- (7) and (8) are SAS code and are generated. (8) is a step boundary, so the following SAS statements are compiled and executed.

  ```
  data one;
    x1=11;
    x2=22;
    x3=33;
  run;
  ```

- (9) is a %MEND statement that ends macro execution.

Example 3.b. When INCLUDEX2 is not equal to 1.

```
%let includex2=-999;

%macro mac3(macvar=);                (1)
  data one;                          (2)
    x1=11;                           (3)
    %if &macvar=1 %then %do;         (4)
      x2=22;                         (5)
    %end;                            (6)
    x3=33;                           (7)
  run;                               (8)
%mend mac3;                          (9)

%mac3(macvar=&includex2)             (10)
```

Now, when MAC3 executes, macro variable MACVAR has the value -999. MAC3, repeated from above for convenience, executes as follows.

- (2) and (3) are SAS code and are generated.

- In (4), the macro processor replaces the macro variable MACVAR with its value, -999.

  ```
  %if &macvar=1 %then %do;          (original statement)
  %if -999=1 %then %do;             (after macro variable replaced with its value)
  ```

  -999=1 is false, so the macro processor does not process the code between %DO and %END. (5) is not generated.

- (7) and (8) are SAS code and are generated. (8) is a step boundary, so the following SAS statements are compiled and executed.

  ```
  data one;
    x1=11;
    x3=33;
  run;
  ```

- (9) is a %MEND statement that ends macro execution.

Example 4. Use a macro loop to generate clauses in a SAS statement.

Data set ONE contains variables X1, X2, and X3, to be renamed to Y1, Y2, and Y3.

```
data one;
```

```
    x1=11;
    x2=22;
    x3=33;
  run;
```

A DATA step to create data set TWO and rename the variables could be coded as follows.

```
  data two;
    set one;
    rename
      x1=y1
      x2=y2
      x3=y3
    ;
  run;
```

The same DATA step code can be generated by a macro. This is more generalized, though in a real-world application, the upper bound of the %DO loop would probably be determined dynamically in the program, not hard-coded as 3.

```
  %macro mac4;                                (1)
    %local j; /* local macro variable */      (2)
    data two;                                 (3)
      set one;                                (4)
      rename                                  (5)
        %do j = 1 %to 3;                      (6)
          x&j = y&j                           (7)
        %end;                                 (8)
      ;                                       (9)
    run;                                      (10)
  %mend mac4;                                 (11)
  %mac4;                                      (12)
```

The RENAME statement generated by the macro has three components.

- "RENAME" precedes the macro loop. It is generated by (5).
- Each iteration of the macro loop, (6) - (8), generates one clause of the RENAME statement. Since each clause is part of a SAS statement, a semicolon must not be included as part of (7).

  ```
  The first time, it generates:   x1 = y1
  The second time, it generates:  x2 = y2
  The third time, it generates:   x3 = y3
  ```

- A semicolon follows the macro loop. It is generated by (9).

Macro MAC4 executes as follows.

- (3) - (5) are SAS code and are generated.
- (6) - (8) are executed by the macro processor as follows.
  - (6) J is replaced with its value, 1. 1 is within the loop bounds, 1 to 3, so %DO loop processing continues.
  - (7) The following SAS code is generated: x1= y1
  - (8) %END ends the %DO statement, and the macro returns to (6).

  - (6) J is incremented from 1 to 2. 2 is within the loop bounds, 1 to 3, so %DO loop processing continues.
  - (7) The following SAS code is generated: x2= y2
  - (8) %END ends the %DO statement, and the macro returns to (6).

  - (6) J is incremented from 2 to 3. 3 is within the loop bounds, 1 to 3, so %DO loop processing continues.
  - (7) The following SAS code is generated: x3= y3
  - (8) %END ends the %DO statement, and the macro returns to (6).

  - (6) J is incremented from 3 to 4. 4 is outside the loop bounds, 1 to 3, so %DO loop processing ends, and processing resumes with (9).

- (9) and (10) are SAS code and are generated. (9) is a semicolon that completes the RENAME statement. (10), a RUN statement, is a step boundary that ends the DATA step and causes the following SAS statements to be compiled and executed.

```
 data two;
   set one;
   rename
   x1 = y1
   x2 = y2
   x3 = y3
   ;
 run;
```

- (11) is a %MEND statement that ends macro execution.

This code shows that it is simple to iteratively generate part of a SAS statement in a macro once you understand the interaction between SAS code (DATA and PROC steps) and macros. This is an important technique with many possible uses, though this particular program is more easily coded with the following RENAME statement.

```
rename x1-x3 = y1-y3;
```

Example 5. Use a macro loop to generate clauses in a SAS statement: an alternate approach.

An equivalent way to code Example 4 is to include just the macro loop code in the macro, as follows.

```
%macro mac5;                      (1)
  %local j;                       (2)
  %do j = 1 %to 3;                (3)
    x&j = y&j                     (4)
  %end;                           (5)
%mend mac5;                       (6)

data two;                         (7)
  set one;                        (8)
  rename                          (9)
    %mac5                         (10)
  ;                               (11)
  run;                            (12)
```

This code executes as follows.

- (1) - (6) define macro MAC5. (6) ends the macro definition, and causes the macro to be compiled.

- (7) - (9) are SAS code and are copied to the input stack for processing at a step boundary.

- (10) invokes macro MAC5. The macro processor executes (3) - (5), inserting the resulting code as if it had been typed between (9) and (11). Execution is similar to (6) - (8) in Example 4, as follows.

  - (2) defines a local macro variable.

  - (3) J is replaced with its value, 1. 1 is within the loop bounds, 1 to 3, so %DO loop processing continues.
  - (4) The following SAS code is generated: x1= y1
  - (5) %END ends the %DO statement, and the macro returns to (3).

  - (3) J is incremented from 1 to 2. 2 is within the loop bounds, 1 to 3, so %DO loop processing continues.
  - (4) The following SAS code is generated: x2= y2
  - (5) %END ends the %DO statement, and the macro returns to (3).

  - (3) J is incremented from 2 to 3. 3 is within the loop bounds, 1 to 3, so %DO loop processing continues.
  - (4) The following SAS code is generated: x3= y3
  - (5) %END ends the %DO statement, and the macro returns to (3).

  - (3) J is incremented from 3 to 4. 4 is outside the loop bounds, 1 to 3, so %DO loop processing ends, and processing resumes with (6).

- (6) is a %MEND statement that ends the macro. At this point, the following SAS code has been placed on the input stack but not compiled or executed.

```
data two;
  set one;
  rename
  x1 = y1
  x2 = y2
  x3 = y3
```

- (11) and (12) are SAS code and are copied to the input stack. (11) is a semicolon that completes the RENAME statement. (12), a RUN statement, is a step boundary that ends the DATA step and causes the following SAS statements to be compiled and executed.

```
data two;
  set one;
  rename
  x1 = y1
  x2 = y2
  x3 = y3
  ;
run;
```

The SAS statements generated and then compiled and executed in this example are the same as in Example 4.

Example 6. Use “Data driven” code to rename variables.

This example is more complex, and illustrates a more automated application.

We want to rename all variables in a data set that start with the letter “X” so that they are prefaced with “OLD_”. For example, rename “X” to “OLD_X” and “XYZ” to “OLD_XYZ”. We don’t know the number of variables that start with “X” or their names.

First, let’s construct a data set to work with. Three variables meet the selection criterion (start with “X”), and one does not.

```
data one;
  x1=11;
  y=22;
  xxx=33;
  xyz=44;
run;
```

Now, read the variable names that begin with "X" from a DICTIONARY table and copy them to macro variables. DICTIONARY tables are a set of read-only SAS data views that contain information about the current SAS session such as data libraries, SAS data sets, SAS macros, and external files in use or available and SAS system options settings. See the *DICTIONARY Tables* chapter in *SAS 9.1.3 Language Reference: Concepts* for an introduction to DICTIONARY tables, and *The SQL Procedure* chapter in the *Base SAS 9.1.3 Procedures Guide* for complete information. For a more detailed explanation of similar examples, see Gilsen (2008).

```
proc sql noprint;                    (1)
  select name
  into :var1-:var&sysmaxlong
    from dictionary.columns
    where libname="WORK"
    and memname="ONE"
    and upcase(substr(name,1,1))="X";
  %let num_vars=&sqlobs;
quit;
```

This code executes as follows.

- The first variable name that meets the selection criteria (starts with upper case “X” in data set WORK.ONE) is copied to macro variable VAR1, the second variable name that meets the selection criteria is copied to macro variable VAR2, and so on.

- SYSMAXLONG is an operating system-specific automatic macro variable containing the largest possible integer value. Specifying SYSMAXLONG as the upper bound of the range of macro variables allows the number of macro variables created by PROC SQL to be data driven. In this example, three variables meet the selection criteria, so macro variables VAR1, VAR2, and VAR3 are created with the following values.

Macro variable	Value
VAR1	x1
VAR2	xxx
VAR3	xyz

- The automatic macro variable SQLOBS contains the number of rows selected by the last PROC SQL statement. In this example, the value of SQLOBS is 3.

Now, execute PROC DATASETS and use a RENAME statement to rename the variables. For each variable name, macro MAC6 generates the text *varname*=OLD_*varname,* which is used as a clause in the RENAME statement.

```
%macro mac6;                     (2)
  %local j;                      (3)
  %do j = 1 %to &num_vars;       (4)
    &&var&j = old_&&var&j        (5)
  %end;                          (6)
%mend mac6;                      (7)

proc datasets library=work nolist;   (8)
  modify one;                    (9)
  rename                         (10)
    %mac6                        (11)
  ;                              (12)
run; quit;                       (13)
```

Before explaining this code, let's briefly focus on the double ampersand (&&) in (5). The double ampersand (&&) is called an "indirect reference" to a macro variable. The macro processor resolves two ampersands to one ampersand (&& to &) rather than applying the & to the text that follows.

The first time the loop executes, macro variable J is 1, and the macro processor processes &&VAR&J left to right in two passes as follows:

```
  Pass 1:
  1. && resolves to &
  2. VAR is text and is unchanged
  3. &J is resolved to 1
  Result: &&VAR&J is resolved to &VAR1

  Pass 2:
  1. &VAR1 is resolved to x1, so &&VAR&J is replaced by x1 in two places, generating
x1=old_x1
```

The second time the loop executes, macro variable J is 2, and the macro processor processes &&VAR&J left to right in two passes as follows:

```
  Pass 1:
  1. && resolves to &
  2. VAR is text and is unchanged
  3. &J is resolved to 2
  Result: &&VAR&J is resolved to &VAR2

  Pass 2:
  1. &VAR2 is resolved to xxx, so &&VAR&J is replaced by xxx in two places, generating
  xxx=old_xxx
```

The && is needed to make the macro processor scan twice. If you code &VAR&J instead of &&VAR&J, the macro processor tries to resolve &VAR and &J in a single pass and concatenate the values. This generates an error unless &VAR was previously defined (in that case, there is no error message but an unexpected result).

Now, let's review this code, which executes similarly to Example 5.

- (2) - (7) define macro MAC6. (7) ends the macro definition, and causes the macro to be compiled.
- (8) - (10) are SAS code and are copied to the input stack.
- (11) invokes macro MAC6. The macro processor executes statements as follows.
  - (3) defines a local macro variable.

  - (4) J is replaced with its value, 1. 1 is within the loop bounds, 1 to 3, so %DO loop processing continues.
  - (5) The following SAS code is generated: x1 = old_x1
  - (6) %END ends the %DO statement, and the macro returns to (6).

  - (4) J is incremented from 1 to 2. 2 is within the loop bounds, 1 to 3, so %DO loop processing continues.
  - (5) The following SAS code is generated: xxx = old_xxx
  - (6) %END ends the %DO statement, and the macro returns to (6).

  - (4) J is incremented from 2 to 3. 3 is within the loop bounds, 1 to 3, so %DO loop processing continues.
  - (5) The following SAS code is generated: xyz = old_xyz
  - (6) %END ends the %DO statement, and the macro returns to (6).

  - (4) J is incremented from 3 to 4. 4 is outside the loop bounds, 1 to 3, so %DO loop processing ends, and macro processing resumes with (7).

  - (7) is a %MEND statement that ends the macro. At this point, the following SAS code has been generated, but not compiled or executed.

```
proc datasets library=work nolist;
  modify one;
  rename
    x1 = old_x1
    xxx = old_xxx
    xyz = old_xyz
```

- (12) and (13) are SAS code and are copied to the input stack. (12) is a semicolon that completes the RENAME statement, and (13) is a step boundary that ends PROC DATASETS and causes the following SAS statements to be compiled and executed.

```
proc datasets library=work nolist;
  modify one;
  rename
    x1 = old_x1
    xxx = old_xxx
    xyz = old_xyz
  ;
run; quit;
```

Note that a more robust version of this example would include the following tests.

- That no existing variable name is longer than 28 characters, since OLD_ adds 4 characters to the name and the maximum variable name length is 32 characters.
- That there are no cases where variable names OLD_ X*name* and X*name* both exist. That is, if the data set has variables XYZ and OLD_XYZ, the RENAME statement generates an error when it tries to rename XYZ to OLD_XYZ.

An alternate way to code this example is to copy the variables that meet the selection criterion to a single space-separated macro variable, use the automatic macro variable SQLOBS to determine the number of variable names in data set WORK.ONE that begin with "X",, and use the %SCAN function to process the variable names, as follows. The PROC DATASETS step and the results are the same as above.

```
proc sql noprint;
  select name into :var_names separated by ' '
  from dictionary.columns
```

```
    where libname="WORK"
    and memname="ONE"
    and upcase (substr(name,1,1))="X";
    %let num_values=&sqlobs;
  quit;

  %macro mac6;
    %local j;
    %do j = 1 %to &num_values;
      %scan(&var_names, &j) = old_%scan(&var_names, &j)
    %end;
  %mend mac6;

  proc datasets library=work nolist;
    modify one;
    rename
      %mac6
    ;
  run; quit;
```

Example 7. Create a macro variable and use it in the same DATA step.

This example shows that you cannot assign a DATA step value to a macro variable with CALL SYMPUT and use a macro variable reference (preceding a macro variable's name with an ampersand) to retrieve the macro variable's value in the same step. Instead, you must use the RESOLVE or SYMGET function.

```
  %let macvar=oldvalue ;                              (1)

  data _null_ ;                                       (2)
     call symput("macvar","newvalue");                (3)
     var1 = "&macvar";                                (4)
     put 'from macro variable reference: ' var1= ;    (5)
  run;                                                (6)
```

This code executes as follows.

- (1) assigns the macro variable MACVAR the value oldvalue. For the purpose of this example, MACVAR must get the value oldvalue prior to the DATA step. That could be in open code as shown here (not in a DATA or PROC step) or in a prior DATA or PROC step.
- (2) - (6) comprise a DATA step. (6) is a step boundary, and it causes the DATA step to be compiled and then executed, as detailed below.

The key to understanding how this DATA step compiles and executes is that CALL SYMPUT assigns the value of a macro variable during step execution, but macro variable references resolve at one of the following times.

- During step compilation.
- In a global statement used outside a step.
- In a SAS Component Language (SCL) program.

The DATA step compiles and executes as follows.

- During DATA step compilation, the macro variable reference in (4) is resolved. MACVAR is replaced with its value, oldvalue, and the following code is compiled.

```
data _null_ ;                                       (2)
  call symput("macvar","newvalue");                 (3)
  var1 = "oldvalue";                                (4)
  put 'from macro variable reference: ' var1= ;     (5)
run;                                                (6)
```

- DATA step execution begins with (2).

- In (3), CALL SYMPUT assigns a DATA step value to a macro variable. CALL SYMPUT takes two arguments:
  - The first argument, “macvar”, is the name of the macro variable.
  - The second argument, “newvalue”, is the value of the macro variable.
- (4) assigns the value “oldvalue” to VAR1. Remember that the macro variable MACVAR was resolved to oldvalue during DATA step compilation.
- (5) writes the following text to the SAS log.

  from macro variable reference: var1=oldvalue
- (6) ends execution of the DATA step.

It is likely that the intent of this code (or similar code) was to assign the value newvalue to VAR1. But, as shown, if you assign a value to a macro variable with CALL SYMPUT, you cannot retrieve the value with a macro variable reference until after the DATA step finishes.

To retrieve the value of the macro variable in the same step, use the RESOLVE or SYMGET function instead of a macro variable reference. RESOLVE and SYMGET are similar, but RESOLVE accepts a wider variety of arguments, such as macro expressions.

In the following code, RESOLVE and SYMGET both correctly retrieve the value of the macro variable in the same step as the CALL SYMPUT. VAR2 and VAR3 have the value "newvalue", as desired. Both functions are included in this example for illustrative purposes, but only one is needed.

```
%let macvar=oldvalue ;                                   (1)

data _null_ ;                                            (2)
   call symput("macvar","newvalue");                     (3)
   var2 = resolve('&macvar') ;                           (4)
   put 'from resolve function: ' var2= ;                 (5)
   var3 = symget('macvar') ;                             (6)
   put 'from symget function: ' var3= ;                  (7)
run;                                                     (8)
```

This code executes as follows.

- (1) assigns the macro variable MACVAR the value oldvalue. As in the previous code, MACVAR has the value oldvalue prior to the DATA step.
- (2) - (8) comprise a DATA step. (8) is a step boundary, and it causes the DATA step to be compiled and then executed as follows.
- DATA step execution begins with (2).
- In (3), CALL SYMPUT assigns a DATA step value, newvalue, to macro variable MACVAR.
- (4) assigns the value of macro variable MACVAR, newvalue, to VAR2. Unlike the macro variable reference (&MACVAR) in the previous code, MACVAR is resolved during DATA step execution, not DATA step compilation.
- (5) writes the following text to the SAS log.

  from resolve function: var2=newvalue
- (6) assigns the value of macro variable MACVAR, newvalue, to VAR3. Unlike the macro variable reference (&MACVAR) in the previous code, MACVAR is resolved during DATA step execution, not DATA step compilation.
- (7) writes the following text to the SAS log.

  from symget function: var3=newvalue
- (8) ends execution of the DATA step.

Example 8. Use CALL EXECUTE to submit SAS code or macro code in a DATA step.

The CALL EXECUTE routine allows you to generate code within a DATA step. This can be very useful for tasks such as calling a macro, DATA step, or procedure for every observation in a data set or for selected observations in a data set.

The syntax of CALL EXECUTE is: CALL EXECUTE(*argument*);. For complete details, see the *SAS 9.1.3 Language Reference: Dictionary*.

CALL EXECUTE must be used with caution to avoid timing errors, because the argument to CALL EXECUTE is passed to the macro processor and processed as follows.

- Macros and macro language elements are resolved or execute immediately.
- SAS language statements (whether generated by CALL EXECUTE or by macro language elements generated by CALL EXECUTE) are processed at the next step boundary (after the current DATA step finishes executing).

This section presents a few simple examples of CALL EXECUTE and shows an example of an unintended result caused by timing issues.

Example 8.1. Call a macro for selected observations of a data set, based on the values of a variable in the data set. Use variable values from the data set as macro parameter values.

Data set ONE has the following values.

```
Obs   company      audited
 1    ibm             1
 2    microsoft       0
 3    aol             1
```

We want to call macro MAC7 for observations in which AUDITED is 1, using values of COMPANY as parameters to MAC7. In this example, MAC7 just does a %PUT statement for illustrative purposes, but in a real application, it would have additional content.

```
%macro mac7(firm=);                                    (1)
  %put in mac7 firm= &firm;                            (2)
%mend mac7;                                            (3)

data two;                                              (4)
  set one;                                             (5)
  where audited = 1;                                   (6)
  call execute ( '%mac7(firm='||company||')' );        (7)
  put "at end of TWO " company=;                       (8)
run;                                                   (9)
```

This code executes as follows.

- (1) - (3) define macro MAC7. (3) ends the macro definition, and causes the macro to be compiled.
- (4) - (9) comprise a DATA step. (9) is a step boundary, and it causes the DATA step to be compiled and then executed as follows.
- For the first observation of data set ONE.
  - AUDITED is 1, so the WHERE statement selection criteria in (6) is met, and the observation is processed.
  - In (7), COMPANY is replaced with its value, ibm, and the following statement is sent to the macro processor.

    ```
    call execute ( '%mac7(firm=ibm)' );
    ```

  - Macro MAC7 begins executing. The macro processor replaces the parameter FIRM with its value, so (2) resolves to the following.

    ```
    %put in mac7 firm= ibm;
    ```

- ● (2) is a macro language element and is executed immediately, writing text to the log.
- ● (3) ends the macro.
- ● (8) writes text to the log, and (9) ends the current iteration of the DATA step.

● For the second observation of data set ONE.

- ● AUDITED is 0, so the WHERE statement selection criteria in (6) is not met, and the observation is not processed.

● For the third observation of data set ONE.

- ● AUDITED is 1, so the WHERE statement selection criteria in (6) is met, and the observation is processed.
- ● In (7), COMPANY is replaced with its value, aol, and the following statement is sent to the macro processor.

```
call execute ( '%mac7(firm=aol)' );
```

● Macro MAC7 begins executing. The macro processor replaces the parameter FIRM with its value, so (2) resolves to the following.

```
%put in mac7 firm= aol;
```

- ● (2) is a macro language element and is executed immediately, writing text to the log.
- ● (3) ends the macro.
- ● (8) writes text to the log, and (9) ends the current (and final) iteration of the DATA step.

Here is part of the SAS log generated by this code.

```
in mac7 firm= ibm                                                     <==== from (2)
at end of TWO company=ibm                                             <==== from (8)
in mac7 firm= aol                                                     <==== from (2)
at end of TWO company=aol                                             <==== from (8)
NOTE: There were 2 observations read from the data set WORK.ONE.
      WHERE audited=1;
NOTE: The data set WORK.TWO has 2 observations and 2 variables.
NOTE: DATA statement used (Total process time):
      real time           0.00 seconds
      cpu time            0.01 seconds

NOTE: CALL EXECUTE routine executed successfully, but no SAS statements were generated.
```

Note that Gilsen (2008) presents alternate ways to accomplish this task using CALL SYMPUT or an SQL SELECT statement to create macro variables.

Example 8.2. Call a macro that executes a procedure once for each observation of a data set. Use variable values from the data set as macro parameters.

Data set ONE has the following values.

```
Obs   company      audited
 1    ibm             1
 2    microsoft       0
 3    aol             1
```

We want to generate a PROC PRINT from each observation. If AUDITED is 1, print the variables INCOME and TAX. Otherwise, print just INCOME. In this example, the PROC PRINT for data sets IBM and AOL should include INCOME and TAX, and the PROC PRINT for data set MICROSOFT should include just INCOME.

```
%macro mac8(firm=, vars=);                                          (1)
  %put in mac8 firm= &firm vars= &vars;                             (2)
  proc print data= &firm;                                           (3)
    title &firm;                                                    (4)
    var &vars;                                                      (5)
  run;                                                              (6)
%mend mac8;                                                         (7)

data two;                                                           (8)
  set one;                                                          (9)
  if audited = 1 then                                               (10)
    call execute ( '%mac8(firm='||company||', vars=income tax)' );  (11)
  else                                                              (12)
    call execute ( '%mac8(firm='||company||', vars=income)' );      (13)
  put "at end of TWO " company=;                                    (14)
run;                                                                (15)
```

This code executes as follows.

- (1) - (7) define macro MAC8. (7) ends the macro definition, and causes the macro to be compiled.
- (8) - (15) comprise a DATA step. (15) is a step boundary, and it causes the DATA step to be compiled and then executed as follows.
- For the first observation.
  - AUDITED is 1. (10) is true, so (11) is executed.
  - In (11), COMPANY is replaced with its value, ibm, and the following statement is sent to the macro processor.

```
call execute ( '%mac8(firm=ibm, vars=income tax)' );
```

  - Macro MAC8 begins executing. The macro processor replaces the parameters FIRM and VARS with their values, producing the following code.

```
%put in mac8 firm= ibm vars= income tax;                            (2)
proc print data= ibm;                                               (3)
  title ibm;                                                        (4)
  var income tax;                                                   (5)
run;                                                                (6)
```

  - (2) is a macro language element and is executed immediately, writing text to the log.
  - (3) - (6) are SAS code and are copied to the input stack for processing at a step boundary (when DATA step TWO finishes executing).
  - (7) ends the macro.
  - (14) writes text to the log, and (15) ends the current iteration of the DATA step.
- For the second observation.
  - AUDITED is 0. (10) is false, so (13) is executed.
  - In (13), COMPANY is replaced with its value, microsoft, and the following statement is sent to the macro processor.

```
call execute ( '%mac8(firm=microsoft, vars=income)' );
```

  - Macro MAC8 begins executing. The macro processor replaces the parameters FIRM and VARS with their values, producing the following code.

```
%put in mac8 firm= microsoft vars= income;                          (2)
proc print data= microsoft;                                         (3)
  title microsoft;                                                  (4)
  var income;                                                       (5)
  run;                                                         (6)
```

- ● (2) is a macro language element and is executed immediately, writing text to the log.
- ● (3) - (6) are SAS code and are copied to the input stack for processing at a step boundary (when DATA step TWO finishes executing).
- ● (7) ends the macro.
- ● (14) writes text to the log, and (15) ends the current iteration of the DATA step.

● For the third observation.

- ● AUDITED is 1. (10) is true, so (11) is executed.
- ● In (11), COMPANY is replaced with its value, aol, and the following statement is sent to the macro processor.

```
  call execute ( '%mac8(firm=aol, vars=income tax)' );
```

- ● Macro MAC8 begins executing. The macro processor replaces the parameters FIRM and VARS with their values, producing the following code.

```
  %put in mac8 firm= aol vars= income tax;                     (2)
  proc print data= aol;                                        (3)
    title aol;                                                 (4)
    var income tax;                                            (5)
  run;                                                         (6)
```

- ● (2) is a macro language element and is executed immediately, writing text to the log.
- ● (3) - (6) are SAS code and are copied to the input stack for processing at a step boundary (when DATA step TWO finishes executing).
- ● (7) ends the macro.
- ● (14) writes text to the log, and (15) ends the current iteration of the DATA step.

● DATA step TWO ends, which is a step boundary. The following code that was copied to the input stack is now processed and executed.

```
  proc print data= ibm;
    title ibm;
    var income tax;
  run;
  proc print data= microsoft;
    title microsoft;
    var income;
  run;
  proc print data= aol;
    title aol;
    var income tax;
  run;
```

Here is part of the SAS log generated by this code.

```
  in mac8 firm= ibm vars= income tax                        <==== from (2)
  at end of TWO company=ibm                                 <==== from (14)
  in mac8 firm= microsoft vars= income                      <==== from (2)
  at end of TWO company=microsoft                           <==== from (14)
  in mac8 firm= aol vars= income tax                        <==== from (2)
  at end of TWO company=aol                                 <==== from (14)
  NOTE: There were 3 observations read from the data set WORK.ONE.  <==== Step boundary
  NOTE: The data set WORK.TWO has 3 observations and 2 variables.
  NOTE: DATA statement used (Total process time):
        real time           0.00 seconds
```

```
      cpu time            0.01 seconds
                                                    <==== Generated SAS code executes now
                                                    <==== since DATA set TWO has finished,
                                                    <==== which is a step boundary

NOTE: CALL EXECUTE generated line.
1         + proc print data= ibm;     title ibm;     var income tax;   run;

NOTE: There were 2 observations read from the data set WORK.IBM.
NOTE: The PROCEDURE PRINT printed page 2.
NOTE: PROCEDURE PRINT used (Total process time):
      real time           0.00 seconds
      cpu time            0.00 seconds

2         + proc print data= microsoft;     title microsoft;     var income;   run;

NOTE: There were 2 observations read from the data set WORK.MICROSOFT.
NOTE: The PROCEDURE PRINT printed page 3.
NOTE: PROCEDURE PRINT used (Total process time):
      real time           0.00 seconds
      cpu time            0.00 seconds

3         + proc print data= aol;     title aol;     var income tax;   run;

NOTE: There were 2 observations read from the data set WORK.AOL.
NOTE: The PROCEDURE PRINT printed page 4.
NOTE: PROCEDURE PRINT used (Total process time):
      real time           0.00 seconds
      cpu time            0.00 seconds
```

Example 8.3. Example of a CALL EXECUTE timing issue.

Data set ALLSALES has the following values.

```
Obs   state1  sales   sellerid
 1    NY       100       1
 2    NY       200       2
 3    NY       300       3
 4    CA       400       4
 5    CA       500       5
 6    CA        50       6
```

We want to call a macro that checks if the maximum sales value for a state is greater than 200, and if so, prints the sales for each seller in that state and the maximum sales value.

First, let’s review code in which the macro is invoked manually. The maximum value of SALES for the specified state is determined with PROC SUMMARY and copied to a macro variable with the CALL SYMPUTX function. If the maximum value of SALES is greater than 200, code to execute PROC PRINT is generated and executed.

```
%macro mac9 ( state= );
  proc summary data = all_state_sales;
    var sales;
    where state1="&state";
    output out = summary1 max=maxsale1;  * Determine maximum value of SALES for the state;
  run;

  data _null_;
    set summary1;
    call symputx ( 'maxsale' , maxsale1 );   * Copy maximum SALES value to macro variable;
  run;

  %if &maxsale > 200 %then
  %do;                                       * If maximum SALES > 200 print report;
    title "Maximum Sale in &state is &maxsale";
    proc print data = all_state_sales;
      where state1="&state";
    run;
  %end;
%mend mac9;

  * Call system manually;
%mac9(state1=CA)
%mac9(state1=NY)
```

A more generalized approach is to read the states from a control file and use CALL EXECUTE to call macro MAC9 for each value of state. In the following code, the control file is self-contained and read with a DATALINES statement for simplicity. In a real application, the control file would more likely be in an external file and read with an INFILE statement and could be generated by another application or user.

```
%macro mac9 ( state= );                                    (1)
  proc summary data = all_state_sales;                     (2)
    var sales;                                             (3)
    where state1="&state";                                 (4)
    output out = summary1 max=maxsale1;                    (5)
  run;                                                     (6)

  data _null_;                                             (7)
    set summary1;                                          (8)
    call symputx ( 'maxsale' , maxsale1 );                 (9)
  run;                                                     (10)

  %if &maxsale > 200 %then %do;                            (11)
    title "Maximum Sale in &state is &maxsale";            (12)
    proc print data = all_state_sales;                     (13)
      where state1="&state";                               (14)
    run;                                                   (15)
  %end;                                                    (16)
%mend mac9;                                                (17)

data _null_;                                               (18)
  input state1 $2.;                                        (19)
  call execute ( '%mac9(state=' || state1 ||')' );         (20)
  datalines;                                               (21)
CA
NY
;run;                                                      (22)
```

This code executes as follows.

- (1) - (17) define macro MAC9. (17) ends the macro definition, and causes the macro to be compiled.
- (18) - (22) comprise a DATA step. (22) is a step boundary, and it causes the DATA step to be compiled and then executed as follows.
- Processing begins for CA, the first value of STATE1 in the control file.
- In (20), STATE1 is replaced with its value, CA, and the following statement is sent to the macro processor.

  ```
  call execute ( '%mac9(state=CA)' );
  ```

- Macro MAC9 begins executing. The macro processor replaces the parameter STATE with its value in (4), (12), and (14), which resolve to the following.

  ```
  where state1="CA";                                 (4), (14)
  title "Maximum Sale in CA is &maxsale";            (12)
  ```

- (2) - (10) are SAS code and are copied to the input stack for processing at a step boundary (when the final DATA step, (18) - (22), finishes executing).
- (11) is a macro language element and is executed immediately. But, macro variable MAXSALE has not yet been created by CALL SYMPUTX. An error similar to the following is generated.

```
WARNING: Apparent symbolic reference MAXSALE not resolved.
ERROR: A character operand was found in the %EVAL function or %IF condition
       where a numeric operand is required. The condition was: &maxsale > 200
ERROR: The macro MAC9 will stop executing.
```

This example demonstrates the timing issue associated with CALL EXECUTE. It assumes that the macro variable MAXSALE was not defined earlier in the SAS session. If it was, the earlier (and most likely incorrect) value of MAXSALE would have determined if the PROC PRINT step would execute. The program would not work as intended, but no error message would be generated.

One way to fix this code is to change the CALL EXECUTE statement to mask macro invocation during compilation with the %NRSTR function, as in the following code. Only (20) has changed from the previous code.

```
%macro mac9 ( state= );                                     (1)
  proc summary data = all_state_sales;                      (2)
    var sales;                                              (3)
    where state1="&state";                                  (4)
    output out = summary1 max=maxsale1;                     (5)
  run;                                                      (6)

  data _null_;                                              (7)
    set summary1;                                           (8)
    call symputx ( 'maxsale' , maxsale1 );                  (9)
  run;                                                      (10)

  %if &maxsale > 200 %then %do;                             (11)
    title "Maximum Sale in &state is &maxsale";             (12)
    proc print data = all_state_sales;                      (13)
      where state1="&state";                                (14)
    run;                                                    (15)
  %end;                                                     (16)
%mend mac9;                                                 (17)

data _null_;                                                (18)
  input state1 $2.;                                         (19)
  call execute ( '%nrstr(%mac9(state=' || state1 ||'))' ); (20)
  datalines;                                                (21)
CA
NY
;run;                                                       (22)
```

This code executes as follows.

- (1) - (17) define macro MAC9. (17) ends the macro definition, and causes the macro to be compiled.
- (18) - (22) comprise a DATA step. (22) is a step boundary, and it causes the DATA step to be compiled and then executed as follows.
- Processing begins for CA, the first value of STATE1 in the control file.
- In (20), STATE1 is replaced with its value, CA, and the following statement is sent to the macro processor.

  ```
  call execute ( '%nrstr(%mac9(state=CA))' );
  ```

- The NRSTR function masks invocation of macro MAC9 during macro compilation. That is, it causes the % to be treated as ordinary text. The following is copied to the input stack for processing at a step boundary.

  ```
  %mac9(state=CA)
  ```

- The same processing takes place for NY, the second value of STATE1 in the control file. In (20), STATE1 is replaced with its value, NY, and the following statement is sent to the macro processor.

  ```
  call execute ( '%nrstr(%mac9(state=NY))' );
  ```

- The NRSTR function again masks invocation of macro MAC9 during macro compilation, and the following is copied to the input stack for processing at a step boundary.

```
%mac9(state=NY)
```

- (22), a RUN statement, is a step boundary that ends the DATA step and causes the following statements on the input stack to be compiled and executed.

```
%mac9(state=CA)
%mac9(state=NY)
```

- Macro MAC9 is executed twice, similarly to how it would be executed if it was invoked manually. Execution for the first invocation of MAC9 is as follows.

- Macro MAC9 begins executing. The macro processor replaces the parameter STATE with its value in (4), (12), and (14), which resolve to the following.

```
where state1="CA";                              (4), (14)
title "Maximum Sale in CA is &maxsale";         (12)
```

- (2) - (6) comprise a PROC step. (6) is a step boundary, and it causes the PROC step to be compiled and then executed. PROC SUMMARY creates data set SUMMARY1 with one observation containing a value of 500 for MAXSALE1.

- (7) - (10) comprise a DATA step. (10) is a step boundary, and it causes the DATA step to be compiled and then executed. CALL SYMPUTX copies the value of MAXSALE1, 500, to macro variable MAXSALE.

- In (11), the macro processor replaces the macro variable MAXSALE with its value, 500.

  %if &maxsale > 200 %then %do; (original statement)
  %if 500 > 200 %then %do; (after macro variable replaced with its value)

  500 >200 is true, so the macro processor processes the code between %DO and %END. (12) - (15) are SAS code and generated. (15) is a step boundary, so the following SAS statements are compiled and executed.

```
title "Maximum Sale in CA is 500";
proc print data = all_state_sales;
  where state1="CA";
run;
```

- Execution for the second invocation of MAC9 is similar. The details are omitted for space reasons.

In this example, the CALL EXECUTE timing issue happened because the DATA step comprised of (7) - (10) needed to execute before the generation of code was finished, so that the value of macro variable MAXSALE would be available in (11). As shown in this example, when execution of a step is required before the generation of code is finished, wrapping %NRSTR around the macro invocation causes the macro invocation (rather than any SAS statements in the macro) to be written to the input stack, resolving the timing issue.

For more information about CALL EXECUTE, including more examples of timing issues, see the *Interfaces with the Macro Facility* chapter in the *SAS 9.1.3 Macro Language Reference* and the *Interfacing with Data* chapter in Carpenter (2004).

CONCLUSION

This paper explained in a simple, informal way the interaction between SAS DATA and PROC step code and macros. Understanding this interaction allows SAS users to understand existing applications and code their own applications more effectively.

For more information, contact the author, Bruce Gilsen, by mail at Federal Reserve Board, Mail Stop 157, Washington, DC 20551; by e-mail at bruce.gilsen@frb.gov; or by phone at 202-452-2494.

REFERENCES

Carpenter, Art. 2004, “*Carpenter’s Complete Guide to the SAS Macro Language, Second Edition*,” Cary, NC: SAS Institute Inc.

Gilsen, Bruce (2007), "More Tales from the Help Desk: Solutions for Common SAS Mistakes," *Proceedings of the SAS Global Forum 2009 Conference.* <http://www2.sas.com/proceedings/forum2007/211-2007.pdf>

Gilsen, Bruce (2009), "Tales from the Help Desk 3: More Solutions for Common SAS Mistakes," *Proceedings of the SAS Global Forum 2009 Conference.* <http://support.sas.com/resources/papers/proceedings09/137-2009.pdf>

Gilsen, Bruce (2008), "Using Data Set Values and Variable Names Outside of the DATA Step," *Proceedings of the SAS Global Forum 2008 Conference*. <http://www2.sas.com/proceedings/forum2008/177-2008.pdf>

SAS Institute Inc. (2004), "*Base SAS 9.1.3 Procedures Guide*," Cary, NC: SAS Institute Inc.

SAS Institute Inc. (2004), "*SAS 9.1.3 Macro Language: Reference*," Cary, NC: SAS Institute Inc.

SAS Institute Inc. (2004), "*SAS 9.1.3 Language Reference: Concepts*," Cary, NC: SAS Institute Inc.

SAS Institute Inc. (2004), "*SAS 9.1.3 Language Reference: Dictionary, Volumes 1, 2, and 3*," Cary, NC: SAS Institute Inc.

ACKNOWLEDGMENTS

The following people contributed extensively to the development of this paper, and their support is greatly appreciated: Donna Hill, Bill Jackson, and Heidi Markovitz at the Federal Reserve Board, and Ian Whitlock. A special thank you goes to Ian Whitlock for engaging in some lengthy email exchanges about macro processing and providing Example 8.3.

TRADEMARK INFORMATION