Paper 211-2007

More Tales from the Help Desk: Solutions for Common SAS ® Mistakes
Bruce Gilsen, Federal Reserve Board

ABSTRACT

In 21 years as a SAS ® consultant at the Federal Reserve Board, I have seen SAS users make the same mistakes year after year. This paper reviews some common mistakes and shows how to fix them. In the context of reviewing these mistakes, the paper provides details about SAS system processing that can help users employ the SAS system more effectively.

1. CREATING A MACRO VARIABLE AND USING IT IN THE SAME DATA STEP.

Before executing a DATA step, the macro variable MACVAR is assigned the value "oldvalue" in either a prior DATA or PROC step or (as in the code below) in open code (not in a DATA or PROC step).

In the DATA step, the macro variable MACVAR is set to "newvalue" with CALL SYMPUT. Then, using a macro variable reference (preceding a macro variable's name with an ampersand), the DATA step variable DATASTEP_VAR1 is set to the value of MACVAR. The value of DATASTEP_VAR1, displayed with a PUT statement, is expected to be "newvalue", but it is "oldvalue".

```
    /* Create or update a macro variable before executing a DATA step */
  %let macvar=oldvalue ;

  data _null_ ;
     /* copy a DATA step variable to a macro variable with CALL SYMPUT */
   call symput("macvar","newvalue");
     /* use macro variable reference, macro variable value is still "oldvar" */
   datastep_var1 = "&macvar";
   put 'from macro variable reference: ' datastep_var1= ;
  run;
```

The PUT statement writes the following text to the SAS log.

```
  from macro variable reference: datastep_var1=oldvalue
```

This problem occurs because you cannot assign a 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 value of the macro variable in the same step.

As explained in the *SAS 9.1.3 Macro Language: Reference*, CALL SYMPUT assigns the value of a macro variable during program 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

Thus, if you assign a value to a macro with CALL SYMPUT, you cannot retrieve the value with a macro variable reference until after the DATA step finishes.

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

In the following DATA step, the RESOLVE function and CALL SYMGET both correctly retrieve the value of the macro variable in the same step as the CALL SYMPUT. DATASTEP_VAR2 and DATASTEP_VAR3 have the value "newvalue", as expected.

```
    /* Create or update a macro variable before executing a DATA step */
  %let macvar=oldvalue ;

  data _null_ ;
```

```
    /* Copy a DATA step variable to a macro variable with CALL SYMPUT */
   call symput("macvar","newvalue");
   datastep_var2 = resolve('&macvar') ;    /* resolve function */
   put 'from resolve function: ' datastep_var2= ;
   datastep_var3 = symget('macvar') ;      /* call symget */
   put 'from symget function: ' datastep_var3= ;
run;
```

The PUT statements write the following text to the SAS log.

```
from resolve function: datastep_var2=newvalue
from symget function: datastep_var3=newvalue
```

2. OVERLAPPING MACRO VARIABLE IN A MAIN MACRO AND A CALLED MACRO.

In the following code, the %DO loop in macro ONE is expected to execute (and call macro TWO) three times, but only executes once.

```
%macro one;
  %do j = 1 %to 3;
    %two;
    /* other macro code here */
  %end;
%mend one;

%macro two;
  %do j = 1 %to 2;
    /* macro TWO loop code here */
  %end;
%mend two;

%one;  /* call macro ONE */
```

To illustrate what happened, add some %PUT statements to the macros and execute macro ONE again.

```
%macro one;
  %do j = 1 %to 3;
    %put PUT 1.1: j= &j;
    %two;
    %put PUT 1.2: j= &j;
    /* other macro code here */
  %end;
  %put PUT 1.3: j= &j;
%mend one;

%macro two;
  %do j = 1 %to 2;
    %put PUT 2.1: j= &j;
    /* macro TWO loop code here */
  %end;
  %put PUT 2.2: j= &j;
%mend two;

%one;  /* call macro ONE */
```

The PUT statements write the following text to the SAS log.

```
PUT 1.1: j= 1
PUT 2.1: j= 1
PUT 2.1: j= 2
PUT 2.2: j= 3
PUT 1.2: j= 3
PUT 1.3: j= 4
```

This problem is an example of a "macro variable scope" error. It occurs because changes to the value of macro variable J in macro TWO effect the value of J in macro ONE. Note that in macro ONE, the value of macro variable J is

2

- 1 before macro TWO is called

- 3 after macro TWO ends

When the %DO loop in macro ONE finishes for the first time, the index variable, J, is increased from 3 to 4 instead of from 1 to 2. Since 4 is outside the loop bounds, the loop does not execute again.

The *SAS 9.1.3 Macro Language: Reference* devotes an entire chapter to the topic of macro variable scope. A detailed discussion is beyond the scope of this paper, but the following programming practice prevents many scope-related problems, including the one in this example.

> If a macro variable is created in a macro, and is not needed after the macro finishes executing (i.e., the macro variable is needed only in the macro or any macros called by the macro), explicitly make it a local macro variable with the %LOCAL statement.

To fix the problem in this example, make J a local macro variable in macro TWO with a %LOCAL statement, as in the following code. Macro ONE is unchanged.

```
%macro two;
  %local j;
  %do j = 1 %to 2;
    %put PUT 2.1: j= &j;
    /* macro TWO loop code here */
  %end;
  %put PUT 2.2: j= &j;
%mend two;
```

Now, the %DO loop in macro ONE executes (and calls macro TWO) three times. The PUT statements write the following text to the SAS log.

```
PUT 1.1: j= 1
PUT 2.1: j= 1
PUT 2.1: j= 2
PUT 2.2: j= 3
PUT 1.2: j= 1
PUT 1.1: j= 2
PUT 2.1: j= 1
PUT 2.1: j= 2
PUT 2.2: j= 3
PUT 1.2: j= 2
PUT 1.1: j= 3
PUT 2.1: j= 1
PUT 2.1: j= 2
PUT 2.2: j= 3
PUT 1.2: j= 3
PUT 1.3: j= 4
```

Macro variable J in macro TWO is local to macro TWO, and does not effect the value of J in macro ONE. In macro ONE, the value of macro variable J is

- 1 before macro TWO is called the first time

- 1 after macro TWO ends the first time

When the %DO loop in macro ONE finishes for the first time, the index variable, J, is increased from 1 to 2. Since 2 is inside the loop bounds, the loop continues to execute (a total of 3 times).

Notes.

1. This problem can occur with any macro variable, but is especially common with index (looping) variables with names like I, II, and J, which users seem to regard as somehow different than other variables.

2. A somewhat analogous DATA step problem is to code a loop with the index variable I, II, or J, without determining if the data set being processed already has a variable with any of those names.

3

3. This problem manifests itself in different ways based on the values of the index variables. For example, without the %LOCAL statement, if the loop in macro ONE was %do j = 1 %to 4;, an infinite loop occurs, because the value of macro variable J in macro ONE is

    1 before macro TWO is called the first time
    3 after macro TWO ends the first time
    4 after the %DO loops ends the first time
    4 before macro TWO is called the second time and beyond
    3 after macro TWO ends the second time and beyond
    4 after the %DO loops ends the second time and beyond

4. If macro variable J in macro ONE is local to macro ONE, it (macro variable J in macro ONE) should also be specified as local, with a %LOCAL statement in macro ONE.

5. Defining local macro variables is especially important if you code a utility macro used by multiple users or applications. Users calling your macro might not (and should not have to) know the names of macro variables that are local to your utility macro. Ensuring that your macro leaves the computing environment unchanged is good programming practice.

3. CONVERTING VARIABLES FROM NUMERIC TO CHARACTER AND CHARACTER TO NUMERIC.

SAS variables are either character or numeric. If you use character variables in a numeric context or vice versa, as users frequently do, the SAS system automatically converts one of the following ways.

- from numeric to character using the BEST12. format

- from character to numeric using the $w informat (w is the length of the character variable)

When automatic conversion takes place, a note similar to one of the following is written to the SAS log. Since the desired result is *usually* (but not always) generated, many users ignore these ubiquitous notes.

```
NOTE: Numeric values have been converted to character
      values at the places given by: (Line):(Column).
      52:8
NOTE: Character values have been converted to numeric
      values at the places given by: (Line):(Column).
      55:9
```

To ensure appropriate results when converting between character and numeric data, use the PUT or INPUT function to control type conversion, as follows.

1. Convert from numeric to character with the PUT function. The syntax is as follows. Use a format large enough to hold the largest possible numeric value being converted.

```
character-variable = put(numeric-variable,format.);
```

The PUT function right-justifies (aligns) the result. Whenever the size (number of digits) of the numeric variable is smaller than the format, the character variable will have leading blanks. Remove the blanks with the COMPRESS function.

```
character-variable = compress(put(numeric-variable,format.));
```

2. Convert from character to numeric with the INPUT function. The syntax is as follows. Make the informat large enough to hold the largest possible character value being converted.

```
numeric-variable = input(character-variable,informat.);
```

The following examples demonstrate automatic type conversions that do not work correctly, and show how to ensure the correct result with the PUT or INPUT function.

Example 1. Assign a very long number to a macro variable with CALL SYMPUT.

In the following DATA step, a long numeric value is assigned to a macro variable. Because all macro variable values are text

(character strings, not numeric values), a numeric value must be converted to a character value when it is assigned to a macro variable. In the CALL SYMPUT statement, the numeric variable NUMVAR1 is automatically converted to character using the BEST12. format. BEST12. is too small to hold the entire value, so NUMVAR1 is truncated during the conversion.

```
data one;
   numvar1 = 1234567.8901234;
   call symput('macvar1',numvar1);
run;
%put macvar1 value: &macvar1;
```

The %PUT statement writes the following text to the SAS log.

```
macvar1 value: 1234567.8901
```

Note that CALL SYMPUTX, which is new in Version 9, generates the same result as CALL SYMPUT in this example, except that a type conversion note is not written to the SAS log.

Fix the problem in this example as follows. Before assigning NUMVAR1 to the macro variable, convert it to character with the PUT function and remove the leading blanks with the COMPRESS function. The maximum size of NUMVAR1 is assumed to be 20 bytes, with a maximum of 7 digits to the right of the decimal point.

```
data one;
   numvar1 = 1234567.8901234;
   call symput('macvar1',compress(put(numvar1,20.7)));
run;
%put macvar1 value: &macvar1;
```

The %PUT statement writes the following text to the SAS log.

```
macvar1 value: 1234567.8901234
```

The nested functions might make COMPRESS(PUT(NUMVAR1,20.7)) look confusing. To understand how this code is processed, evaluate it from the inside parentheses outward, as follows.

a. PUT(NUMVAR1,20.7) generates a 20 byte right-justified character string, "   1234567.8901234". The code now resolves to

```
compress("     1234567.8901234")
```

b. COMPRESS("   1234567.8901234") removes the blanks, resolving to the value that is copied to macro variable MACVAR1.

```
"1234567.8901234"
```

Example 2. Concatenate a numeric value to a character value.

In the following DATA step, the numeric variable NUM1 has the value 1. The user wants to concatenate NUM1 to "2", resulting in "21". Since NUM1 is numeric, it is automatically converted to character with the BEST12. format before concatenation takes place. BEST12. right-justifies (aligns) the result, so 1 is converted to "   1" (11 spaces followed by "1").

```
data one;
   num1=1;
   char1 = "2" || num1 ;
```

The value of CHAR1 is determined by the prior status of CHAR1, as follows.

● CHAR1 does not already exist.

  Character variable CHAR1 is created with length 13, its value is "2   1" (11 spaces between 2 and 1).

● CHAR1 already exists, with a length less than 13.

  CHAR1 has the value "2". The length of a character variable is determined the first time it is used, and cannot be

5

changed by subsequent statements. If, for example, the length of CHAR1 is 8, the concatenated value ("2   1") is truncated after the 8th character, so CHAR1 is "2".

- CHAR1 already exists, with a length of at least 13.

    CHAR1 has the value "2   1" (11 spaces between 2 and 1).

To fix the problem in this example, convert NUM1 to a character value with the PUT function, then concatenate to "2". The expected result, "21", is generated.

```
data one;
  num1=1;
    /* If num1 is always a single digit. */
  char1 = "2" || put(num1,1.) ;
  put char1= ;
    /* If num1 could be 1-4 digits (values from 0-9999).
        Use a different variable name for demonstrative purposes. */
  char2 = "2" || compress(put(num1,4.)) ;
  put char2= ;
run;
```

The PUT statements write the following text to the SAS log.

```
char1=21
char2=21
```

Note that if CHAR1 and CHAR2 do not already exist, they are created with lengths of 2 (CHAR1) and 5 (CHAR2).

Example 3. Extract part of a numeric variable with SUBSTR.

The numeric variable ID contains several concatenated codes. The user wants to extract the code contained in the first four digits. The first argument to SUBSTR must be character, so the SAS System does automatic type conversion of ID from numeric to character with the BEST12. format. The result is as expected for a 12-digit ID value, but not for a 13-digit ID value.

```
data one;
    /* ID has 12 digits, char1_start has the desired value */
  id=111222333444;
  char1_start = substr(id,1,4);      /* 1st 4 characters are a code */
  put "12-digit number: id= " id 14. "  " char1_start=;

    /* ID has 13 digits, char2_start does not have the desired value */
  id=1112223334445;
  char2_start = substr(id,1,4);      /* 1st 4 characters are a code */
  put "13-digit number: id= " id 14. "  " char2_start=;
run;
```

The PUT statements write the following text to the SAS log.

```
12-digit number: id=   111222333444  char1_start=1112
13-digit number: id=  1112223334445  char2_start=1.11
```

To understand this result, consider the following DATA step, in which SUBSTR returns the entire character string. As before, ID is automatically converted to character with the BEST12. format. BEST12. handles a 12-digit ID value as expected, but returns the result in scientific notation for a 13-digit ID value.

```
data one;
  id=111222333444;                /* ID has 12 digits */
  char_all1 = substr(id,1);      /* return all characters */
  put "12-digit number: id= " id 14. "  " char_all1=;

  id=1112223334445;                /* ID has 13 digits */
  char_all2 = substr(id,1);      /* return all characters */
  put "13-digit number: id= " id 14. "  " char_all2=;
run;
```

6

The PUT statements write the following text to the SAS log.

```
12-digit number: id=    111222333444  char_all1=111222333444
13-digit number: id=  1112223334445  char_all2=1.1122233E12
```

To fix the problem in this example, convert ID to character with the PUT function, then extract the first four characters. The maximum size of ID is assumed to be 20 bytes.

```
data one;
     /* ID has 12 digits */
  id=111222333444;
  char1_start = left(put(id,20.));
  char1_start = substr(char1_start,1,4);
  put "12-digit number: id= " id 14. "  " char1_start=;

     /* ID has 13 digits */
  id=1112223334445;
  char2_start = left(put(id,20.));
  char2_start = substr(char2_start,1,4);
  put "13-digit number: id= " id 14. "  " char2_start=;
run;
```

The PUT statements write the following text to the SAS log.

```
12-digit number: id=    111222333444  char1_start=1112
13-digit number: id=  1112223334445  char2_start=1112
```

The two statements that extract a code can be combined into one equivalent but slightly less readable statement as follows.

```
char1_start = substr(left(put(id,20.)),1,4);

char2_start = substr(left(put(id,20.)),1,4);
```

Notes.

1. The STRIP function is an alternative to the COMPRESS function.

2. An alternate way to concatenate character strings is with the functions CAT, CATS, CATT, and CATX, which are new in Version 9. One important caution is that by default, the length of the character variable created by these functions is 200.

4. IMPLICIT RETAIN IN A MERGE STATEMENT.

Data sets ONE and TWO have been sorted by the common variable ID.

```
   Data set ONE              Data set TWO

 Obs    id    var1       Obs    id     var2

  1      .     11         1      5     5555
  2      .     22         2      6     6666
  3      .     33
  4      4     44
  5      5     55
  6      6     66
```

The following code merges data sets ONE and TWO and prints the result.

```
data three;
  merge one two;
  by id;
run;

proc print data=three;
run;
```

7

PROC PRINT for data set THREE generates the following output.

```
Obs    id    var1    var2
 1      .     11       .
 2      .     22       .
 3      .     33       .
 4      4     44       .
 5      5     55     5555
 6      6     66     6666
```

The user decides to do both of the following whenever VAR2 is missing from the merged data set.

- Set VAR2 to the value of VAR1.

- Set the variable MISSING_VAR2 to 1 to flag the values of VAR2 that were missing.

The DATA step that merges ONE and TWO is recoded as follows.

```
data three;
  merge one two;
  by id;
  if var2 eq . then do;
    missing_var2=1;
    var2 = var1;
  end;
run;
```

PROC PRINT for data set THREE now generates the following output. Two unexpected results appear.

- MISSING_VAR2 was expected to be 1 in the first 4 observations, but is missing in observations 2 and 3.

- VAR2 is not set to the value of VAR1 in observations 2 and 3.

```
                            missing_
Obs    id    var1    var2    var2

 1      .     11      11       1
 2      .     22      11       .
 3      .     33      11       .
 4      4     44      44       1
 5      5     55    5555       .
 6      6     66    6666       .
```

This problem results from what might be called the "MERGE statement retain rule." As explained in the *SAS 9.1.3 Language Reference: Concepts* (*Reading, Combining, and Modifying SAS Data Sets* chapter, *Match-Merging* section, Example 2),

"When SAS reads the last observation from a BY group in one data set, SAS retains its values in the program data vector for all variables that are unique to that data set until all observations for that BY group have been read from all data sets."

Let's see how data set THREE is generated, noting that the first value of the BY variable ID, a missing value, occurs in the first 3 observations of data set ONE but not at all in data set TWO.

Generating observation 1 of data set THREE.

- VAR2 is initially set to missing, because data set TWO has no observations in the current BY group.

- The IF statement is true, so MISSING_VAR2 is set to 1, and var2=var1; sets VAR2 to 11.

Generating observations 2 and 3 of data set THREE.

- SAS has already read all observations in data set TWO for the current BY group, so it retains VAR2 (because VAR2 is unique to data set TWO). The value of VAR2 continues to be 11.

- MISSING_VAR2 is not from data set TWO, so it is not retained. At the beginning of every observation, it is set

8

to missing.

● The IF statement is not true, so MISSING_VAR2 is not set to 1.

Generating observation 4 of data set THREE.

● A new value of the BY variable ID, 4, is encountered. This causes VAR2 to be set to missing, because data set TWO has no observations in the current BY group.

● The IF statement is true, so MISSING_VAR2 is set to 1, and var2=var1; sets VAR2 to 44.

Another way to illustrate the problem is to add PUT statements to the DATA step. Note how VAR2 is initially missing in observation 1, is set to 11 in observation 1 by the statement var2 = var1;, and is not reset to missing until observation 4.

```
data three;
  merge one two;
  by id;
  put "after by : " _N_= var2= missing_var2=;
  if var2 eq . then do;
    missing_var2=1;
    var2 = var1;
  end;
  put "after end: " _N_= var2= missing_var2=;
run;
```

The PUT statements write the following text to the SAS log.

```
after by : _N_=1 var2=. missing_var2=.
after end: _N_=1 var2=11 missing_var2=1
after by : _N_=2 var2=11 missing_var2=.
after end: _N_=2 var2=11 missing_var2=.
after by : _N_=3 var2=11 missing_var2=.
after end: _N_=3 var2=11 missing_var2=.
after by : _N_=4 var2=. missing_var2=.
after end: _N_=4 var2=44 missing_var2=1
after by : _N_=5 var2=5555 missing_var2=.
after end: _N_=5 var2=5555 missing_var2=.
after by : _N_=6 var2=6666 missing_var2=.
after end: _N_=6 var2=6666 missing_var2=.
```

There are various ways to fix this problem. One solution is that if VAR2 is missing at the start of an observation, set it back to missing at the end of the observation.

```
data three;
  merge one two;
  by id;
  if var2 eq . then do;
    missing_var2=1;
    var2 = var1;
  end;
  output;
  if missing_var2 = 1 then var2 = .;
run;
```

PROC PRINT for data set THREE now generates the expected output, as follows.

```
                              missing_
Obs    id    var1    var2     var2

 1     .     11       11       1
 2     .     22       22       1
 3     .     33       33       1
 4     4     44       44       1
 5     5     55     5555       .
 6     6     66     6666       .
```

9

CONCLUSION

This paper reviewed and showed how to fix some common mistakes made by new SAS users, and, in the context of discussing these mistakes, provided details about SAS system processing. It is hoped that reading this paper enables users to better understand SAS system processing and thus employ the SAS system more effectively in the future.

For more information, contact

Bruce Gilsen
Federal Reserve Board, Mail Stop 157
Washington, DC 20551
phone: 202-452-2494
e-mail: bruce.gilsen@frb.gov

REFERENCES

Gilsen, Bruce (2003), "Deja-vu All Over Again: Common Mistakes by New SAS Users," *Proceedings of the Sixteenth Annual NorthEast SAS Users Group Conference.* <http://www.nesug.org/html/Proceedings/nesug03/bt/bt010.pdf>

Gilsen, Bruce (2006), "Tales from the Help Desk 3: More Solutions for Simple SAS Mistakes," *Proceedings of the Nineteenth Annual NorthEast SAS Users Group Conference.* <http://216.146.228.136/Proceedings/nesug06/as/as01.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. All rights reserved. Reproduced with permission of SAS Institute Inc., Cary, NC.

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

ACKNOWLEDGMENTS

TRADEMARK INFORMATION