TAKING THE MYSTERY OUT OF SAS® MACRO WHEN USING CALL SYMPUT

K. Larry Landers, Trilogy Consulting Corporation, Inc., San Mateo, CA
Monique Bryher, MRI Consulting, Inc., Los Angeles, CA

Abstract

The SAS manuals and guides provide good information about using CALL SYMPUT to write data values to the macro environment, and, then, to retrieve those values in subsequent DATA steps or PROCs. Even so, it is easy to write seemingly valid code, only to find that the results are not as expected; or to use the same block of code in slightly different contexts, only to find that one works, while the other does not. Much of the confusion in such situations can be attributed to an insufficient understanding of what SAS is doing in the SYMPUT routine. Through a series of examples using CALL SYMPUT statements, we will illustrate how SAS processes macro variables and discuss the considerations for handling character, numeric and null values within or between DATA steps.

Introduction

A brief discussion is in order about how SAS handles all code presented for processing. First, the word scanner examines the code for syntax errors and for the presence of possible macro **tokens**. A word beginning with either an ampersand (&) or a percent sign (%) is considered a macro **token**. The next action depends upon whether a token is found. If no macro token is found, the code simply passes to the Base SAS compiler for processing, but if a macro token is found, the code is sent first to the macro compiler and then to the Base SAS compiler. The macro compiler evaluates the syntax of the macro code, compiles the code, and creates the macro environment so that macro variables are available to the Base SAS compiler.

The nature of the information residing in a macro variable can be virtually anything, since a macro variable exists as a "text string" -- a string which can vary in length from 0 to 32K bytes. Even if the programmer places a numeric value into a macro variable, the macro compiler sees the digits only as bytes of text. Thus, the programmer must take care to manipulate these text strings appropriately for each task. The programmer must, in effect, "take control" when creating a macro variable so that no ambiguity exists as to its content. This, in turn, will enable the information to be retrieved reliably from the macro environment.

Syntax

A macro variable may be created in three ways:

- a %GLOBAL statement (which defines it with a null value): %global *Name* ;
- a %LET statement (which may define and/or assign it a value): %let *Name* = *Value* ;
- the SYMPUT routine, available only as a CALL statement in a DATA step: CALL SYMPUT (*Name* , *Value*) ;

where *Name* is the name of the macro variable to be created; *Value* is the value to be assigned to *Name*.

To retrieve *Value* from the macro environment, you refer to the macro reference *Name*.

Numeric Values

When we speak about numeric values here, we are referring to the values of numeric variables in a SAS data set. When it operates on such variables, CALL SYMPUT uses, by default, the BEST12. format to create *Name* as a right-aligned, 12-byte macro variable containing *Value*. Being familiar with BEST12. as SAS's default format for any numeric variable for which no format has been specified, many programmers assume they understand what SAS is doing as it takes this default action. Unfortunately, they often fail to recognize that a *right-aligned, 12-byte* macro variable created with CALL SYMPUT can be very different from a macro variable created with a %LET statement, which is *left-aligned* and may have a length up to *32K bytes.*

What does a *12-byte, right-aligned* macro variable look like? Figure 1 shows the creation of six macro variables in a DATA step via CALL SYMPUT statements. Look at the code in the left column, and then between the chevrons in the right column (<< >>) to see what has been written to the macro environment. In Line 1, note that CALL SYMPUT has placed nine spaces before the number '123' to create *Value* as a 12-byte, right-aligned field. In Line 2, the PUT function and format create a seven-byte number '123.000', which is then preceded by one space to comply with the specified 8.3 format, and followed by four spaces to complete the 12-byte, right-aligned *Value*. Line 3 creates a macro variable with neither preceding blanks, nor trailing blanks, by using, first, the PUT function to convert the numeric variable *X* to a character string, and, then, applying the LEFT and TRIM functions to the string before placing *Value* in the macro environment. The reasons for this will be discussed below. In a manner analogous to Lines 1 through 3, Lines 4 through 6 show how *Value* is processed when the numeric variable has a missing value.

```
%macro a ;                                                    What SAS sees:

  data a ;
    x = 123 ;                                                 Ruler:         <<         1  >>
    y =   . ;                                                        Column: <<123456789012>>
❶   call symput ('x_mv1', x) ;                          ➠       &x_mv1:   <<         123>>
❷   call symput ('x_mv2', put(x, 8.3)) ;                 ➠       &x_mv2:   << 123.000    >>
❸   call symput ('x_mv3', trim(left(put(x, 8.3)))) ;     ➠       &x_mv3:   <<123.000>>
❹   call symput ('y_mv1', y) ;                           ➠       &y_mv1:   <<           .>>
❺   call symput ('y_mv2', put(y, 8.3)) ;                 ➠       &y_mv2:   <<     .     >>
❻   call symput ('y_mv3', trim(left(put(y, 8.3)))) ;     ➠       &y_mv3:   <<.>>
    run ;
    %mend a ;
```

Figure 1

In Figure 2, we see what happens later in the execution stream as two of these macro variables, &*x_mv1* and &*y_mv1*, are used. The left column shows the DATA step code, while the right shows the macro compiler's "translation" of the code sent to Base SAS.

```
%macro b ;                                                    SAS log (MPRINT option):

  data b ;                                                    DATA B ;
❶ x_mv1_a = &x_mv1 ;                                    ➠     X_MV1_A = 123 ;
❷ x_mv1_b = input(&x_mv1, 8.3) ;                        ➠     X_MV1_B = INPUT( 123, 8.3) ;
❸ x_mv1_c = input("&x_mv1", 8.3);                       ➠     X_MV1_C = INPUT("         123", 8.3);
❹ x_mv1_d = input(trim(left("&x_mv1 ")),8.3);           ➠     X_MV1_D = INPUT(TRIM(LEFT("         123 ")), 8.3);
❺ y_mv1_a = &y_mv1 ;                                    ➠     Y_MV1_A = . ;
❻ y_mv1_b = input(&y_mv1, 8.) ;                         ➠     Y_MV1_B = INPUT( ., 8.) ;
❼ y_mv1_c = input("&y_mv1", 8.) ;                       ➠     Y_MV1_C = INPUT("           .", 8.) ;
❽ y_mv1_d = input(trim(left("&y_mv1 ")),8.);            ➠     Y_MV1_D = INPUT(TRIM(LEFT("           . ")), 8.);
    run ;                                                     RUN ;
  %mend b ;
```

Figure 2

In Line 1 of Figure 2, Base SAS sees the resolved *Value* of *&x_mv1* as the numeric argument of a simple assignment statement, and the Figure 3 output shows *X_MV1_A* to have been created appropriately. The same is true of *Y_MV1_A*, created in Line 5.

Now the trouble begins. The code in Line 2, creating *X_MV1_B*, shows *Value* as the numeric argument of an INPUT function, and its "translation" seems to be, again, a straightforward assignment statement. But a look at Figure 3 shows that *X_MV1_B* is missing -- a value that appears to be incorrect when compared with the SAS log. It is missing due to the 8.3 format specification which instructs Base SAS to look at only the first eight bytes of macro variable &*x_mv1* (Figure 1, Line 1), which are all blanks. SAS has done **exactly** what it was told to do, and **exactly** what the manual says it is going to do, but the result, probably, is neither desired, nor expected.

The problem with Line 6, the analogous line for *Y_MV1_B*, is subtler, for, while the variable has been assigned a missing value as intended, that missing value is **not** the result of the "dot" appearing in the assignment statement. Rather, *Y_MV1_B* is missing for the same reason as that for Line 2: SAS has assigned it a missing value because the first eight bytes of the argument to the INPUT function are blank.

Numeric Variables Created from CALL SYMPUT Macro Variables

Non-Missing Values

	❶	❷	❸	❹
OBS	X_MV1_A	X_MV1_B	X_MV1_C	X_MV1_D
1	123	.	.	0.123

Missing Values

	❺	❻	❼	❽
OBS	Y_MV1_A	Y_MV1_B	Y_MV1_C	Y_MV1_D
1	.	.	.	.

Figure 3

If the macro variable is brought in as quoted text, as shown in Lines 3 and 7 (Figure 2), the result is the same missing value -- for the same reason. Though not demonstrated here, the SYMGET function cannot handle this situation either. It does, however, terminate the DATA step with an error message: "The macro variable name is either all blank or missing." While this message is accurate, it is difficult to reconcile with the presence of the non-blank, resolved *Value* shown in the SAS Log, as on Line 3.

The above situations are precisely why we urge you to "take control" by writing clear, unambiguous code that allows you to know what a macro variable looks like as it sits out in the macro environment. The code shown in Lines 4 and 8 will help you do so consistently and confidently. Here, the macro variable is, first, read into "quoted text"; second, shifted to the left to remove any preceding blanks; and, third, trimmed of any trailing blanks. This quoted, shifted, trimmed text provides an unambiguous argument for the INPUT function.

The importance of specifying an appropriate format for the INPUT function can be seen by looking at Line 4. The format applied to macro variable &*x_mv1*, which is equal to '123', creates the database variable *X_MV1_D* as '0.123'. Note that *X_MV1_D* has **not** been created by any default action. Whether *X_MV1_D* has the correct value depends on what the programmer wished to do. If the programmer intended to write integers to the macro environment and to format those integers with the INPUT function, then *X_MV1_D* has been correctly assigned a value of '0.123'. However, if the programmer intended to create *X_MV1_D* as an integer with an 8.3 format ('123.000'), Line 4 should have been written with a sufficiently large integer informat for the INPUT function to read the full value, then followed by a FORMAT statement:

```
x_mv1_d  =  input(trim(left("&x_mv1 "), 10.) ;
format  x_mv1_d   8.3 ;
```

It may sometimes be appropriate to explicitly use BEST12. as an informat for *Value*, as shown in Figure 4, since BEST12. will read all twelve bytes of the macro variable and can be used with both integer and decimal numbers. However, do not assume that BEST12. can handle any number given it. It will always be limited by the 12-byte length of the CALL SYMPUT/BEST12. relationship.

Code	Resulting Value Assignment
❶ x mv1 d = input(trim(left("&x mv1")), best12.) ;	123
❷ x mv1 d = input(trim(left("&x mv1 ")), best12.3) ;	0.123

Figure 4

Figure 5 shows the problems that can occur if you use BEST12. when handling large numbers. In Line 1, the default action by BEST12. upon LARGENUM creates a truncated value for macro variable &*largenum*. If a DOLLAR20. format is used to create FROMNUM, as in Line 3, the number of significant digits in &*largenum* is further reduced by the need for commas and a dollar sign. This situation can be avoided, as shown in Lines 2 and 4, where the macro variable &*charnum* is created by explicitly converting LARGENUM to a trimmed, left-aligned,

20-byte *Value*, which is then used to create FROMCHAR. Note that the difference between FROMNUM and FROMCHAR is more than $90,000, and that FROMNUM has become a larger number due to the loss of significant digits and rounding. This type of conversion, bypassing CALL SYMPUT's default action, was shown earlier in a slightly different context (Figure 1, Lines 3 and 6). Creating *&charnum* with a PUT function format specification of twenty bytes allows for all the digits, commas and the dollar sign. To also show cents, you would simply increase the length of this specification to '23.' to provide for the decimal and pennies, and create FROMCHAR with a DOLLAR23.2 format.

Code		Data Value Assignment
`%macro c ;`		
`  data  null  ;`		
`    largenum = 12345678909876 ;`		
`❶  call symput ('largenum', largenum) ;`		<<123456789098>>
`❷  call symput ('charnum',  trim(left(put(largenum, 20.))));`		<<12345678909876>>
`    run ;`		
`  data c ;`		
`❸  fromnum  = input(trim(left("&largenum")), dollar20.) ;`	➪	$12,345,679,000,000
`❹  fromchar = input(trim(left("&charnum ")), dollar20.) ;`	➪	$12,345,678,909,876
`    format  fromnum  fromchar  dollar20. ;`		
`    run ;`		
`  %mend c ;`		

Figure 5

Character Values

When *Value* is a character value, CALL SYMPUT does **not** use BEST12. Instead, it uses the format of *Value* that is applicable at the moment the macro variable is defined. Usually, but not always, the applicable format is the same as *Value*'s initial definition. While no default format is being employed, it is important to understand that a default action **is** being taken by CALL SYMPUT -- an action that uses the format of *Value* to determine how to place its value into the macro variable *Name*.

Before proceeding, we need to review the many ways in which a character variable may be defined in Base SAS:

1. a LENGTH statement: `LENGTH  charvar  $10 ;`
2. an INPUT statement:
   - with a complete format specification (length is 6):
     a. values of charvar will be left-aligned: `INPUT  @1  charvar  $6. ;`
     b. values of charvar will be right-aligned: `INPUT  @1  charvar  $char6. ;`
   - with a partial format specification (length defaults to 8): `INPUT  @1  charvar  $ ;`
3. the first appearance in an assignment statement:
   - with a literal value (length of ANSWER is 3): `ANSWER = 'Yes' ;  ANSWER = 'Maybe' ;`
   - using other variables (length of FULLRESP is the sum of the lengths of PART_1, PART_2 and PART_3): `FULLRESP = PART_1 || PART_2 || PART_3 ;`

The examples that follow illustrate how the format of *Value* can affect the creation of *Name*.

In Line 1 of Figure 6, the %LET statement creates *&macname1* in the macro environment (see right column) as a 6-byte, left-aligned macro variable. In Line 2, the *Value* argument to CALL SYMPUT is a 6-byte text literal, which is, by definition, left-aligned. Therefore, *&macname2* also becomes a 6-byte, left-aligned macro variable.

Lines 3 through 5 use a data set variable SUBJNAME for *Value* in the CALL SYMPUT statement. Since SUBJNAME has been defined to be a left-aligned, 15-byte character variable, the code in Line 3 creates *&macname3* as a left-aligned, 15-byte macro variable ('J. Doe' followed by nine blanks). In Line 4, the PUT function uses a $13. format to indicate how SUBJNAME is to be handled, and creates *&macname4* as a left-aligned, 13-byte macro variable ('J. Doe' followed by seven blanks). Then, in Line 5, the operation of the RIGHT function upon SUBJNAME creates *&macname5* as a right-aligned, 15-byte macro variable ('J. Doe' preceded by nine blanks). While macro variables

&macname4 and *&macname5* were created with an explicit formatting of *Value* in the CALL SYMPUT statement, each result would have been the same if *Value* had been a data set variable defined with those properties.

```
                                                                What SAS sees:
                                                                Ruler:     <<          1     >>
%macro d ;                                                         Column: <<123456789012345>>
❶ %let macname1 = J. Doe ;                                ➠   &macname1: <<J. Doe>>
 data d ;
❷ call symput ('macname2', 'J. Doe') ;                    ➠   &macname2 <<J. Doe>>
   length  subjname $15 ; subjname = 'J. Doe' ;
❸ call symput ('macname3', subjname) ;                    ➠   &macname3 <<J. Doe          >>
❹ call symput ('macname4', put(subjname, $13.)) ;         ➠   &macname4 <<J. Doe        >>
❺ call symput ('macname5', right(subjname)) ;             ➠   &macname5 <<          J. Doe>>
❻ call symput ('macname6', trim(left(put(subjname, $15.)))); ➠ &macname6 <<J. Doe>>
   run ;
   %mend d ;
```

Figure 6

The different effects of these macro variables is easily seen in Figure 7, where each line was created by placing the respective macro variable name in a TITLE statement, for example:

title "Placement of *&macname4* in Title" ;

The examples in Figure 7 affect only the appearance of the TITLE statements, not the accuracy of the information. If such macro variables were used in creating data set variables, however, there is a high probability of erroneous value assignment.

A programmer can control this situation by writing code as shown in Figure 6, Line 6 -- the result of which is seen in the last line of Figure 7. As with numeric variables, the use of TRIM, LEFT and PUT functions to explicitly manipulate *Value* allows the programmer to know what is placed in, and read from, the macro environment.

```
&macname1, &macname2:                      Placement of J. Doe in Title
&macname3:                               Placement of J. Doe          in Title
&macname4:                               Placement of          J. Doe in Title
&macname5:                                Placement of        J. Doe in Title
&macname6:                                 Placement of J. Doe in Title
```

Figure 7

Null Values

Even though macro variables are defined as text variables, or 'character-based', a 'missing' value for a macro variable is not a *blank*, as you might expect from Base SAS definitions. Rather, it is *null*, or non-existent. Assigning a null value to a macro variable is often done when its assignment depends upon data to be encountered downstream, or upon a sequence of events determined by branching code. A macro variable is usually initialized as null by: (1) a %GLOBAL statement, as discussed earlier; or (2) a %LET statement without a specified value, as in

```
%LET macvar1 = ;
```

A null macro variable may also be produced by a CALL SYMPUT statement during program execution -- if, for example, the CALL SYMPUT occurs within a block of conditional code that is never executed. As a general rule, you should write code that either allows for the possibility of null values, or ensures that null values will never be assigned.

The important issue, however, is what happens when a macro variable with a null value is referred to in an assignment statement or an INPUT function argument. The code in Lines 1 and 2 of Figure 8 may look like valid code, but when the macro variable has a null value, each will produce a DATA step error from the Base SAS compiler. In Line 1, this is because you must assign a value to a numeric variable in a DATA step -- it may be missing, but it cannot be null. Line 2 will produce an error because the INPUT function does not have a value argument, only a format specification. Lines 3 and 4 will not produce errors, but each will result in the assignment

of a missing value to the DATA step variable, which, then, may have a significant effect upon program execution. On a side note, notice in the column "What SAS sees" for lines 3 and 4, that, because the macro variable is null, the beginning and ending quotes are immediately adjacent to one another. When this prints in the Log, it may appear that a coding problem exists – perhaps a typo or an omission -- and the question may be "What is happening?" To be certain the program has executed properly, a reader must be very familiar with the code, knowing where macro references appear. Note in Line 5, however, that the addition of a single space between the last character of the macro reference and the ending quote creates a clear, unambiguous statement in the Log. The question now becomes "Should N_E be assigned a missing value?"

Code:		What SAS sees:	Note:
❶ n_a = &macvar1 ;	➠	N_A = ;	** Error:
❷ n_b = input(&macvar1, best12.0) ; length n_c $8 ;	➠	N_B= INPUT(, BEST12.0) ; LENGTH N_C $8 ;	** Error:
❸ n_c = "&macvar1" ;	➠	N_C = "" ;	
❹ n_d = input("&macvar1", best12.0) ;	➠	N_D = INPUT("", BEST12.0) ;	
❺ n_e = input("&macvar1 ", best12.0) ;	➠	N_E = INPUT(" ", BEST12.0) ;	

Figure 8

CONCLUSIONS

One of the primary uses of SAS macro variables is to make information available to various parts of a program. The CALL SYMPUT (*Name*, *Value*) statement provides for the creation of a macro variable whose value may vary depending upon information obtained within a DATA step at the time a program is run.

The value assigned to the macro variable *Name* depends upon the properties of *Value* -- its type, length, alignment and format -- at the moment the CALL SYMPUT statement is executed. By explicitly manipulating the properties of *Value*, a programmer can avoid problems resulting from SAS default actions and retrieve information from the macro environment with greater reliability.

Fortunately, it is very easy to exercise control over the properties of *Value* whether using a CALL SYMPUT statement to create the macro variable, or making a subsequent reference to the macro variable: (1) use the PUT function with an appropriate format specification to explicitly convert or define *Value* as a character string; (2) use the TRIM and LEFT functions to control the presence of blanks; and (3) include a single blank space after a macro variable name whenever it appears between double quotes (e.g., "*&macvar* "). Used together, these tools enable a programmer to write code that can easily be interpreted whenever a macro name reference is present, and that clearly and unambiguously assigns *Value* to *Name*.

REFERENCES

SAS® Language Reference, Version 6, 1st Edition.

SAS® Guide to Macro Processing, Version 6, 2nd Edition.

SAS® Macro Facility: Tips and Techniques, Version 6, 1st Edition.

AUTHORS' ADDRESSES

K. Larry Landers
c/o Trilogy Consulting Corporation, Inc.
155 Bovet Road, Suite 201
San Mateo, CA 94402
(800) 515-3502
handl@flash.net

Monique Bryher
MRI Consulting, Inc.
6043 Shirley Avenue
Tarzana, CA 91356
(818) 774-0043
rebwest@aol.com