

ET09

Data Step Interfaces with the Macro Facility

Toby Dunn, Dunn Consulting, San Antonio, Tx

Abstract:

Traditionally when one thinks of how a Macro works within a program, the process is considered to be hierarchical in nature, i.e. the Macro compiles and executes before the Data Step compiles and executes. Primarily, this thought concept poses no problems, but it does however, hamper the ability to use data step variables with macros or even load data step values into a macro variable. Lucky for us SAS® has a whole suite of functions that allows the Data Step to interact with the Macro facility at Data Step execution. This paper will explore cover the uses and possible problems when using the following Data Step interfaces: Call Execute, Resolve, Call Symdel, Symexist, Symget, SymgetN, SymGlobl, Symlocal, Call Symput, Call SymputN, and Call SymputX. The intended audience is intermediate to advanced SAS programmers.

Introduction

There are many occasions when it is beneficial to have the data talk to or drive a program. SAS provides a whole host of call routines and function for just this purpose. The way to achieve this functionality is by having the Data Step interface with the Macro Facility. However, due to the differences in the compilation and execution times of the two different languages a seemingly easy task becomes one of frustration and disappointment. Although each of the call routines and functions we will explore could in their own right have a paper dedicated to them this paper will condense the information down to the essential knowledge needed to successfully run them and review problems and pitfalls to avoid when using them.

I have often stated that these call routines and functions should not be used by beginner programmers. The reason is that they twist around the different compile and execute times of the Macro code and Data Step code. Unexpected results can occur if these actions are not perfectly clear in the mind of the programmer. During the normal operation of Macro code and Data Step code, Macro code is compiled and executed firsthand then any Data Step code is compiled and executed. This is simple, straightforward and makes a whole lot of sense given Macro code generates Data Step code. When a Data Step interface to the Macro facility is used the two steps are intertwined with each other. The Data Step code is compiled and executed then halted while it is executing the Data Step is halted and control gets passed to the Macro facility to compile and execute the Macro code. When the Macro Facility is finished the resulting code and control is passed returned to the Data Step. This intermingling of compilation and execution times has misled more than a few programmers. Understanding the process can be a daunting task but is crucial to getting correct results when using Data Step interfaces to the Macro Facility.

Call Symput

Since Call Symput is probably the best known and most widely used Data Step interfaces to the Macro Facility. It only makes sense to begin our review here. Call Symput assigns a Data Step value to a macro variable. Even with its extensive online documentation and numerous papers that can still leave many programmers baffled and confused. Before delving into the problems that can be encountered when using Call Symput, let us look at the syntax and its basic functionality.

Syntax: Call Symput(<Macro Variable Name> , <Character Value>)

The first argument to the Symput routine is the name of the macro variable to be assigned the value from the second argument. If the macro variable already exists it will be assigned the value and if it does not exist it will be created. The argument can take the form of a literal string enclosed in quotes, the name of a Data Step character variable, or some combination thereof.

The second argument is the character value that will be assigned to the macro variable. This value can be a literal string enclosed in quotes, a Data Step character or numeric variable, or some combination thereof. The important thing to note is that the second argument must be a character value. If a numeric value or variable is to be used it must first be converted to a character value. If no explicit conversion is performed SAS will implicitly convert the numeric value to a character value. As we will see a little later in this section this may lead to problems.

```
Data _Null_ ;
Char = 'Character Value' ;
Num = 123 ;

Call Symput( 'Character' , Char ) ;
Call Symput( 'Numeric' , Left( Put( Num , 8. ) ) ) ;
Call Symput( 'Combo' , Char||' And '||Left( Put( Num , 8. ) ) ) ;
run ;

%put _User_ ;
```

The following results are written to the Log:

```
GLOBAL COMBO Character Value And 123
GLOBAL NUMERIC 123
GLOBAL CHARACTER Character Value
```

There are five general problems that may occur with the use of Call Symput. The first problem is when a programmer attempts to use the macro variable before a step boundary is reached. The second one is where a programmer tries to create a macro variable from every observation in a data set for a given variable. The next two problems occur when using a numeric and character variable and the results get truncated or extra spaces pad the values. Finally, we will review problems with the scope of the macro variable created by the Call Symput routine.

Normally, all Macro code is compiled and executed before the Data Step code compiles and executes. As long as the macro variable is created before the compilation of the Data Step code the macro variable can be referenced inside of the Data Step with no problems. When Call Symput is used to create a macro variable, the compilation time of the macro variable occurs somewhere in the middle of the execution of the Data Step code. If the Data Step code references a macro variable that is created by Call Symput SAS will try to resolve it at the time of the macro compilation and execution time. Since Call Symput gets executed at the Data Steps execution time the macro variable is not available to be used.

The following code and log demonstrate just such a scenario:

```
:
Data _Null_ ;
938 Var1 = 'Alpha' ;
939
940 Call Symput( 'MyVar' , Var1 ) ;
941
942 Var2 = "&MyVar" ;
WARNING: Apparent symbolic reference MYVAR not resolved.
943
944 put Var1= Var2= ;
```

```
945 run ;
```

```
Var1=Alpha Var2=&MyVar
```

The assignment statement to Var2 did not work properly because SAS tried to resolve the reference to MyVar during the time of the Macro compilation and execution. However, MyVar has not yet been created and will not until the execution of the Data Step. SAS issues a warning and since SAS cannot resolve the &MyVar and regards the statement as if it were just text and assigns that value to Var2.

A similar problem occurs when there is no explicit step boundary and the macro variable created in Call Symput is used in a global statement.

```
Data _Null_ ;  
Var1 = 'Alpha Beta Theta' ;  
  
Call Symput( 'MyTitle' , Var1 ) ;
```

```
Title "&MyTitle" ;  
Proc Print  
  data = AAA ;  
run ;
```

In this case there is no explicit step boundary or in this case run statement to end the data step. Normally SAS begins compiling the code and does not stop until encountering the Proc in Proc Print. However, in this case, SAS finds the Title statement preceding the Proc and since the Title statement is global, it executes it immediately. The problem with this scenario is the Title statement references the macro variable MyTitle which has not yet been created. The result is that the data set AAA will be printed and the title will be have the text '&MyTitle'. Simply adding a run statement would have solved this problem, so if anyone tells you that a step boundary is not necessary you can tell them they are wrong and can prove why.

Another problem that many have with Call Symput is creating some number of macro variables from the values in a data step variable. This usually means the design of the program is wrong and the problem occurs often enough to be addressed.

```
Data MyColors ;  
infile cards ;  
input Colors $ ;  
cards ;  
Blue  
Red  
Green  
Yellow  
;  
run ;  
  
Data _Null_ ;  
set MyColors ;  
Call Symput( 'Color' , Colors ) ;  
run ;  
  
%put _User_ ;
```

The log reveals the following:

GLOBAL COLOR Yellow

As the example shows the Call Symput only created one macro variable and not the four that were expected. Or did it? In actuality Call Symput created four macro variables; unfortunately they all were called the same name. Therefore, SAS overwrote the value for the macro variable with each iteration of the data step. The way to avoid this problem is to add a unique prefix or suffix to the macro variable name. Commonly this is accomplished by using a counter or the automatic variable `_N_` as a suffix.

```
Data _Null_ ;  
set MyColors ;  
Call Symput( 'Color'||Left( put( _N_ , 8. ) ) , Colors ) ;  
run ;
```

```
%put _User_ ;
```

The following is written to the Log:

```
GLOBAL COLOR2 Red  
GLOBAL COLOR3 Green  
GLOBAL COLOR4 Yellow  
GLOBAL COLOR1 Blue
```

As mentioned earlier the second argument to Call Symput can be a character value, a numeric value, or some combination. Regardless of the type of the value going into the macro variable it must be regarded as a character value. If no manipulation of the data step value is performed SAS will use the Best12 format to convert the numeric values to character values. For character values SAS will use the format associated with the variable or utilize the Best8 format if one is not. This implicit conversion leads to some rather interesting results which may not be the results intended.

Consider the case where a numeric value that is longer than 12 numbers is used.

```
984 Data _Null_ ;  
985 Num = 123456789012345 ;  
986 Call Symput( 'MyNum' , Num ) ;  
987 put Num= Best15. ;  
988 run ;
```

Resulting Log Notes:

NOTE: Numeric values have been converted to character
values at the places given by: (Line):(Column).

986:24

Num=123456789012345

NOTE: DATA statement used (Total process time):

real time 0.00 seconds

cpu time 0.00 seconds

```
989  
990  
991 Data _Null_ ;  
992 Num = &MyNum ;  
993 put Num= Best15. ;  
994 run ;
```

Num=1234567900000000

As can be seen from the above log there was a lack of numeric precision due to this explicit conversion. If however, an explicit conversion is made with the appropriate format type and length then the problem is solved.

```
995 Data _Null_ ;
996 Num = 123456789012345 ;
997 Call Symput( 'MyNum' , Left( Put( Num , 15. ) ) ) ;
998 put Num= Best15. ;
999 run ;
```

Num=123456789012345

NOTE: DATA statement used (Total process time):

real time	0.00 seconds
cpu time	0.01 seconds

```
1000
1001
1002 Data _Null_ ;
1003 Num = &MyNum ;
1004 put Num= Best15. ;
1005 run ;
```

Num=123456789012345

NOTE: DATA statement used (Total process time):

real time	0.00 seconds
cpu time	0.00 seconds

Why did I use the Left function? The reason is all numeric values are right aligned in SAS so to avoid possible leading spaces the Left function is used to remove them. I choose to use the Left function but in version 9 the strip function will also work.

At this point you be thinking that when using numeric values there are special considerations that must be compensated for. But what could go wrong with a character value? Well if the character value that is contained in a variable has fewer characters than 8 is possible to get trailing blanks in the macro variable. When used in something like a title statement this causes the title to have some number of blank spaces between the end of the value resolved and the rest of the text. So use the trim function (or the Strip function in version 9). Let us review a few examples to see what will happen:

```
Data _Null_ ;
Length MyTitle $8 ;
MyTitle = 'Alpha' ;
Call Symput( 'MyTitle1' , MyTitle ) ;
Call Symput( 'MyTitle2' , Trim(MyTitle) ) ;
run ;
```

```
%put MyTitle1 = >>>&MyTitle1<<< ;
%put MyTitle2 = >>>&MyTitle2<<< ;
```

Resulting Log:

```
MyTitle1 = >>>Alpha <<<
MyTitle2 = >>>Alpha<<<
```

The last problem that needs to be addressed is the scope of the resulting macro variable when using Call Symput. Starting with version 8 SAS will attempt to assign the macro variable to the most local symbol table and work outward until reaching the global symbol table. There are three case which are an exception to this rule: when Call Symput is used after a SQL statement, when a computed %GOTO statement is used, or when %Sysbuff and Call Symput are used jointly. In these cases the scope of the macro variable will be local to the macro which used the Call Symput unless the macro variable has been previously defined in another symbol table. Since the scope of any macro variable created by Call Symput in open code will always be assigned to the global symbol table the only time to worry about the scope is when Call Symput is used inside of a macro.

```
%macro ScopeTest ;

Proc sql noprint ;
  select *
    from dictionary.columns ;
quit ;

Data _Null_ ;
call symput( 'ScopeTest' , 'Local' ) ;
run ;

%put _User_ ;
%mend ScopeTest ;

%ScopeTest
```

Resulting Log:

```
SCOPETEST SQLLOOPS 12
SCOPETEST SQLRC 0
SCOPETEST SQLOBS 1
SCOPETEST SCOPETEST Local
SCOPETEST SQLXOBS 0
```

As can be seen the scope of the macro variable ScopeTest is local to the macro ScopeTest. Explicitly declare the scope in a %global statement to circumvent this default action.

```
%macro ScopeTest ;
%Global ScopeTest ;

Proc sql noprint ;
  select *
    from dictionary.columns ;
quit ;

Data _Null_ ;
call symput( 'ScopeTest' , 'Global' ) ;
run ;

%put _User_ ;
%mend ScopeTest ;

%ScopeTest
```

Results in the following being printed to the Log:

```
SCOPETEST SQLLOOPS 12
SCOPETEST SQLRC 0
SCOPETEST SQLOBS 1
SCOPETEST SQLXOBS 0
GLOBAL SCOPETEST Global
```

Call SymputX

Obviously there are quite a few pitfalls that can cause problems with Call Symput. To resolve these issues SAS implemented Call SymputX in version 9, which is like Call Symput on steroids. Call SymputX not only assigns Data Step values to a macro variable at the execution time of the Data Step, but also strips leading and trailing blanks to allow the programmer to choose the scope of the macro variable. If one is using version 9 or higher Call SymputX is the natural choice over Call Symput.

Syntax: Call SymputX(<Macro Variable Name> , <Character Value> , <Symbol Table>)

The first two arguments to Call SymputX are the same as Call Symput. The third argument is optional and allows the programmer to specify the symbol table to assign the macro variable too. It can be a quoted literal text string, a Data Step character variable, or some expression that results in a valid character value. There are three valid values G, L, and F. A value of G specifies that the macro variable be stored in the global symbol table regardless of whether or not SAS can local it in a local symbol table. A value of L specifies that SAS is to store the macro in the most local symbol table. For A value of F (or if no value is specified), SAS uses the same scoping rules as Call Symput.

Consider the Call Symput scoping problem and substituting Call SymputX instead.

```
%macro ScopeTest ;

Proc sql noprint ;
  select *
    from dictionary.columns ;
quit ;

Data _Null_ ;
Call SymputX( 'ScopeTest' , 'Global' , 'G' ) ;
run ;

%put _User_ ;
%mend ScopeTest ;

%ScopeTest
```

Results in the following being printed to the Log:

```
SCOPETEST SQLLOOPS 12
SCOPETEST SQLRC 0
SCOPETEST SQLOBS 1
SCOPETEST SQLXOBS 0
GLOBAL SCOPETEST Global
```

As can be seen from the code and log output using Call SymputX with the third argument specified as 'G' does in fact allocate the macro variable ScopeTest to the global symbol table.

Let us now compare how to get the same Call Symput results when handling the Numeric and Character problems previously discussed.

Numeric Problem:

```
1078 Data _Null_ ;
1079 Num = 123456789012345 ;
1080 Call SymputX( 'MyNum' , Put( Num , 15. ) ) ;
1081 put Num= Best15. ;
1082 run ;
```

Num=123456789012345

NOTE: DATA statement used (Total process time):

real time	0.00 seconds
cpu time	0.00 seconds

```
1083
1084 Data _Null_ ;
1085 Num = &MyNum ;
1086 put Num= Best15. ;
1087 run ;
```

Num=123456789012345

The use of Call SymputX and a numeric format that is large enough to encompass the number.

Character Problem:

```
Data _Null_ ;
Length MyTitle $8 ;
MyTitle = 'Alpha' ;
Call SymputX( 'MyTitle1' , MyTitle ) ;
run ;
```

Results in the following log:

MyTitle1 = >>>Alpha<<<

As the log output demonstrates Call SymputX handles everything for us.

Call SymputN

Is Call SymputN the answer to the numeric variable problem previously mentioned? No, it is not. Unlike SymgetN, Call SymputN can only be used in SCL programs. Use Call SymputN only when developing SCL code.

Call Execute

Call Execute is designed to resolve the argument passed it. Any open code will be placed upon a stack and executed after the associated Data Step is finished. Sounds pretty simple, and normally it would be but when macro code is involved this function can be quite confusing.

Syntax: Call Execute(Argument)

Call Execute can contain a text string enclosed in quote, the name of a Data Step character variable, or an expression that resolves to a character value. It is important to note that when macro code is passed to Call Execute it must be enclosed in single quotes to delay the compilation and execution so that this procedure will occur at the Data Steps execution time rather than the macros.

```
Data _Null_ ;  
Name = 'ABC' ;  
Call Execute( 'Proc Print' ) ;  
Call Execute( 'Data = ' || Name || ';' ) ;  
Call Execute( 'Run ;' ) ;  
Run ;
```

The code shown will take the value in variable Name add it to the text inside of the Call Execute code and it will place all of that code in a memory stack. When the Data _Null_ step is finished it will then compile and execute the proc print code. While this may not look too helpful at first, imagine running the same code where there are many values for Name. Now the true power of Call Execute begins to shine.

Since writing multiple Call Execute statements can become tedious and error prone consider how the whole process could be wrapped in a macro and then that macro call used in the Call Execute statement.

```
%Macro Print( Name = ) ;  
  Proc Print  
    Data = &Name ;  
  Run ;  
%Mend Print ;  
  
Data _Null_ ;  
Name = 'ABC' ;  
Call Execute( '%Print( Name = " || Name || "' ) ) ;  
Run ;
```

The code now produces the exact same output as the previous example. What occurred here was that SAS recognized that the Call Execute contained macro code. SAS then halted the execution of the Data Step and passed control over to the macro facility. The macro facility resolved the macro and that resolution produced open code it passed that open code back to the Data Step. Call Execute now has something it could deal with and promptly did what it does best and placed that code in the memory stack to be executed when the Data step finished. The important thing to remember is that the macro call inside of the Call Execute has to be enclosed inside of single quotes so that it does not get executed before the Data Step.

Notice that the macro code, in the previous example the &Name inside of the macro gets resolved so that the proc print code from proc to the run statement will be fully resolved and ready to be executed as soon as the Data Step is finished. The result is all macro code is executed immediately. Thus, macro code that requires some step to generate (such as a macro variable) may not work as it did when called outside of a Call Execute statement.

```
%Macro Test ;
```

```
Data _Null_ ;
  Call SymputX( 'MyVar' , 'ABCDE' ) ;
Run ;
```

```
%put ABCDE = &MyVar ;
%Mend ;
```

```
Data _Null_ ;
  Call Execute( '%Test' ) ;
Run ;
```

Produces the following in the log:

```
WARNING: Apparent symbolic reference MYVAR not resolved.
ABCDE = &MyVar
```

NOTE: CALL EXECUTE generated line.

```
1 + Data _Null_ ;    Call SymputX( 'MyVar' , 'ABCDE' ) ;    Run ;
```

Notice that the put statement did not work as expected and that the call execute code was compiled and executed after the put statement. Since the %put statement is macro code it gets compiled and executed immediately while the Data Step code is held and run when the put statement is finished. To overcome this limitation the execution of the called macro needs to be delayed by use of the %Nrstr macro quoting function. %Nrstr works at compile time only and thus will cause the macro facility to pass the macro call back to the Data Step which will place the macro call in the memory stack and execute the macro after the calling Data Step finishes. Remember that the %Nrstr must be placed inside of the single quotes so that it gets executed when the macro call is passed to the macro facility from the Data Step. Otherwise it will be executed during the normal macro compile time and the called macro would be resolved when the Call Execute passes the code to the macro facility.

```
%Macro Test ;
  Data _Null_ ;
  Call SymputX( 'MyVar' , 'ABCDE' ) ;
Run ;
```

```
%put ABCDE = &MyVar ;
%Mend ;
```

```
Data _Null_ ;
  Call Execute( '%nrstr( %Test' ) );
Run ;
```

Produces the following in the log:

```
+ %Test
```

NOTE: DATA statement used (Total process time):

```
real time      0.00 seconds
cpu time       0.00 seconds
```

```
ABCDE = ABCDE
```

Symget, SymgetN, and Resolve

Now that we have reviewed how to create macro variables from within the Data Step it is time to consider how to retrieve a macro variables value from within the Data Step during the execution time of the Data Step. There are three functions with which to do this Symget, SymgetN and the Resolve function. We will look at each one separately and then we will look at how to use these three functions to retrieve a macro variables value that was created within the same Data Step.

Symget

The Symget function returns the value of a macro variable to the Data Step as a character value to the Data Step during the execution of that Data Step.

Syntax: Symget(Argument)

The argument must be the name of a macro variable within quotes, a Data Step character variable, or some expression which when resolved will yield a macro variables name.

```
%let C = CCC ;  
  
Data _Null_ ;  
  
Y = Symget( 'C' ) ;  
  
Put Y= ;  
Run ;
```

The log shows:

```
Y=CCC
```

There are three issues that the programmer should keep in mind. The first concern is if the variable to receive the value from Symget has not explicitly already had its length determined the length will be set to the maximum allowable length for a character variable in a Data Step. This also means that if the value contained in the resolved macro variable is longer than the allowable length of a character variable in a Data Step then the returned value will be truncated. The second matter to watch for when using Symget is that if the macro variable cannot be found Symget will return a missing value and cause a nonfatal error, issuing a warning to the log. If you are using version 9 the best way to avoid this problem is to use SymExist, Symglobl, or Symlocal (refer to the SymExist, Symglobl, or Symlocal section for example). The last thing to remember is the fact that Symget will not continue to resolve a macro variables value. So if a macro variable has a value which can still be resolved further Symget will not attempt to do so and instead will return the first unresolved value.

SymgetN

SymgetN is a SCL only function according to the SAS documentation. However, in version 9.1.3 SymgetN can be used in the Data Step. A word of warning, what you are about to read is undocumented and has only been tested in version 9.1.3. SymgetN works much like Symget except that it retrieves values from a

macro variable that can be interpreted as a numeric. The Data Step variable receiving this value that has not explicitly had its type and length specified will be of type numeric with a length of 8. Like Symget, if SymgetN cannot find the macro variable it will return a missing value, cause a nonfatal error, and issue a warning in the log.

Syntax: SymgetN(Argument)

The argument must be the name of a macro variable within quotes, a Data Step character variable, or some expression which when resolved will yield a macro variables name.

```
%let Num = 123 ;
```

```
Data _Null_ ;  
Num = SymgetN( 'Num' ) ;  
Type = Vtype( Num ) ;
```

```
Put Num= Type= ;  
Run ;
```

Results in the following being written to the log:

```
Num=123 Type=N
```

Resolve

The Resolve function is a hybrid of Symput, Symget, and Call Execute. Why would I say such a thing? Well, Resolve will not only retrieve a macro variables value but it will also execute macro code. Thus we can retrieve the value of a macro, pass it in a %let statement, create a macro variable, and run a macro.

Syntax: Resolve(Argument)

The argument is either a text expression with single quotes with the ampersands or percent signs included, a Data Step character variable, or some character expression that will result in a valid macro variable name or macro name with the appropriate tokens included.

```
%let X = 'Test' ;
```

```
Data _Null_ ;  
X = Resolve( '&X' ) ;  
Put X= ;  
Run ;
```

In log:

```
X=Test
```

Taking this example one step further we can accomplish the following (which cannot be done with Symget):

```
%let X = Test ;  
%let Y = X ;
```

```
Data _Null_ ;
X = Resolve( '&&&Y' ) ;
Put X= ;
Run ;
```

In Log:

X=Test

Use the Resolve function to create macro variables by supplying a %let statement as the argument.

```
Data _Null_ ;
X = Resolve( '%let YYY = YYY ; ' ) ;
X= ;
Run ;
%put YYY = &YYY ;
```

In the Log we see:

X=

YYY = YYY

The Resolve function sent the macro code to the Macro facility. The code was processed and the macro variable YYY was created. Variable X was not assigned a value since executing the macro did not resolve to any code or value to return back to the Data Step.

To take the use of macro code to its conclusion with regards to the Resolve function one need only call a macro with the Resolve function.

```
%Macro Multi( Number= ) ;
    %eval( &Nnumber * 100 )
%Mend Multi ;
```

```
Data _Null_ ;
X = 25 ;
```

```
Y = Resolve( '%Multi( Number = ' || Left( Put( X , 8. ) ) || ' ) ' ) ;
```

```
Put X= Y= ;
Run ;
```

The resulting log shows:

X=25 Y=2500

The code shown above works because the code contained inside of the macro is pure macro code and a function style macro. This means running the macro will create a value that can be returned back to the Data Step and be assigned to variable Y. Be careful if the macro resolves to Data Step code that code will be returned back to the Data Step variable.

```
%Macro Test ;
    Data One ;
```

```

    X = 1 ;
  Run ;
%Mend Test ;
Data _Null_ ;
X = Resolve( '%Test' ) ;
Put X= ;
Run ;

```

Results in the following log:

```

X=Data One ;    X = 1 ;    Run ;

```

The Data Step code inside of the Macro Test has now becomes the value for the Data Step variable X. Before we leave the issue of the Resolve function I must mention some warnings. While the Resolve function can do more than the Symget function keep in mind it does requires greater resources. If the Resolve function cannot find the macro or macro variable it will issue a warning and return the argument back to the Data Step. Finally, if the Resolve function is used in an assignment statement and the length of the variable receiving the value from the Resolve function has not been set then the length will be set to the maximum length of a character variable.

Many programmers think that a macro variable can not be created in a Data Step and then retrieved and used. This would be true except for the three functions previously mentioned will allow this (normally the macro is referenced outside of these functions). In these cases the macro facility tries to resolve these macro variables before they are created. However, these three functions (Symget, SymgetN, and Resolve) allow the user to delay the execution of the macro variables until the Data Steps execution time and at which time the macro variables will be available.

```

Data _Null_ ;

Call SymputX( 'MyMVar' , '123' ) ;

Symget  = Symget( 'MyMVar' ) ;
SymgetN = SymgetN( 'MyMVar' ) ;
Resolve = Resolve( '&MyMVar' ) ;

Put Symget= SymgetN= Resolve= ;
Run ;

```

Produces the following in the log:

```

Symget=123 SymgetN=123 Resolve=123

```

Call Symdel

Many times it is advantageous to be able to delete global macro variables. However, if there are quite a few macro variables to delete it can become cumbersome to use %Symdel and write out all of them. In these cases Call Symdel is the tool for the job. Call Symdel deletes global macro variables from the global symbol table. The reason that it only deletes global macro variables is that local macro variables do not persist beyond the scope of the macro in which they reside.

Syntax: Call Symdel(<Macro Variable Name> , option)

The first argument to Call Symdel is the name of the macro variable you wish to have deleted. It can be in the form of a quoted literal text string, a Data Step character variable, or an expression which will resolve to a valid macro variable name. The second argument is optional and allows the programmer to turn off the default warning when an attempt to delete a nonexistent macro variable occurs. By default it is set to 'WARN' and can be turned off by using 'NOWARN'.

The best example of this situation is where all of the global macro variables need to be deleted. Anyone who has ever run SAS in interactive mode knows that not cleaning up your work space is just asking for a disaster to happen.

```
Data MVars ;  
Set SASHelp.VMacro ;  
Where Scope = 'GLOBAL' ;  
Run ;
```

```
Data _Null_ ;  
Set MVars ;  
  Call Symdel( Name ) ;  
Run ;
```

The code above will delete all user defined global macro variables that exist in a SAS session. It should be noted that the first Data Step is needed due to the fact that SAS will not allow one to delete macro variables from the table while it is reading it.

SymExist, SymGlobl, and SymLocal

Starting in version 9 SAS added SymExist, SymGlobl, and SymLocal. These three new functions test for the existence of a macro variable and how a macro variable will resolve. While these functions by themselves are pretty fairly useless their true worth shines through when they are combined with the other Data Step interfaces to the Macro Facility.

```
Syntax: SymExist( Argument )  
        SymGlobl( Argument )  
        SymLocal( Argument )
```

The argument can be the name of a macro variable in double quotes with no ampersand, a Data Step character variable, or an expression which results in a valid macro variable name.

SymExist looks for the macro variable in any symbol table starting with the most local symbol table and works its way outward to the global symbol table. While SymGlobl and SymLocal will indicate if a given macro variable will resolve using the global or local macro symbol table.

All three of these functions will return a 1 if the macro variable name is found or a value 0 if the macro variable name cannot be found.

```
%Macro Symbol ;  
%Local LMVar NMvar ;
```

```
Data _Null_ ;
```

```
if SymExist( "NMvar" ) then Put 'NMVar Exists' ;  
if SymGlobl( "GMvar" ) then Put 'GMVar Is Global' ;
```

```
if SymLocal( "LMvar" ) then Put 'LMVar Is Local' ;  
run ;
```

```
%Mend ;
```

```
%Global GMvar NMVar ;
```

```
%Symbol
```

Writes the following in the log:

```
NMVar Exists  
GMVar Is Global  
LMVar Is Local
```

Although the detection of the macro variables seems straight forward this is not always the case. Consider a name collision situation:

```
%Macro Symbol ;  
%Local Mvar ;
```

```
Data _Null_ ;
```

```
if SymExist( "Mvar" ) then Put 'MVar Exists' ;  
if SymGlobl( "Mvar" ) then Put 'MVar Is Global' ;  
if SymLocal( "Mvar" ) then Put 'MVar Is Local' ;  
run ;
```

```
%Mend ;
```

```
%Global Mvar ;  
%Symbol
```

Writes the following in the log:

```
MVar Exists  
MVar Is Local
```

What happened with SymGlobl? SymGlobl and SymLocal only indicate which symbol table if any the macro variable will use when the variable is resolved. Even though the macro variable Mvar exists as both a global and a local macro variable when used inside of a macro the local macro variable will be used so SymGlobl returns a 0. This is just one more reason why variables names should be chosen with care.

As previously mentioned the true power of these functions is only seen when they are used with the other Data Step interfaces with the Macro Facility. Recall when we discussed Symget and when a non-existing macro name is used, Symget would return a missing value, issue a warning to the log, and cause a non-fatal error. This problem can be alleviated by the use of these functions to check that the macro variable you are trying to retrieve is actually there.

```
Data _Null_ ;  
If SymExist( "MVAR" ) then do MyVar = Symget( 'MyVar' ) ;  
Run ;
```

Conclusion

We have seen how to create macro variables using Symput, SymputX, and the Resolve function, how to execute macros at the Data Steps execution time using Call Execute and the Resolve function, and finally how to delete macro variables using Call Symdel and check for the existence of macro variables using SymExist, SymGlobl, and SymLocal. We reviewed the necessity to pay special attention needs to be used when using these interfaces to the macro facility as they interleave the compilation and execution times of Macro and Data Step code. I hope that by reading this paper you have found the inspiration to work with these Data Step interfaces to the Macro Facility.

References

SAS Institute Inc., Online documentation <http://support.sas.com/onlinedoc/913/docMainpage.jsp>

Thanks to: Paul St. Louis, Paul Choate, Ron Fehd, and Nat Wooding for their kind words, technical review and editing abilities.

Contact Information

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

Toby Dunn
550 Heimer Rd.
San Antonio, Tx. 78232
TobyDunn@HotMail.com

SAS® and all other SAS Institute Inc. product or service names are registered trademarks or trademarks of SAS Institute Inc. in the USA and other countries. ® indicates USA registration.