

The SAS® Data step/Macro Interface

Lawrence Heaton-Wright, Quintiles, Bracknell, Berkshire, UK

ABSTRACT

The SAS® macro facility is an extremely useful part of the SAS® System. However, macro variables and the macros themselves do not have to be exclusively referenced from within the macro facility. The SAS® data step can create, resolve and execute macros using 2 functions: SYMGET and RESOLVE and 2 routines: CALL SYMPUT and CALL EXECUTE.

This paper aims to demonstrate the usefulness and relative ease these functions and routines:

The CALL SYMPUT routine allows the creation of SAS macro variables from within a SAS data step

The CALL EXECUTE routine allows the execution of a SAS macro from within a SAS data step

The SYMGET function creates data step variables from input SAS macro variables

The RESOLVE function is similar to the SYMGET function but accepts a wider variety of arguments than SYMGET

The audience for this paper does not require a high level of technical expertise but should be familiar with the workings of the SAS macro facility. The contents are not restricted to one particular operating system.

INTRODUCTION

The SAS® macro facility is an extremely useful part of the SAS® System. However, macro variables and the macros themselves do not have to be exclusively referenced from within the macro facility.

The data step interface consists of 2 routines and 2 functions that can create, resolve and execute macros using 4 functions: CALL SYMPUT, SYMGET, RESOLVE and CALL EXECUTE.

The CALL SYMPUT routine allows the creation of macro variables from within a data step.

The CALL EXECUTE routine allows the execution of a macro from within a data step.

The RESOLVE and SYMGET functions allow the conversion of macro variables into data step variable values.

THE CALL SYMPUT ROUTINE

This routine is used by the programmer to create macro variables using either data step variable values or data set information.

The basic syntax for this routine is:

CALL SYMPUT (*macro-variable* , *value*);

Macro-variable is the name of a macro variable to be initialised with *value*. This can be a character string that is a valid SAS® name enclosed in quotes, the values of a character variable in a data step or a character expression.

Value is the value to be assigned to *macro-variable*. This can be a character string enclosed in quotes, the value of a data step variable which can be character or numeric or a data step expression.

FIXED MACRO VARIABLE CREATION

```
CALL SYMPUT ('exvar', 'testing');
```

The character string testing is assigned to the macro variable EXVAR.

CREATED FROM DATA STEP VARIABLES

```
DATA p_values;
    INPUT test : $8. label : $25.;
    CALL SYMPUT (test, label);
CARDS;
Exact Fisher's Exact
CMH Cochran-Mantel-Haenzel
;
RUN;

%PUT _USER_;
```

LOG file:

```
GLOBAL CMH Cochran-Mantel-Haenzel
GLOBAL EXACT Fisher's
```

The character string **Fisher's Exact** is assigned to the macro variable EXACT.

The character string **Cochran-Mantel-Haenzel** is assigned to the macro variable CMH.

DYNAMIC CREATION USING EXPRESSIONS

```
DATA totals;
    INPUT trt total @@;
    CALL SYMPUT ("trt" || PUT(trt,1.),
        COMPRESS("N=" || PUT(total,3.)));
CARDS;
1 100 2 120 3 85
;
RUN;
%PUT _USER_;
```

The character string **N=100** is assigned to the macro variable TRT1.

The character string **N=120** is assigned to the macro variable TRT2.

The character string **N=85** is assigned to the macro variable TRT3.

From the LOG window:

```
GLOBAL TRT1 N=100
GLOBAL TRT2 N=120
GLOBAL TRT3 N=85
```

TIMING OF MACRO VARIABLE CREATION

Remember the macro variable created using CALL SYMPUT is not available for use until the data step has executed (this is not true in all cases - see the RESOLVE function later in this paper).

```
DATA not_available;
    test='23rd Ocotber 2002';
    CALL SYMPUT ('NotAvail',test);
    retrieve="&NotAvail";
RUN;
```

LOG file:

```
1
2    DATA not_available;
3        test='23rd Ocotber 2002';
4        CALL SYMPUT ('NotAvail',test);
5        retrieve="&NotAvail";
WARNING: Apparent symbolic reference NOTAVAIL not resolved.
6    RUN;
```

FORMATTING OF MACRO VARIABLES

I would recommend using the TRIM, LEFT, COMPRESS and COMPBL functions as appropriate when creating macro variables, otherwise the macro variables created will possibly have trailing or leading blanks.

```
DATA blanks;
  LENGTH test $8.;
  test='a';
  CALL SYMPUT ('TrailBl',test);
  CALL SYMPUT ('TrimBl',TRIM(test));
RUN;

%PUT Macro variable with trailing blanks    = ***&TrailBl***;
%PUT Macro variable without trailing blanks = ***&TrimBl***;
```

Submitting this section of code produces this in the log file:

```
Macro variable with trailing blanks =***a    ***
Macro variable without trailing blanks = ***a***
```

THE CALL EXECUTE ROUTINE

A programmer can use this routine to enable execution of macros from within a data step.

Probably one of the best uses of this routine is when using the data step iteration language [DO loops, etc.] to execute a macro using the values from the iteration.

To illustrate the simple usage of this routine, the following DATA _NULL_ steps will execute this macro:

```
%MACRO PrnDSet (DataSet=report, Where=treat IS NOT MISSING);
  TITLE1 "CALL EXECUTE EXAMPLE";
  TITLE2 "DATASET=&DataSet, "
        "WHERE CLAUSE='&Where'";
  PROC PRINT DATA=&DataSet;
    WHERE &Where;
  RUN;
%MEND PrnDSet;
```

THE BASIC CALL

```
DATA _NULL_;
  SET report END=eod;
  CALL EXECUTE('%PrnDSet');
RUN;
```

This data step will call the PRNDSET macro one time for each observation in the input data set (3 times in this case).

CONDITIONAL LOGIC

Obviously this routine is far more useful when the programmer controls how many times the macro is executed. Conditional logic can be applied to the CALL EXECUTE routine as illustrated below:

```
DATA _NULL_;
  SET report END=eod;
  IF eod THEN DO;
    CALL EXECUTE('%PrnDSet');
  END;
RUN;
```

Using the IF THEN DO functionality enables the programmer to control how many times the requested macro is executed. In this case the macro PRNDSET will now only be executed once.

Referencing macro parameters within CALL EXECUTE

To further enhance the CALL EXECUTE routine, we can use data values or hard-coded values as the values of the macro parameters (keyword or positional).

Below we have 2 examples. The first shows the use of hard-coding macro parameters. The second show the usage

of data values as the macro parameters. In the second example the REPORT data set prints each treatment group separately. This is enabled through the use of the concatenation of the value of TREAT. Remember that all macro variable values are character, hence the use of the PUT statement to avoid messages in the log file about conversions from numeric to character.

```
DATA _NULL_;
  SET report END=eod;
  IF eod THEN DO;
    CALL EXECUTE('%PrnDSet
                  (DataSet=report) ');
  END;
RUN;

DATA _NULL_;
  SET report;
  CALL EXECUTE ('%PrnDSet (Where=%STR(treat=||PUT(treat,1.)|| ')) ');
RUN;
```

PRODUCING A NULL REPORT WHEN THERE ARE NO RECORDS IN A DATA SET

Check the SASHELP library for the VTABLE SAS View for no observations in a report data set.

Create a macro variable containing the number of observations in a report data set.

If number of observations equals zero then invoke the null report macro.

```
DATA _NULL_;
  SET SASHELP.vtable (WHERE=(libname='WORK' AND
                             memname='REPORT'));
  CALL SYMPUT ('Nobs', COMPRESS (PUT (Nobs, BEST.)));
RUN;

%IF %EVAL(&Nobs)=0 %THEN %DO;
  invoke null report macro
%END;
%ELSE %DO;
  invoke report macro
%END;
```

PRODUCING PATIENT NARRATIVE LISTINGS

Use CALL EXECUTE to initiate patient printouts from a selected list of patients

```
PROC SQL;
  CREATE TABLE subjid AS
    SELECT DISTINCT subjid FROM RAW.ae
    WHERE serious='Y'
    ORDER BY subjid;
QUIT;

DATA _NULL_;
  SET subjid;
  BY subjid;
  CALL EXECUTE('%PrnData
                (subjid=||PUT(subjid,4.)|| '));
RUN;
```

THE SYMGET FUNCTION

The SYMGET function can be used to retrieve macro variable values into data step variables at program run-time.

The basic syntax for the function is:

SYMGET (*argument*);

In general the argument is the name of a macro variable within quotation marks but without the ampersand (&) prefix.

CREATING DATA STEP VARIABLES FROM MACRO VARIABLE VALUES

To assign the value of the automatic SAS macro variable SYSPROCESSNAME to a data step variable, PROCESSNAME, we specify the following step:

```
DATA temp;
    ProcessName=SYMGET('SysProcessName');
RUN;
```

TEMP data set contains:

```
Obs      PROCESSNAME

1          DMS Process
```

AN ALTERNATIVE TO USING SYMGET

An alternative to using SYMGET is to use double-quotes to resolve a macro variable:

```
DATA temp;
    ProcessName="&SysProcessName";
RUN;
```

This results in the same value for PROCESSNAME but for SYMGET the length of PROCESSNAME has been defaulted to 200, whereas in the second step PROCESSNAME has the length of the value of the macro variable SYSPROCESSNAME.

AUTOMATIC ERROR-TRAPPING

If the macro variable does not exist then the data step variable will be assigned a missing value and a message will be written to the LOG window:

```
DATA temp;
    ProcessName=SYMGET('SysProcessNam');
RUN;
```

The LOG window will contain the message:

NOTE: Invalid argument to function SYMGET at line 31 column 17.

ProcessName= _ERROR_=1 _N_=1

NOTE: The data set WORK.TEMP has 1 observations and 1 variables.

NOTE: DATA statement used:

```
real time          0.01 seconds
cpu time           0.01 seconds
```

However, when using the double-quoting method a warning is sent to the LOG file rather than a note:

```
DATA temp;
    ProcessName="&SysProcessNam";
    PUT _ALL_;
RUN;
```

The LOG window will contain the message:

```
4 DATA temp;
5 ProcessName="&SysProcessNam";
WARNING: Apparent symbolic reference SYSPROCESSNAM not resolved.
6 PUT _ALL_;
7 RUN;
```

ProcessName=&SysProcessNam _ERROR_=0 _N_=1

NOTE: The data set WORK.TEMP has 1 observations and 1 variables.

NOTE: DATA statement used:

real time	0.00 seconds
cpu time	0.00 seconds

USING A CHARACTER EXPRESSION IN THE SYMGET FUNCTION

Where the SYMGET function has an advantage is when you use a character expression to construct the macro variable name.

```
%LET tot1=(n=234);
%LET tot2=(n=212);
%LET tot3=(n=245);
DATA temp;
DO i=1 TO 3;
    total=SYMGET('tot' || PUT(i,1.));
    PUT total=;
END;
RUN;
```

From the LOG window:

```
total=(n=234)
total=(n=212)
total=(n=245)
```

THE RESOLVE FUNCTION

The RESOLVE function can be used in a similar manner to the SYMGET function but also allows the resolution of macro names. The RESOLVE function also allows users to create a macro variable using the SYMPUT routine and resolve the value in the same data step.

```
DATA temp;
    invar='Timing of resolution';
    CALL SYMPUT('res_func',invar);
    resvar=RESOLVE('&res_func');
RUN;
```

The TEMP data set now contains:

Obs	invar	resvar
1	Timing of resolution	Timing of resolution

USING MACRO INVOCATION AND THE RESOLVE FUNCTION

RESOLVE can also take the value of a macro invocation (either directly or from a data step variable). The following example is taken from the SAS Macro Language manual:

```
%let event=Holiday;
%macro date;
    New Year
%mend date;

data test;
    length var1-var3 $ 15;
    when='%date';
    var1=resolve('&event'); /* macro variable reference */
    var2=resolve('%date'); /* macro invocation */
    var3=resolve(when);    /* DATA step variable with macro invocation */

    put var1= var2= var3=;
run;
```

From the LOG window:

var1=Holiday var2=New Year var3=New Year

NOTE: The data set WORK.TEST has 1 observations and 4 variables.

NOTE: DATA statement used:

real time	0.00 seconds
cpu time	0.00 seconds

OTHER MACRO INTERFACES

One of the other ways of creating a macro variable without using %LET or CALL SYMPUT is using PROC SQL. This is an extremely useful method of creating macro variables. In fact for the earlier example of accessing the SASHELP data views, I would normally use the PROC SQL methodology as this appears to be the most efficient method (both in terms of CPU and real time) of accessing these data views.

```
PROC SQL NOPRINT;
  SELECT DISTINCT PUT(nobs,BEST.)
    INTO :nobs
    FROM SASHELP.vtable
   WHERE libname="WORK" AND
          memname="REPORT";
QUIT;
```

The macro creation is carried out by the use of the INTO : clause. The NOPRINT option is useful here as it will stop the value of NOBS [from SASHELP.VTABLE] being printed to the output window.

The INTO clause can also be used to create a macro variable containing all values of a column by using the SEPARATED BY clause.

```
PROC SQL NOPRINT;
  SELECT DISTINCT subjid INTO :listsubj
    SEPARATED BY "," FROM RAW.ae;
QUIT;
```

In this case the macro variable, LISTSUBJ, would contain a unique list of subject ids from the adverse event data set delimited by a comma.

IN CONCLUSION

In summary, we can use CALL SYMPUT to create macro variables that can contain data step variable values. CALL EXECUTE can be used to invoke macros from within a data step. The SYMGET function can be used to retrieve macro variable values for use in data steps. The RESOLVE function performs with similar results to the SYMGET function but allows users to retrieve the values of executed macros or macro variables constructed within the same data step.

Overall, these functions and routines provide a powerful link between the well-known world of the data step and the power of SAS® macros.

In general the CALL SYMPUT routine is probably the most commonly used of the interfaces detailed in this paper. CALL EXECUTE is probably not used as widely as CALL SYMPUT but is an incredibly useful routine which enables the programmer to control macros from within a data step. SYMGET is probably more common than RESOLVE but in general these functions are not used as much probably because users can resolve macro variables using double-quotes.

REFERENCES

SAS® Macro Language: Reference, Version 8

Data Set used in Call Execute example programs:

```
DATA report;
    LENGTH outvar $15.;
    INPUT treat count @@;
    total=322;
    percent=(count/total)*100;
    outvar=PUT(count,3.) || "/" || PUT(total,3.) ||
        " (" || PUT(percent,2.) || "%)";
    CARDS;
1 100 2 123 3 99
;
RUN;
```

CONTACT INFORMATION

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

Lawrence Heaton-Wright
Quintiles Limited
Station House, Market Street
Bracknell, Berkshire, RG12 1HX
United Kingdom
Work Phone: +44 1344 708320
Fax: +44 1344 708106
Email: lawrence.heaton-wright@quintiles.com
Web: www.quintiles.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.

Other brand and product names are trademarks of their respective companies.