PhUSE 2015

Paper CS02

SAS® Macros Bits and Pieces

Lawrence Heaton-Wright, Quintiles

INTRODUCTION

SAS Macros are incredibly useful. They allow us as programmers to create efficient programs to perform almost any task we care to think about. They also allow us to show off a bit ... there are some really funky statements, options and macro functions out there that are really quite fun to use. This paper is a collection macro options, functions, statements and macro-related code that I've found useful over the last 20 years as a programmer.

The aim of this paper is to introduce some of the more unusual macro options, statements and functions as well as identifying why such code would be utilised in a macro. These useful bits and pieces can then be used as part of a programmer's macro toolbox.

MACRO STATEMENTS

- %GOTO AND %LABEL
- %RETURN
- %SYMDEL
- PARMBUFF AND &SYSPBUFF

MACRO FUNCTIONS

- %UPCASE
- %EVAL AND %SYSEVALF
- %SYSFUNC
- %SCAN AND %SUBSTR
- %SYMEXIST

MACRO OPTIONS

- DEBUGGING AND LOG PRINTING OPTIONS
- MINOPERATOR

AUTOCALL MACROS, MAUTOSOURCE, SASAUTOS AND MAUTOLOCDISPLAY

AUTOMATIC MACRO VARIABLES

DATA STEP/PROC SQL INTERACTIONS

- CALL SYMPUTX
- CALL EXECUTE
- PROC SQL INTO CLAUSE

%GOTO AND %LABEL

These statements are paired together within macro code. The %GOTO statement is just that ... it goes to the line in the macro identified by the label defined in the %GOTO. There are many papers and programming theories that say that a go to statement should not be used.

In many other programming languages a GOTO statement points to a specified line number which means that is any code is added between the GOTO and line number specified, the line number needs to be updated to accommodate this.

In SAS the %GOTO jumps to a specified label. I think of this construct as a one way sub routine (the same sort of functionality is available in DATA steps but with the addition of a RETURN statement). This means that the %GOTO and %LABEL combination is a neat way of controlling the early exit from a macro.

```
%MACRO macroname;
        <macro code section 1>
        %IF exit condition 1 %THEN %GOTO exit;
        <macro code section 2>
        %IF exit condition 2 %THEN %GOTO exit;
        <macro code section 3>
        %IF exit condition 3 %THEN %GOTO exit;
        <macro code section 4>
        %exit:
        <exit macro code >
%MEND macroname;
```

The label that is the target of the %GOTO statement must exist in the current macro which means that you cannot branch to a label in another macro with a %GOTO statement. Note that no percent sign is used when referencing the label in the %GOTO statement.

One of the reasons for using this code combination is to ensure that some code is executed after the label statement such as re-setting options, clearing out temporary data sets and macro variables. There can be multiple statements within a macro using a %GOTO statement all jumping to the same label.

If you have designed a macro where you check input parameters and data for the existence of certain conditions and/or variables you can check these sequentially and have %GOTO links as a consequence of a %IF clause. The other way to achieve this sort of functionality is to have a series of %IF…%END statements checking a status variable. This can be quite a tricky thing to document and maintain. A %GOTO statement is easier to understand because it does exactly what the statement says.

In the above example the macro code section 1 executes, then the 1st exit condition is checked. If this is met then the macro jumps to the %EXIT: label and executes the exit macro code. If the 1st exit condition is not met then the macro executes macro code section 2, then the 1nd exit condition is checked. If this is met then the macro jumps to the %EXIT: label. This continues until all exit conditions and macro code section shave been checked. If none of the exit conditions are met the all macro code sections (including the exit code section) are executed.

I use this functionality when designing and writing quite large macros with lots of different steps in. Used in conjunction with error or logic checks (for instance a user has entered incorrect information in a macro variable) these statements can be used to jump out of the normal sequence of macro steps and close out of the macro in a controlled manner. Using this kind of functionality allows the programmer to custom-design error messages telling the user what has caused the macro to be stopped.

%RETURN

This statement causes the normal termination of the currently executing macro. This can be used to stop a macro executing when the %GOTO/%LABEL combination cannot be used (i.e. trying to exit out of a sub-macro to a label in the parent macro). This is a very useful option when you just want to stop the macro executing if a condition is met and we're not interested in performing any clean up functionality (although you can always use a %LET to change the value of a flag variable).

There is also the %ABORT statement but I have found that %GOTO and %RETURN cater for all the situations where I need to terminate a SAS macro early.

```
%MACRO return_macro;
    <macro code>
    %IF exit condition %THEN %RETURN;
    <more macro code>
%MEND return_macro;
```

%SYMDEL

This macro statement allows you to delete a macro variable from the global macro symbol table. This statement is especially useful if you want to clean up a SAS session where you use multiple programs and you don't want macro variables to retain values across programs.

```
%SYMDEL variable_name … more macro variables as required;
```

PARMBUFF AND &SYSPBUFF

Adding the PARMBUFF option to the %MACRO statement will populate the SYSPBUFF automatic macro variable with the values of the parameters supplied to the SAS macro. This is potentially very useful if you have strict naming conventions around SAS macro variables. For instance if a variable starts with "YN" then you know it should only contain values YN, variables starting "DT" could signify dates. By using the PARMBUFF option, the values of the input parameters can be scanned through and checked. One of the useful things is that SYSPBUFF only contains the values of parameters in the macro invocation call. So if you have default values attached to the macro variables in the %MACRO statement and these are not changed in the invocation call, then these are not sent to SYSPBUFF.

```
%MACRO testpbuff (Var=, Check=default) / PARMBUFF;

  %PUT SYSPBUFF = &SysPBuff.;

%MEND testpbuff;

%testpbuff (Var=variable);
```

The LOG contains: SYSPBUFF = (Var=variable)

```
%testpbuff (Var=variable, check=change);
```

The LOG contains: SYSPBUFF = (Var=variable, check=change)

One of the uses of SYSPBUFF is to ensure that if a variable is changed from the default value that you intend it to be, you can check that value is one of a required list of options.

%UPCASE

This macro function does exactly what you think it does. It converts values to uppercase. This is incredibly useful when you're checking the values of macro variables in an IF statement (such as data checking).
I frequently set all of my input macro variables to uppercase so that any possible combinations of lower and upper-case letters are eliminated. For instance I might have a macro variable which I intend to only be Y or N. The user could enter Y, y, N or n. I can immediately reduce the combinations to check in half by uppercasing the macro variable.

```
%LET ynVar=%UPCASE(&ynVar);
```

I can then combine this with the IN operator (detailed in the MINOPERATOR section) to ensure that only the intended values are entered in the macro variable:

```
%IF &ynVar IN (N Y) %THEN %DO; <more macro code> %END;
```

If the variable was not upper-cased then the SAS code would be:

```
%IF &ynVar IN (N n Y y) %THEN %DO; <more macro code> %END;
```

If there are more options or longer text strings then it could get quite congested within the IN statement.

You can also use this function as part of a statement without changing the macro variable:

```
%IF %UPCASE(&SysProcessName) NE DMS PROCESS %THEN %DO;
```

This means that I am typing less text in my macro and giving the program less chance to fall over through spelling mistakes or typos.

%EVAL AND %SYSEVALF

As all macro variables are defined as character variables by SAS (including numeric values), if we need to perform some form of mathematical operation on macro variables these 2 functions allow us to evaluate macro variables as numeric values rather than character values.
%EVAL allows you to evaluate integer expressions as numeric values.

```
%IF %EVAL(&x) LT 15 %THEN %PUT INTEGER LESS THAN 15;

%ELSE %IF %EVAL(&x) GE 15 %THEN %PUT INTEGER 15 OR ABOVE;
```

%SYSEVALF allows you to evaluate non-integer expressions.

```
%LET x=15; %LET Y=24.2;

%LET z=%EVAL(&x+&y);
```

In this example the SAS LOG file results in the following message:

ERROR: A character operand was found in the %EVAL function or %IF condition where a numeric operand is required. The condition was: 15+24.2

However, if we use SYSEVALF like:

```
%LET z=%SYSEVALF(&x+&y);
```

We get the following:

```
x=15 y=24.2 z=39.2
```

%SYSFUNC

This macro function allows you to utilise data step functions within macro code. There are some functions you are not allowed to use. It is slightly vexing not to be able to use PUT and INPUT but, fortunately we can use INPUTN, INPUTC, PUTN, and PUTC.

One of the main uses I've found for %SYSFUNC is to utilise functions such as OPEN, CLOSE, ATTRN, ATTRC to perform tasks such as identifying the number of records in a data set or whether a variable exists within a data set.

To obtain the number of observations in a data set, this code is extremely efficient as no extra data sets or procedures are used.

```
%LET dsid=%SYSFUNC(OPEN(WORK.DATASET));

%LET nobs=%SYSFUNC(ATTRN(&dsid.,NOBS));

%LET rc=%SYSFUNC(CLOSE(&dsid.));
```

Another use for %SYSFUNC is to enable the use of the GETOPTION function to retrieve the setting of a SAS system option into a macro variable. This is especially useful if you want to turn off certain options (SOURCE, SOURCE2, MPRINT, MLOGIC, etc.) during the course of your macro but then to reset them once your macro has completed execution.

%SCAN AND %SUBSTR

These macro functions operate in a similar manner to their data step equivalents.

%SCAN has loads of different modifiers and options that can be used but, typically, I use it as the macro version of the SCAN function. I commonly use it for scanning through directory path and filenames searching for text strings such as folder names.

```
%LET progname=%SCAN(c:\temporary\program.sas,-1,\);
```

Results in PROGNAME containing “program.sas”

%SUBSTR is used virtually identically to its data step counterpart.

```
%LET temp=%SUBSTR(c:\temporary\program.sas,4,9);
```

Results in TEMP containing “temporary”

%SYMEXIST

%SYMEXIST checks whether a macro variable exists (i.e. is available for processing). This can be a very useful function if you are checking whether a macro variable has a certain value but if the variable does not exist then an error will be generated because you are trying to compare a value to a macro variable that doesn’t exist. The SYMEXIST function can be used to ensure that the macro variable exists, then a check can be made of the actual value.

```
%IF %SYMEXIST(progname) %THEN %DO; <macro code using &progname>; %END;
```

SYSMEXIST returns a value of 1 if the macro variable exists. It returns a value of 0 if not.

DEBUGGING AND LOG PRINTING OPTIONS

There are many options to aid the programmer in de-bugging macros. However, be warned as they are extremely verbose ... in other words, use these options with care unless you want potentially very large LOG files. When developing and de-bugging programs it is worth having some (or all) of these options turned on. However, once into production it is worthwhile turning these options off as they generate tremendous amounts of data in the LOG file.

- MPRINT displays the SAS statements that are generated by the macro execution.
- MPRINTNEST prints nesting information to be displayed in the MPRINT log output. This is useful when macros are called from within macros, etc. so that you can trace what macro is currently executing and also what macro (if any) it was invoked from.
- MLOGIC shows the trace of the macros execution. This generates a LOT of output but can be really useful in identifying logical issues within a macro.
- MLOGICNEST does the same thing as MPRINTNEST but for MLOGIC.
- SYMBOLGEN displays the results of resolving macro variables.
- SOURCE/SOURCE2 – these options are not macro LOG options. They apply to data step/procedure code (SOURCE2 is from included files). If these are turned off then the conventional LOG messages showing DATA step/PROC resolution if turned off. This can be very useful when it comes to macro suites where you want to control messages printed to the LOG.

Top tip: use a keyboard macro file to add these options or turn them off.

MINOPERATOR

This can be used either in an OPTIONS statement or as an option to the %MACRO statement. The default system option is NOMINOPERATOR which means that if you want to have it available across all your macros you need to change this global setting. An alternative option is to add it after the macro name as part of the %MACRO statement.

OPTIONS MINOPERATOR; or %MACRO macroname / MINOPERATOR;

This option allows the use of the macro IN operator in an expression (similar to the DATA step IN operator). This option was introduced in version 9 and allows macros to be programmed more efficiently. Prior to this if I wanted to check whether a macro variable contained several values then the %IF statement would need to be constructed like:

```
%IF &Var. EQ A OR &Var. EQ B OR &Var. EQ C %THEN %DO;
```

I think we can all agree that this is pretty frustrating and not very efficient in terms of maintenance and ease of understanding. With the MINOPERATOR option this code can be changed to

```
%IF &Var. IN (A B C) %THEN %DO;
```

This code is much easier to understand, it's more elegant and maintain such code as it's familiar to the well-known DATA step code.

AUTOCALL MACROS, MAUTOSOURCE, SASAUTOS AND MAUTOLOCDISPLAY

SAS supplies a number of autocall macros which act like macro functions but are technically macros.

%CMPRES and %QCMPRES
%COMPSTOR
%DATATYP
%KVERIFY and %VERIFY
%LEFT and %QLEFT
%LOWCASE and %QLOWCASE
%SYSRC
%TRIM and %QTRIM

These macros, by default, are available because the system option MAUTOSOURCE is turned on (indicating that autocall macros can be used) and the SASAUTOS option points to the SASAUTOS fileref (which is automatically generated when SAS is installed).

If you change the SASAUTOS option then SAS no longer knows where to locate its own autocall macros unless you include the SASAUTOS fileref as part of the list of filerefs where SAS macros are stored.

In general, I have found it to be advantageous to ensure that these options are included in a project set-up file like this:

```
OPTIONS MAUTOSOURCE SASAUTOS = (SASAUTOS PROJMAC GLOBMAC);
```

Where PROJMAC and GLOBMAC are filerefs defined by the project set up file where user-defined macros are stored. SAS will then search through SASAUTOS first, then PROJMAC and then GLOBMAC for the macro definition. If it can't find the macro definition then an error will result.

MAUTOLOCDISPLAY displays the source location of the autocall macro. This is incredibly useful if you're having to debug a program which uses a project-specific version of a global macro. Having the location of where the autocall macro is executed from is vital information.

Typically a LOG file will contain a message like this when the MAUTOLOCDISPLAY is invoked:

MAUTOLOCDISPLAY(CMPRES): This macro was compiled from the autocall file
D:\SAS94\x86\SASFoundation\9.4\core\sasmacro\cmpres.sas

Unfortunately there doesn't seem to be a similar option for macros invoked from a compiled macro catalog.

AUTOMATIC MACRO VARIABLES

These are macro variables that are initialised every time SAS is invoked. There are approximately 30 different variables created. These contain information about the SAS session such as the date and time and operating system information. These can be referenced as any SAS macro variable. For a full list see the SAS help guide. A useful method for displaying the variables and their values is using %PUT _AUTOMATIC_; which will show the variable name and the associated value.

Some of the most useful automatic macros variables are:

Variable Name	Contents
SYSDATE	Date that SAS job/session began executing in DATE7 format
SYSDATE9	Date that SAS job/session began executing in DATE9 format
SYSTIME	Time that SAS job/session began executing in TIME5 format
SYSPROCESSNAME	Name of the current SAS job/session Interactive SAS is defined with SYSPROCESSNAME=DMS Process Batch SAS jobs show the file and pathname of the SAS job
SYSENV	Is the SAS job/session interactive? SYSENV=FORE for interactive session, SYSENV=BACK for batch mode
SYSUSERID	User or current login ID of the current SAS job/session
SYSVER	Contains the SAS version used for the current SAS job/session. SYSVERLONG and SYSVERLONG4 contain similar information but with the addition of the maintenance release information.
SYSSCP and SYSSCPL	Contains the Operating System information for the current SAS job/session.

These automatic macros variables can be utilised in sending information about the SAS job/session to every LOG file created using a set-up/autoexec file. There are several automatic macro variables that contain return codes and information about the DATA step/procedure that has just executed. For example the SYSLIBRC macro variable contains a return code as to whether the last LIBNAME statement executed correctly (i.e. does the library referenced actually exist?).

CALL SYMPUTX

The CALL SYMPUT routine has been used for years to create macro variables within a DATA step. In version 9 CALL SYMPUTX was introduced which introduced many upgrades.

One is that the user can specify whether the macro variable is defined in the local or global macro variable table. This is an incredibly useful option. Previously CALL SYMPUT created macro variables in the most local symbol table. With CALL SYMPUTX being able to define the scope of the macro variable is incredibly useful as if we need to create global macro variables within a sub-macro we now don't need to use a %GLOBAL statement. This is especially useful if you're creating data-driven macro variables (footnotes for example).

Numeric to character conversion is automatically applied resulting in less code (no need for COMPRESS and PUT functions). The resulting character variable automatically has leading and trailing spaces removed.

This routine was introduced in v9 and is much, much better than plain old CALL SYMPUT.

CALL EXECUTE

This brilliant data step call routine allows you to execute macros from inside a data step. The macro that is called can have multiple procedures and data steps which makes this an extremely powerful routine. One thing to be aware of:

> You cannot create macro variables during the execution of the macro called by CALL EXECUTE. This means that if your macro has %LET or CALL SYMPUT(X) or PROC SQL INTO: macro variable creation statements then the program will fall over due to this restriction.

Quite often I use a CALL EXECUTE macro call when I need to iterate through several steps utilising different parameters/subjects.

```
%MACRO ExampleExecute (usubjid=);
   DATA _selected_data;
     SET adam.advs (WHERE=(usubjid EQ "&usubjid."));
   RUN;

   **** MORE DATA STEPS/PROCEDURES ***;

%MEND ExampleExecute;

DATA _NULL_;
  SET adam.adsl (WHERE=(saffl eq 'Y'));
  CALL EXECUTE (CATT('%ExampleExecute (usubjid=',usubjid,');'));
RUN;
```

In the example above the versatility of the CALL EXECUTE step can be seen. I am repeating the same steps for each subject (in this case subjects from the from the safety population) for each subject from the vital signs data set. This kind of data step/macro interface is incredibly useful for producing patient profiles and/or narratives as you can control the subjects going into the macro by selecting a data set. However, if you need to create macro variables within the course of your macro then CALL EXECUTE will not serve your purposes.

Generally, in these circumstances, I'll create a control data set which contains the unique parameter. An index/identification variable is created as well as defining the number of records in this data step. This information will then be used to control the number of iterations through the macro.

```
DATA _control;
  SET adam.adsl (WHERE=(saffl eq 'Y')) END=eof;
  id=_N_;
  IF eof THEN CALL SYMPUTX('N_Subj',_N_,'G');
RUN;

%MACRO Example;
  %DO i=1 %TO &N_Subj.;
    DATA _NULL_;
      SET _control (WHERE=(id EQ &i.));
      CALL SYMPUTX ('Usubjid',usubjid,'L');
    RUN;

    DATA _selected_data;
      SET adam.advs (WHERE=(usubjid EQ "&usubjid."));
    RUN;

    **** MORE DATA STEPS/PROCEDURES ***;
  %END;
%MEND Example;
%Example;
```

PROC SQL INTO CLAUSE

This is an alternative method of creating macro variables to using a DATA step and CALL SYMPUTX. Where this method really scores highly is the ability to produce a combined list of values in one macro variable. Typically I'll use it when I want to identify a list of subject numbers for investigation and I want to quickly print out the data for these subjects without having to subset the data using merges (or other methods).

```
PROC SQL NOPRINT;
  SELECT DISTINCT(QUOTE(usubjid)) INTO: subjects SEPARATED BY ','
    FROM adsl
    WHERE trtsdt LT "01APR2014"D
  ;
QUIT;
```

This will produce a macro variable which contains a list of subjects that satisfy the WHERE clause:

```
"ABC101001","ABC102004","ABC104001"
```

Breaking down the PROC SQL statement we have:

- The NOPRINT option ensures that the list of subjects selected are not displayed the output destination
- The SELECT DISTINCT statement ensures that only unique values of USUBJID are selected
- The QUOTE function ensures that the selected values of the variable are quoted
    - in this case USUBJID is character so we need quotes to ensure any WHERE clause this macro variable is used in will not produce an error
- The INTO: statement tells the SQL processor to send the results to the specified macro variable (in this case SUBJECTS)
- The SEPARATED BY clause tells the SQL processor to send all the results into one macro variable separated by the specified character(s)
- The FROM statement is the source data
- The WHERE clause is how we determine what data to select

The macro variable produced can then be used like:

```
PROC PRINT DATA=dataset WIDTH=MIN HEADING=H;
  WHERE usubjid IN (&subjects.);
RUN;
```

CONCLUSION

There is no specific conclusion to take away from this paper. However, I hope to have shown how wide-ranging and versatile SAS macro programming is and that there will be enough hints and tips for you to take away from this paper and experiment with and expand your macro programming knowledge.

CONTACT INFORMATION

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

Lawrence Heaton-Wright
Quintiles
500 Brook Drive,
Green Park,
Reading,
Berkshire.
RG2 6UU
United Kingdom

Work Phone: + 44 118 4508320
Email: lawrence.heaton-wright@quintiles.com
Web: www.quintiles.com