PhUSE EU Connect 2018

Paper CT01

Getting organized for SAS Macro development: hints, tips and tools for a structured approach

Iain Humphreys, PRA Health Sciences, Reading, UK

ABSTRACT

The SAS® Macro language is widely used in the handling and reporting of clinical data, often forming part of a system or application intended to automate or streamline some of the more common programming purposes for which SAS software is used.

During macro development, certain tasks repeatedly occur and for reasons of efficiency, consistency and good programming practice it is useful to have a small library of utilities to perform these tasks.

Aimed at programmers perhaps just embarking on software development using macros, this paper explores some of the common macro programming tasks (with examples of code); it also provides hints and tips for taking a structured approach to macro development, regardless of the scale of the application.

Setting out with some simple tools in place, in conjunction with a structured approach, will aid in achieving consistency, efficiency and hopefully will lead to well-structured and maintainable code.

INTRODUCTION

In the clinical arena, the SAS Macro language often plays a prominent (though not necessarily exclusive) role in the development of applications: systems intended to automate or streamline many of the typical data handling and reporting purposes for which the SAS Base language is routinely employed. Such systems will be used more frequently, by a wider variety of users, and applied to a wider range of situations, than many of the individual SAS programs written to perform study-specific tasks. Although there is a strong case for a structured approach and good programming practices to be followed regardless of the type or scope of a program, it is especially critical in the case of applications or utilities being developed for wider and more general use. It becomes quickly apparent to developers that certain tasks are performed frequently and for these it is useful to have some standard programming constructs or utilities in place. Taking a modular approach to programming has advantages:

- Repetitive tasks can be accomplished consistently and efficiently, without repeating (sometimes lengthy) code
- Modules can be separately and independently validated, making easier the task of overall application validation

This paper describes some common programming tasks that lend themselves to a standard approach and in some cases to encapsulation within a utility module. In addition to taking a standard approach to common programmatic tasks, attention should be paid to macro design. The paper will outline some good programming practices that will help to achieve consistency and robustness.

This paper is aimed at SAS programmers with a basic understanding of the macro language and how it interacts with the SAS Base product, and who are perhaps beginning their first application development project. Hopefully it will equip them with some simple code examples for commonly occurring tasks, and some good programming practices to get them organized and help them on their way.

NOTE: examples of code within this paper are mostly for illustration of a specific concept. For reasons of brevity, features such as parameter validation and error handling have sometimes been omitted; these would be present in a real solution.

GOOD PROGRAMMING PRACTICE FOR MACRO DEVELOPMENT

Several techniques for developing robust macros are now well established and described [1]. These include the following:

- Parameter validation
- Error handling
- Keeping the environment tidy

It therefore follows that having some utilities to hand which help us adhere to good practice would be beneficial.

PARAMETER VALIDATION

One of the most important recommendations is to validate the parameters passed to the macro, in order that missing or invalid values are detected and handled ‘cleanly’, i.e. preventing an error being generated when the code resulting from the macro is executed. As well as basic testing for the presence or absence of a parameter value, it is also

sometimes necessary to verify the value. Frequently parameters refer to items in SAS libraries, so being able to access SAS library metadata is useful. Parameter validation often represents a significant proportion of a macro's code volume and having a small library of utilities available, that are also easy to call, allows the developer to quickly build such checks and deal with parameter validation efficiently.

Examples of common parameter-checking tasks include:

- Checking for a non-missing value of a parameter
- Verifying its existence, if the parameter is one of the following
    - A SAS dataset
    - A variable within a SAS dataset
    - A file or directory name
    - A file or library reference
- Verifying against a list of permissible values

A check to establish whether a parameter is missing can be achieved by the g_util_chkParamBlank utility macro:

```
%macro g_util_chkParamBlank(param) ;
    %sysevalf(%superq(param)=,boolean)
%mend g_util_chkParamBlank ;
```

In general, it is preferable to write SAS macros with named (keyword) parameters, since these are more easily understood by users calling the macro. However, for small utilities such as this, intended for use by the application developer rather than the end user, a function-style macro with a single positional parameter is easier to call. The macro returns a 1 if the parameter is blank, or a 0 otherwise. For example, to check that a value for macro parameter TMT_VAR (treatment variable) has been supplied, the developer would simply code a statement within the calling macro such as:

```
%if %g_util_chkParamBlank(&tmt_var) %then %do ;
    * << statements taking appropriate action for missing parameter >> ;
%end ;
%else %do ;
    * << statements taking appropriate action for non-missing parameter >> ;
%end ;
```

Similar checks can be applied to any parameter for which a value is required. Having established that a parameter has been populated, the developer can then proceed to check the validity of the value. For the purposes of verifying the existence of datasets, variables and other entities, there are several ways of accessing the metadata held in SAS dictionaries [2]. One useful technique for verifying the presence of a SAS library item is to make use of the SAS file handling functions. Here is an example of a simple utility that uses such functions to check that a variable exists within in a dataset:

```
%macro g_util_chkVarExist(ds=, var=) ;
  %local _dsid _varexist _rc _vxist ;
  %let _varexist = 0 ;

  %let _dsid = %sysfunc(open(&ds));
  %if &_dsid > 0 %then %do ;
      %let _vxist=%sysfunc(varnum(&_dsid, &var)) ;
      %if &_vxist > 0 %then %let _varexist = 1 ;
      %let _rc = %sysfunc(close(&_dsid)) ;
  %end ;
  &_varexist
%mend g_util_chkVarExist ;
```

In this instance, the OPEN function is used to open the dataset of interest, and the VARNUM function called to return the number of the variable of interest within this set (a zero is returned if the variable does not exist). Again, a simple function-style utility has been created, returning a 1 if the variable exists, or 0 if it does not. Having already established that the TMT_VAR parameter is populated, the developer can thus code another simple check:

```
%if not %g_util_chkVarExist(ds=&_ADaM..&data, var=&tmt_var) %then %do ;
   * << statements taking appropriate action for a variable not found >> ;
%end ;
```

Lastly, a simple check of a parameter's setting against a finite list of permissible values can easily be achieved with a utility such as this:

```
%macro g_util_chkValidStat (_stat) ;
  %local __master __found __i;
  %let __master=N|MEAN|STD|MEDIAN|MIN|MAX ;
  %let __found = 0 ;
  %let __i=1 ;
  %do %while(%scan(&__master, &__i, |) NE %str()) ;
      %if %upcase(&_stat) eq %scan(&__master, &__i, |) %then %let __found = 1 ;
      %let __i = %eval(&__i+1) ;
  %end ;
  &__found
%mend g_util_chkValidStat ;
```

In this example, the macro defines a local macro variable __MASTER to hold a list of valid statistic names and initializes a variable __FOUND to contain the result of the search. The macro receives the statistic to be checked as its single parameter, and then proceeds to compare it with each item in the list of valid names. If the statistic name is found on the list the macro returns a 1; otherwise a 0. Again, the developer can subsequently use a simple function-style call to test whether a statistic name is valid:

```
%if not %g_util_chkValidStat(&stat) %then %do ;
   * << statements taking appropriate action for an invalid statistic name >> ;
%end ;
```

ERROR HANDLING

Parameter validation and other macro checks that fail should be handled appropriately by the macro. Since one of the purposes of such checks is to prevent a SAS error or other unwanted outcome when the code produced by the macro executes, it is often desirable to do two things:

- Terminate the macro before the error or unwanted outcome can occur
- Notify the user of the outcome

In this example, a blank parameter is detected; the end user of the macro is notified via a message to the log that an invalid call has been attempted, a return code is set, and the macro terminates without further processing.

```
%if %g_util_chkParamBlank(&tmt_var) %then %do ;
    %put %STR(W)ARNING: MUST SUPPLY VALUE FOR TMT_VAR PARAMETER - EXITING MACRO ;
    %let &rc = 4 ;
    %goto exit ;
%end ;
```

The warning message is written to the SAS log to alert the user that there is a problem. It is common for a return code of zero to represent a successful completion, with various error conditions represented by non-zero codes. Furthermore, it is considered good practice for each macro to have its own individual return code. In this instance the code represented by &RC will have already been declared as a global macro symbol and given a unique name. Declaring the return code as a global variable makes it accessible outside of the macro in which it was defined. In this way inter-dependence can be established between separate macros: a macro's return code can be tested and, if unsuccessful, can prevent subsequent processing from taking place. This is useful when several lower-level macros are called from within a higher-level macro. Finally, the %GOTO statement prevents the macro's normal process flow being followed, by signifying a jump to an endpoint labelled EXIT (i.e. a position within the macro code designated by the label '%EXIT:'). This area of code usually performs some 'cleaning up' prior to termination of the macro.

In addition to detecting and handling invalid or missing values passed as parameters, the macro may need to deal with errors or warnings issued by SAS on execution of the code generated by the macro once all macro references are resolved. To detect errors and warnings from SAS DATA and PROC steps and global statements, the automatic global variable &SYSERR can be tested: a value of zero indicates successful execution; non-zero values indicate a problem. In this way, critical PROC or DATA steps can be checked, and the macro terminated early to prevent further unnecessary processing.

KEEPING THINGS TIDY: INITIALIZING AND RESETTING THE ENVIRONMENT

To prevent any unwanted effect from previous processing in the same SAS session, it is usually desirable to initialize the programming environment, prior to running (or re-running) the application. Typically, this is done by:

- Removing unwanted temporary (work) datasets

- Re-setting options, as needed
- Deleting or resetting certain global macro variables.

This would typically be done at the start of an application's primary macro (i.e. the one called by the end user).

Similarly, and in order not to interfere with any subsequent process that may run in the same SAS session, it is good practice to restore the environment to a clean state at the end of processing. Commonly this is achieved by designating the last part of a macro program to this function, and giving this section of code a label; the section is either reached naturally by the successful execution of all previous steps in the macro, or it could be 'jumped to' by a %GOTO statement triggered by a failing check such as those discussed above.

Actions that might be taken within such a section of code will include:

- Deleting temporary datasets
- Resetting any options that were altered by the actions of the macro
- Releasing any FILENAMES or LIBNAMES issued by the macro
- Issuing a final message and return code to the user

Such a code section would probably exist in the primary macro but also in any other sub-macros called by the primary, as required.

Some examples of utilities for performing clean-up tasks follow. For the purposes of removing work datasets a small macro such as this is useful:

```
%macro g_util_deleteds(datasets) ;
  %if %upcase(&datasets) = _ALL_ %then %do ;
    %* --- CLEAR ALL DATASET(S) FROM WORK LIBRARY --- ;
    proc datasets library=work memtype=data nolist kill ;
    run ; quit ;
  %end ;
  %else %do ;
    %* --- CLEAR NAMED DATASET(S) FROM WORK LIBRARY --- ;
    proc datasets library=work memtype=data nolist nowarn ;
      delete &datasets ;
    run ; quit ;
  %end ;
%mend g_util_deleteds ;
```

This simple utility macro permits the deletion of all temporary datasets, by specifying the keyword _ALL_ in the call to the macro; this might be typically done at the time of initialization. Alternatively, the developer can name one or more specific work datasets for deletion; this might be desirable at other times during the application's execution, for example if some other datasets need to be kept for subsequent processing by the application.

Global symbols are often established by developers to pass information between various macros, and also to the SAS syntax yielded by macro resolution. Because of their importance in controlling often complex processing, it is important for macro developers to maintain a list of all such symbols, and to ensure they are reset at the time of initialization to ensure there are no unwanted settings left from any previous processing within the same SAS session. It may also be appropriate to remove certain symbols on termination of the macro. Again, a simple utility macro helps the developer to keep on top of this:

```
%macro g_util_symReset ;
  %local __i ;

  %* --- Delete individual symbols --- * ;
  %symdel
    _listings
    _figures
    _tables
  / nowarn ;

  %* --- Delete a series of symbols --- * ;
  %if %symglobl(_n_tablecols) %then %do __i = 1 %to &_n_tablecols ;
      %symdel
        _tablecol_&__i
      / nowarn ;
  %end;
```

```
  %symdel _n_tablecols / nowarn ;
%mend g_util_symReset ;
```

The developer maintains a list of all global macro variables used across the entire suite of macros that make up an application and includes them here for deletion. Symbols can be individually named in a %SYMDEL statement; alternatively, the presence of a symbol can be tested with %SYMGLOBL. This is useful where a series of symbols (e.g. _tablecol_1, _tablecol_2, … _tablecol_*n*) might exist, and the number *n* is given by a further symbol (e.g. _n_tablecols). A loop can then be established to delete all the symbols in the series.

OTHER FREQUENTLY OCCURRING MACRO PROGRAMMING TASKS

This section considers some more general programming tasks that lend themselves to having a utility macro to hand:

- Counting observations in a dataset
- Determining a variable's type
- Building a list of items
- Counting the number of items in a list

DETERMINING THE NUMBER OF OBSERVATIONS IN A DATASET

Often the developer needs to know how many observations are present in a dataset. Depending on the result, the main macro can then take different actions. For example, if a dataset contains observations, these might be reported; however, if the set is empty, a 'place holder' might be produced instead - essentially a blank table with a message to state that there are no records to report. The following is an example of a utility to perform such a test:

```
%macro g_util_obs(dsnam) ;
   %local __nobs __dsid __rc ;
   %let __nobs = -1 ;

   %let __dsid = %sysfunc(open(&dsnam)) ;

   %if &__dsid > 0 %then %do ;
       %let __nobs = %sysfunc(attrn(&__dsid, nobs)) ;
       %let __rc   = %sysfunc(close(&__dsid)) ;
   %end ;
   %else %do ;
       %put NOTE: Unable to open &dsnam ;
   %end;

   &__nobs
 %mend g_util_obs;
```

Again, a function-style macro is defined with a single positional parameter for the dataset of interest. The macro returns the number of observations in the dataset (zero or a positive integer), or a value of -1 if the dataset is not found. This makes it easy for the developer to call the macro, perhaps conditionally, as shown here:

```
%if %g_util_obs(&inds) GT 0 %do ;
    * << statements taking appropriate action when observations are present >> ;
%end ;
%else %do ;
    * << statements taking appropriate action when no observations are found >> ;
%end ;
```

The utility need not necessarily be called from another macro; it could also be referenced within a SAS DATA step:

```
data _null_ ;
  x = %g_util_obs(ae5) ;
  if x ge 0 then do ;
```

DETERMINING WHETHER A VARIABLE IS CHARACTER OR NUMERIC

Sometimes it is necessary to check that a variable of appropriate type has been referenced. An example would be checking that an analysis variable is numeric, if the naming of the variable is under the control of the end user. Or perhaps when working with a pair of variables there might be a requirement that they are both of the same type. The

following utility uses the SAS file functions to determine the type of the variable passed as a parameter along with its dataset name.

```
%macro g_util_vtype(dset, var) ;

     %local __dsid __vno __vtype __vfound __rc ;
     %let __vfound = 0 ;

     %let __dsid  = %sysfunc(open(&dset)) ;
     %if &__dsid > 0 %then %do ;
         %let __vno  = %sysfunc(varnum(&__dsid, &var)) ;
         %if &__vno > 0 %then %do ;
              %let __vfound = 1 ;
              %let __vtype = %sysfunc(vartype(&__dsid, &__vno)) ;
         %end ;

         %let __rc = %sysfunc(close(&__dsid));

         %if &__vfound eq 1 %then %do ;
             &__vtype
         %end ;
         %else %do ;
             %put NOTE: Variable &var does not exist in dataset &dset ;
         %end ;
     %end ;
     %else %do ;
         %put NOTE: Dataset &dset not found ;
     %end ;

%mend g_util_vtype ;
```

The macro can be called in ‘function style’, and, if the variable exists, its type is returned as N or C:

```
%if %g_util_vtype(&inds, &avar) EQ C %then %do ;
    * << statements for appropriate action when variable type is character >> ;
%end ;
%else %do ;
    * << statements for appropriate action when variable type is not character >> ;
%end ;
```

BUILDING A LIST OF ITEMS

This is a frequent macro programming task: to build a list of items (e.g. dataset names, variable names, parameter names), and then to work through the list, processing each item in turn. A useful means of achieving this, amongst others, is to use PROC SQL, as shown in the next example.

Consider building a list of laboratory parameter codes, such that this can be used to drive some graphical reporting, with a separate plot for each parameter. A simple PROC SQL step querying the relevant analysis dataset and using the INTO :<*symbol-name*> notation can achieve this. Here we use the keyword DISTINCT to build a unique list of all biochemistry parameters, delimited by the pipe symbol, and stored in a macro symbol (_PCODES).

```
proc sql noprint ;
    select distinct paramcd into :_pcodes separated by '|'
    from derived.adlb
    where upcase(parcat1) eq 'BIOCHEMISTRY';
quit;
%put &_pcodes ;
```

The resulting SAS log will display the list created, for example:

```
ALP|ALT|AST|BILI|CA|CHOL|CREAT|ECREATCR|HDL|K|LDL|SODIUM
```

Note: if this step is coded within a macro, and the resulting macro symbol _PCODES is required outside of the macro, then it should be declared as global. Similar steps can be employed to query the SAS data dictionary tables,

for example to build a list of all datasets in a library containing date variables, or all variables in a dataset that are character.

PROCESSING ITEMS IN A LIST

Having built, or otherwise obtained a list of items (e.g. perhaps passed as a macro parameter), there is often a need to parse it: in other words, to work through it, isolating each item in turn and performing some process with it. In order to do this, it is useful to know how many items are present. If a list has been created as shown above with PROC SQL, this is easy since the number of items returned by the query will be held in the automatic SAS macro symbol &SQLOBS. But for other occasions it is useful to have a small function-style utility to count the items in a list. This example assumes the pipe character (|) has been used to delimit the list, but permits the user to override this by specifying another character in the 2nd (DELIM=) parameter; for space-delimited lists, this should be set to %str( ).

```
%macro g_util_itemcount(
          list  =
         ,delim = |
      ) ;
    %local __i __items ;
    %let __i = 1 ;
    %let __item&__i = %qscan(&list, &__i, &delim) ;

    %do %while (%length(&&__item&__i) GT 0) ;
        %let __i = %eval(&__i + 1) ;
        %let __item&__i = %qscan(&list, &__i, &delim) ;
    %end ;
    %let __items = %eval(&__i - 1) ;
    &__items
%mend g_util_itemcount ;
```

The utility scans the item list passed as a parameter, incrementing a counter with each new item encountered, as determined by the delimiting character. Note: this is a simplified version of the utility, shown here for illustrative purposes and in the interest of brevity. Whereas it will function correctly for lists that contain no blank values, the extended version permits the designation of null items in a list, with the option to count either all items or non-null items only. Now the developer can employ this in processing the list. Continuing the laboratory reporting example above, the program can loop through the list of parameter codes, e.g. to call a plotting macro for each:

```
%do __i = 1 %to %g_util_itemcount(list=&_pcodes) ;
    %local __item&__i ;
    %let __item&__i = %qscan(&_pcodes, &__i, |) ;
    %put Processing parameter &__i &&__item&__i ;
    * << statements for plotting a parameter >> ;
%end ;
```

The resulting SAS log will display, e.g.:

```
Processing parameter 1 ALP
Processing parameter 2 ALT
...
```

FURTHER CONSIDERATIONS FOR MACRO DEVELOPERS

In addition to some general utilities and coding techniques for common programmatic tasks, it is often beneficial for developers to consider some other, broader aspects; these are discussed briefly below.

NAMING CONVENTIONS

In the typical organization, there may be many macros, developed by many programmers. Some may be deployed locally for a specific purpose; others may be made globally available, and ownership may be transferred from one function to another. Devising a convention for the naming of macros, their parameters and for global symbols can help make a macro's purpose and scope easier to understand. For example, global macros might be given a prefix 'g_'; utility macros might contain '_util' in their name. A standard parameter name for an input dataset might be 'inds', and so forth.

DOCUMENTATION

This greatly assists the maintenance of a macro, particularly by a programmer other than the original developer, and is essential for formal validation (see below). A specification document is recommended for each macro which should describe, at a minimum: a brief statement of its purpose, a description of its function and logic flow; its

parameters and any default values; any parameter validation that is carried out; any return codes that are issued and their meaning; any dependencies (e.g. pre-requisite global symbols), any higher-level programs or macros that might call it; any lower-level macros called by it; any constraints in its usage.

It is also recommended that a list is maintained of all the global macro symbols used in an application; keeping track of these helps avoid clashes and ensures inclusion in relevant initialization processes. If a suite of macros form part of a larger application, this should be accompanied by a User Guide describing the functionality. Lastly, as for any program, the macro itself should have an adequate program header and plenty of comments within the code to describe each section, as well as employing the usual techniques such as indentation, to aid readability.

FORMAL VALIDATION

This is the process whereby an independent tester subjects the macro to a battery of pre-specified tests, designed to verify that it functions exactly as the developer intended. It is recommended to follow an established process for this, requiring rigorous documentation of the tests performed, and formal approval. Encapsulating common tasks within macro utilities can help simplify the validation process, since the individual modules can be separately tested. Ultimately, a judgment is needed: the process can be costly in terms of time; however, if formal validation removes the subsequent need for independent double programming for QC purposes, whenever the macro is used, the investment may be quickly recouped.

MACRO COMPILATION

Compiling a SAS macro (or suite) and making it available in the form of a catalog has a number of advantages. Chief amongst these is the ability to control the version of the code in use. End users will not see (or be able to alter) the macro source code, preserving the validity of any formal testing. Furthermore, a catalog version number can be associated with each macro at the time of compilation, and this version number can be reported to the SAS log via a simple utility called at the start of each major macro. In the event of a user reporting a problem, the developer can see from the log which catalog version is being used.

ALTERNATIVE TECHNIQUES

Lastly, there are some other programming techniques that are useful. Examples are: CALL EXECUTE, which is another way of writing data-driven code; and PROC FCMP, which allows a developer to bundle SAS code into a function to perform a specific task. These techniques are beyond the scope of this paper, but both provide an alternative to writing SAS macro code and are described elsewhere [2] [3].

CONCLUSION

The development of SAS macros, either stand-alone or as a module of a larger application, for use by others is a multi-faceted activity, requiring a disciplined and rigorous approach. Users often attempt to use software in ways that developers cannot always anticipate. Although not the primary function of most macros, parameter validation and error handling become key tasks, often accounting for a large proportion of code written. Having a few relevant utilities to hand and pausing to consider some good programming practices will hopefully help developers get organized and create structured, robust code with increased efficiency and consistency.

REFERENCES

[1] Gregory, Martin. 2009. “Techniques for writing robust SAS macros” PhUSE 2009, AD01

[2] Humphreys, Iain. 2009. “Accessing SAS metadata and using it to help us develop data-driven SAS programs” PhUSE 2009, TU06

[3] Smith, Jason. 2014. “Define Your Own SAS Functions for Date and Numeric Validation Using PROC FCMP” PhUSE 2014, CC07

CONTACT INFORMATION

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

Iain Humphreys
PRA Health Sciences
500 South Oak Way
Reading RG2 0TD, UK
Email: HumphreysIain@prahs.com
Web: www.prahs.com