

SAS macro validation criteria

Jim Groeneveld, OCS Consulting, Rosmalen, the Netherlands.

A. ABSTRACT

Developing and using general SAS® macros may reduce development time of dedicated program code, errors and frustration. It may increase the quality and the quantity of the resulting output. However, especially in pharmaceutical research all generated SAS code has to be validated, including newly developed macros. Though the validation process may be partially the same for dedicated SAS programs and macro code there are also obvious differences between them.

SAS macro validation involves many issues, which can mainly be summarized into the following categories:

1. testing with various data and argument input parameters, especially checking for possible bugs and errors;
2. studying source code with regard to program layout, heading, history, comments, defaults, version control;
3. studying source code with regard to program efficiency, non-interference with existing datasets and variables;
4. verifying all kinds of checks on user input and specified datasets;
5. verifying readability of log files and possible results output.

The various characteristics of standard SAS macros will be discussed with their implications regarding validation.

B. INTRODUCTION

SAS macros, performing general routine tasks, which are being written with the purpose to be widely useful in various areas of research, or in certain specific areas, like pharmaceuticals, can be called standard or general macros. Once written they serve to save the user a lot of work and time, they limit the chances on programming errors and they promote standardization of output, whether that consists of datasets or tables, graphs and the like. But, as with many other new solutions they have to prove themselves, they have to be tested, it has to be verified that they do what they pretend to do, nothing more, nothing less, always, without errors, reliably and justified. They have to be validated and approved, certified. And once approved they have to be validated again after every change, bug fix, new feature and so on, at least for the part that has changed but rather completely again and preferably in the environment (calling programs) where they are being applied.

Macros automate and replace often applied (and written as dedicated source code) functionality within SAS programs in a very easy and generally applicable way. Macros may be run from the original **source code** as well as in a **compiled, unreadable** form. From a user point of view a standard macro may be regarded to be divided into two logically different parts:

- a user interface, consisting of **user documentation** on its use, functionality, input, output, etc.;
- the actual macro (source) code, including programmer comments, not of interest to a common user.

Throughout this document the programmer, developer, validating person and the like are being indicated as "he", "him" and "his" without excluding "she", "her" and "her" in the meaning of those words.

C. ADVANTAGES

User

The advantages of SAS macros (routine programs) over dedicated SAS code to the user are:

1. gain in programming time
2. gain in productivity, increase in quantity of programming results
3. less chances on programming errors, increase in quality and efficiency of programs
4. reusability of macro code in many programs
5. less dedicated program code, less copy and paste from other programs
6. gain in readability and transferability of dedicated programs
7. standardization of output (datasets, tables), increase in quality of programming results
8. use of macros similar to functions in SAS or macro code, returning a (return) value
9. high reliability (after validation and approval) of macro and remaining dedicated code

The user has to do with the macro header, the arguments to use in the macro call, the documentation, explaining those arguments in detail, the purpose of the macro, its input (datasets, etc.) and the results (readable output, datasets, etc.).

Developer

The macro programmer also has to do with the macro source code and the comments therein.

The advantages of developing, maintaining and validating SAS macros are:

1. maintenance of routine functionality in a single macro instead of repeated or reinvented in many programs
2. extensibility (if "well-written" macro code to leave room for easy extensions)

PhUSE 2006

3. determination of standard output layout or style (company dependent)
4. easily transferable to another programmer
5. divide expert programming work over more than one expert programmers

D. VALIDATION

Purpose

The main purpose of the validation process is to check whether the program, a SAS macro in this instance, serves its main goals, which generally are:

- to produce the one and only correct result, given correct input, in all instances;
- to clearly refuse to produce any (erroneous) result at all in case of erroneous input and to halt (abort) a whole running application program (without quitting the SAS GUI).

Input to a SAS macro is considered to consist of both data (e.g. datasets) and parameter specifications, which indicate how to handle the data.

Guidelines

Outlined in brief SAS macro validation should consist of the following aspects (more or less in (chrono)logical order):

1. Testing with known, provided data and known results, check for correct results (white box testing);
2. Testing with unknown data and unknown results, check for plausible results (black box testing);
3. Testing with all variations of argument input parameters, check their effects, testing with existing but known erroneous data and argument parameters (fool proof testing), check for appropriate responses, error reports and their consistent layout and verify whether all parameters are being checked on validity (legal SAS names, limited options, numbers, etc.), verify checks on existence of specified dataset(s) and variables within dataset(s), also verify checks on non-existence of newly to create permanent dataset(s), and/or an indication that these may be overwritten if existing, verify possible cross checks between input parameters;
4. Verify logic of default values of parameters of arguments, if any;
5. Read source code, verify correct programming according to functional specifications and the extent of algorithm efficiency;
6. Check for descriptive heading and user documentation with examples, purpose and verify presence of identifying information: programmer, SAS version, macro version and its date;
7. Verify the presence of a version numbering system and preferably version control support;
8. Verify presence of history of development, each change briefly indicated and especially check changes from last version (see history), all variations of bug fixes;
9. Verify readability, layout of source code and (substantial amount of) comment, and check for grammatical errors and typing errors, especially in comments;
10. Verify that all macro variables, unless explicitly %GLOBAL or macro call arguments, are explicitly %LOCAL and initialized;
11. Verify that all macro created temporary SAS variables in (temporary) datasets have unique names (stored as macro values);
12. Verify that all macro created temporary datasets have unique names (stored as macro values), not interfering with others and verify that they explicitly are deleted after use (also in case of errors);
13. Check for sufficient flexibility, allowing the user to specify titles and footnotes outside the macro (instead of fixed macro generated title and footnotes);
14. Verify saved state of SAS system options (GETOPTION) if options are being temporary changed in the macro, and reset OPTIONS at the end of the macro;
15. Verify readability, layout of readable output (tables, any results), possibly standard footnotes on each output page with date, SAS version, macro name, version., etc.;
16. Check log files for easy reading macro log part, incl. macro header in log and verify presence of review of user specified parameters in log files;
17. Verify optional debugging system options for feedback in log file;
18. Verify presence of auxiliary macros (separate or appended in same file);
19. Check method of (compiled) macro storage and autocall in main program, preferably not via %INCLUDE;
20. Verify presence of a test plan or procedure for SAS and macro code validation or write it;
21. Write validation report, both in case of approval and disapproval;
22. Optionally sign the validated macro electronically (e.g. using PGP or similar software).

These aspects of macro programming and validation will be discussed in detail below, not only from the viewpoint of the person who does the validation, but also from the viewpoint of the developer, who may account for the validation process and improve his macro accordingly. Next to a large amount of advice on many aspects of macro validation many programming hints are issued with respect to improving macro code quality. Such quality may reduce validation time, increase chances on a positive outcome of the validation process and last but not least increase user-friendliness.

1) WHITE BOX TESTING

Testing with known, provided data and known results, check for correct results (white box testing).

First of all a macro should be tested in the context it was meant for. Appropriate, legal parameters should be specified for its arguments, preferably default parameter values. These can be dataset names, libnames, variable names, format names, (macro specific) option keywords to indicate the macro's actions, etc.. Valid and appropriate, existing datasets should be used of which the contents are known. It should be clear what the result of the macro processing

PhUSE 2006

should be in terms of output datasets or output listings. These should be verified for correctness and possibly checked against other known correct results.

This is the main kind of testing that a developer carries out too while developing his macros. These tests of course should be completely successful; that is the goal of the macro.

2) BLACK BOX TESTING

Testing with unknown data and unknown results, check for plausible results (black box testing).

Black box testing should be carried out by trying all input options one at a time, while all the time normal output should be produced. This does not necessarily include all possible combinations of input options under all possible circumstances. It may be virtually impossible to check all possible variations in the functionality of a complex macro. During development new features have been added one by one and tested in their target environment mainly, while testing them in other situations might be either redundant or not applicable. So is validation, though during the validation process it may be difficult to judge sufficiently reliably in which situations additional checking is superfluous. Thus during validation checking actually should be performed at least as or even more extensively than during development.

3) FOOL PROOF TESTING

Testing with all variations of argument input parameters, check their effects, testing with existing but known erroneous data and argument parameters (fool proof testing), check for appropriate responses, error reports and their consistent layout and verify whether all parameters are being checked on validity (legal SAS names, limited options, numbers, etc.), verify checks on existence of specified dataset(s) and variables within dataset(s), also verify checks on non-existence of newly to create permanent dataset(s), and/or an indication that these may be overwritten if existing, verify possible cross checks between input parameters.

This is one of the most important tasks of the macro: it explicitly should check all user specifications in order to produce no-nonsense results. It should know that datasets, that are to be processed (read, analyzed) and hence expected to exist, do indeed exist. If such a dataset does not exist, the macro may as well abort. This also applies to user specified variables as macro parameters. A newly to create dataset on the other hand actually should not yet exist or should be allowed to be overwritten. Furthermore legal names have to be user specified, illegal names of course do not exist either (but can not be checked with respect to that) and neither can be created. Cross checks are necessary to determine the possible presence of legal, but discrepant parameters (e.g. the same variable specified twice or in two different variable lists). Finally a macro may have a limited number of specific arguments (numbers, letters or words).

A macro should extensively check all parameter values and during the validation process not only the macro code has to be studied to see whether these checks are present, but the macro should also extensively be checked whether it can get to a state that the macro author did not foresee and which aborts the macro in a very user unfriendly way, generating many SAS errors, or a state in which the macro continues producing erroneous results. If it has been programmed fool proof it should react to any inconsistency in a user friendly way with a clear error message and abort without producing any results (if possible). A brief example is a small part of the parameter checking within a macro:

```
%*~~~~~;
%* Check arguments for legal contents and discrepancies, calling e_r_r_o_r_s ;
%* _____;

%LET ErrCount = 0;
%IF ( &Data EQ ) %THEN
%DO;
  %PUT *** &MacName *** &ERR: Dataset not specified, macro &MacName will abort;
  %LET ErrCount = %EVAL ( &ErrCount + 1 );
%END;
%ELSE %IF ( %SYSFUNC(EXIST(&Data)) EQ 0 ) %THEN
%DO;
  %PUT *** &MacName *** &ERR: Dataset &Data does not exist;;
  %PUT macro &MacName will abort;
  %LET ErrCount = %EVAL ( &ErrCount + 1 );
%* See also SAS Guide to macro processing, vs. 6, 2nd ed. p. 265: macro Exist;
%END;
%ELSE
%DO; %* Check for non-existing variables within dataset;
  %ChkVar (Data=&Data, CalledBy=&MacName, VarList=
    &Group1 &Group2 &Group3 &Group4 &Group5 &Group6 &Group7 &Group8 &Group9,
    Result = ErrCount); %* adding up non-existent variables ;
%END; %* ChkVar is an external auxiliary macro which checks variable existence;
%IF ( &Group1 EQ ) %THEN
%DO;
  %PUT *** &MacName *** &ERR: Group1 not specified, macro &MacName will abort;
```

PhUSE 2006

```
%LET ErrCount = %EVAL ( &ErrCount + 1 );  
%END;
```

and so on and on and on, and finally:

```
%IF ( &ErrCount GT 0 ) %THEN %GOTO Aborting; /* label near the end of the macro;  
/* -----end-of-checks----- ;
```

The macro label *Aborting* and what follows is to be seen in the example in the paragraph 'Datasets' below.

The difference between white box, black box and fool proof testing is that white box testing consists of checking a few known situations, similar to the situation(s) which led to the development of the macro. User specified parameters are only legal and correct. Black box testing also tries to check legal, comparable, complex, but yet unknown situations to see whether the macro is suited for those too, while fool proof testing tries to crash the macro, while it never should.

4) DEFAULTS

Verify logic of default values of parameters of arguments, if any.

Macro parameter values may have default values, which should be quite appropriate in most cases. The default input dataset, for instance, might be set to the last dataset processed in the SAS program at that point, just like the default for quite some PROCs. Other default values may indicate the most common statistics to be calculated (valid number, missing number, mean, standard deviation, median, etc.), a standard option on how to handle missing values, a usual numeric format for a variable and a common ORDER in PROC REPORT. The example below is a part of the code that is presented in paragraph 'Heading and Documentation' below:

```
( Data = _LAST_ /* Input data set specification R */  
Processing of this value, is carried out by the macro as indicated below:  
/* Process keyword _LAST_ with Data;  
%IF (%UPCASE(&Data) EQ _LAST_) %THEN %LET Data = &SYSLAST;
```

Setting the automatic macro variable SYSLAST to the value of the current macro variable &Data at the end of the macro is done by:

```
%LET SYSLAST=&Data; * reset to original SYSLAST value;
```

This is necessary because &Data (especially if it wasn't _LAST_) may not be the last input dataset to be processed within the macro, while to the user, who calls the macro, it is the last dataset being logically processed in the calling code at that point. If the macro creates a new permanent dataset, named by the user, then that name should be assigned to macro variable SYSLAST in case of successful processing. In case of errors, recognized by the macro, SYSLAST has to be set to the input dataset &Data (see paragraph 'Datasets' below).

Other default values are very much macro dependent. It is left to the judgment of the validating person to decide whether chosen default values are appropriate in the context.

5) SOURCE CODE

Read source code, verify correct programming according to functional specifications and the extent of algorithm efficiency.

Here the validating person should focus on the SAS code, the algorithms applied and their efficient programming. He should try to understand the program code and to judge not only whether the solution was programmed efficiently and optimally, but also whether it was programmed correctly at all. In order to enable a reviewer to interpret the program code as good as possible it is very much necessary to keep the code short and logical, not more complex than needed, but neither more compressed than needed. Furthermore the need for extensive comments within the program code can not be sufficiently stressed. These facilitate the validating person, as well as another programmer, taking over the macro development, to quickly understand the code.

Keyword parameters should preferably be used in the macro argument list, to avoid user errors while calling the macro. Only if a macro has only one parameter, a positional parameter is appropriate.

While reading datasets (SET) it is recommended to only read the variables needed (KEEP dataset option).

Data step code and many PROCs should be terminated by a RUN statement; some PROCs, like PROC DATASETS and PROC SQL (also) need a QUIT statement.

6) HEADING AND DOCUMENTATION

Check for descriptive heading and user documentation with examples, purpose and verify presence of identifying information: programmer, SAS version, macro version and its date.

Macro program headers may take many forms. The main objective is readability and amount of information about the macro and its arguments. If there is no separate documentation, like a Word document, the documentation should be part of the header comment. That is the most important aspect that has to be verified. A proposed layout is: The macro comment box starts rather above, instead of below the %MACRO statement itself and contains a summary of the macro name, the version number, the version date and the author's name etc.. A %MACRO statement itself may take more than one line, i.e. a single line per argument with:

PhUSE 2006

- argument name
- default argument value or empty
- short comment (between /* and */) , description of argument and its values (see example)
- comment may contain Required or Optional indicator (R/O)

In that way the arguments need not to be described within the main comment box (the title box), but they may be described more thoroughly in later comments.

An example is:

```
*****;
** SAS macro MR2RM      vs. 2.7.3k, 10 May 2006, by Jim Groeneveld, Netherlands **;
*****;
**
*; %MACRO    MR2RM      /* Conversion of Multiple Records To Repeated Measures */
/* -----
/* ----- A l w a y s   s p e c i f i e d   a r g u m e n t s ----- */
/* Argument Default      Description and remarks      Required/Optional: R/O */
/* -----
( Data      =_LAST_      /* Input data set specification                        R */
, Out      =              /* Output data set specification (may be InputData) R */
, ByList    =              /* Variable List of which multiple Identical value R */
                        /* combinations determine a single new record */
                        /* (often variables like a Case Number) */
/*
/* ----- F r e q u e n t l y   s p e c i f i e d   a r g u m e n t s ----- */
/* Argument Default      Description and remarks      Required/Optional: R/O */
/* -----
, OverWrit=N            /* specify Y to overwrite an existing Out dataset    O */
, VarList =              /* Multiple Records variable name List to convert    O */
                        /* to Repeated Measures using PreFxLst and Index, */
                        /* No value or '_ALL_' involves all variables */
/*
```

and so on. The %MACRO statement ends with and is followed by more extensive, descriptive comment, e.g.

```
) / DES      =          'Convert Mult Recs to Repeated Measures' /* STORE */ ;
%*
%* -----
%*
%* File       : MR2RM.SAS / MR2RM.273
%*
%* Purpose    : Transformation of datasets with Multiple Responses as single
%*              variables in multiple records (observations) into one with
%*              those as multiple variables (Repeated Measurements) in a
%*              single record.
%*
%* Programmer : Jim Groeneveld
%* Date       : 10 May 2006
%* Version    : 2.7.3k
%*
%* SAS version: 6.12 and W95, 8.2 and W95/W98/W2K/WXP, 9.1.3 and WXP
%*
%* Validation : - (not yet validated, check results, use at your own risk)
%* Disclaimer : The author is not liable in any way for any negative, adverse
%*              consequences of the use or misuse of this program / macro.
%*
%* -----
%*
%* Notes:
%* =====
%*
%* ~)-Generating an index variable automatically should be applied with care: *
```

The main question is: is a macro sufficiently user-friendly and comprehensible in terms of purpose, documentation and macro call with its arguments and (default) parameter values? This is to be judged by the validating person.

7) VERSION NUMBERING AND CONTROL

Verify the presence of a version numbering system and preferably version control support.

Version numbering

Next to their descriptive names macros should preferably have version numbers to discriminate various stages in the development of them and to be able to check whether one has the last version. The last maintenance date should, in

PhUSE 2006

the macro comment, be related to the version number. The number should preferably not be part of the name. It may vary from a very simple, single sequence number to a complex numbering scheme as often present with commercial or important software. A suggestion is a multi-part number, consisting of a major number part, a minor number part, and patch (or revision or maintenance) part. Possibly a build part can be added, which is for internal use only during development. An example of a Major.Minor.Patch[.Build] number is 2.4.1.c. All numbers start at 0 and a validated first release should have number 1.0[.0]. Validated versions may have even minor numbers, experimental, but not yet validated versions may have odd minor numbers.

During the development and maintenance (life-cycle) process of SAS macros, especially while they are already used for production purposes, it is very much recommended to keep a history of changes, additions and bug fixes along with the macro version number. Archiving of the production SAS program code should include the applied macro version or at least the applied macro version number. That way the code can be used for reproduction purposes.

Newer versions

Changing a macro actually changes all programs calling that macro and with some programs those changes may appear to cause unintended or even erroneous results, either unexpected or if the macro is known not to be backwards compatible. The goal of any version control system is to force a certain older, but known to be reliable, version to run and to prevent the default, newest version to run always. It is the intention to guarantee the same macro version to always run with the original calling code.

Newer macro versions may never change functionality, but merely fix bugs (which did not emerge in practice until then) and add functionality, which explicitly should be specified in the macro call. So, they are backwards compatible. Newer versions of a macro should be as much as possible backwards compatible. Then a newer version may replace an older version when running reproduction SAS programs.

Backward incompatibility

Once some significant functionality has changed, breaking backward compatibility, e.g. defaults or changed argument names, it should not be used for reproduction purposes with code that used to call the older macro version. Instead the older macro version has to be used. In such a case version control is essential. Backward incompatibility examples are:

- changed or removed argument names;
- changed default parameter values;
- changed or removed parameter values with specific meanings;
- changed functional actions of existing parameter values.

During the development and maintenance (life-cycle) process of SAS® macros, especially while they are already used for production purposes, it is very much recommended to keep a history of changes, additions and bug fixes along with the macro version number. Archiving of the production SAS program code should include the applied macro version or at least the applied macro version number. That way the code can be used for reproduction purposes. The user of the macro explicitly is encouraged to specify the version number in the macro call to ensure that version to run always, also after new versions have become available: Version=[number].

Description

This version control system consists of an additional argument Version in the macro call and of macro code interpreting the Version value. It has the current version number as the default value. This number is checked against the hard coded version number inside the macro, and if they agree (or if Version is specified empty) the macro is allowed to run as usual. If the user explicitly specifies an older, known backwards incompatible version number it will not perform its normal action, but then the macro stops with an appropriate message and the user should take care to replace that (newest) macro by the appropriate older version of the macro. Possibly incompatible or at least different results are thus being avoided. If the user specifies another, but backwards compatible version number, the newer macro may allow itself to run anyway. It would even be possible to let the newer macro automatically run the older one if forced to, but that is not always necessary. The Version argument is part of the %MACRO statement:

```
, Version = 2.7.3k    /* forced version specification for version control 0 */
```

The code that interpretes the version number is:

```
%IF (&Version NE AND &Version NE &MacroVs) %THEN
%DO; /* if Version not default and not empty */
  %PUT *** &MacName *** &ERR: Specified version &Version does not match &MacroVs;;
  %PUT .                               macro &MacName will abort;
  %LET ErrCount = %EVAL (&ErrCount + 1);
  %GOTO Finish;
%END;
```

In there &Version represents the version number from the macro argument, possibly user specified. &MacroVs represents the fixed current (newer) macro version number, &MacName contains the macro name and &ErrCount counts (user specified) parameter errors. At the end of the macro there is a label Finish: to which a jump can be made.

PhUSE 2006

Alternatives

An alternative action of an incompatible newer macro may be to reproduce the different behaviour of the older version in case the older version number is specified. An other alternative is always also to allow currently undocumented, outdated argument names and parameter values as long as these can be interpreted unambiguously.

8) HISTORY

Verify presence of history of development, each change briefly indicated and especially check changes from last version (see history), all variations of bug fixes.

It is important to track changes from a first useful release of the macro. If in any subsequent version new bugs emerge it is possible to analyze from which change the bug most likely has been caused. For users it is important to know which bugs have been solved and which new features added. An example of a piece of history comment is:

```
%* History of MR2RM:                                     *;
%* =====                                              *;
%* version   date      description of changes, additions, improvements *;
%* -----   - - - - - - - - - - - - - - - - - - - - - - - - - - - *;
%* 0.0       14 May 1998 first working release converting a single variable *;
%* 0.9       4 Jan 1999 made suitable for both alpha and numeric variables *;
%* 1.0       5 Jan 1999 minor improvements and bug removals             *;
%* 1.0.1     15 Feb 1999 added len (length of result variables)          *;
%* 1.0.2     16 Feb 1999 added some minor checks on arguments           *;
%* 2.0       14 Jun 1999 made suitable for any number of MR variables at once *;
```

9) CODE READABILITY

Verify readability, layout of source code and (substantial amount of) comment and check for grammatical errors and typing errors, especially in comments.

The validating person should “walk through” the source code in order to judge several coding aspects. He should verify the presence of sufficient and comprehensible comment and section (sub) headers (notable comments), including their hierarchy within the code. English is the most preferred language in comments and documentation. Comments may be on separate lines (before the described code) if long or introductory. Inline comment consists of short text, a few important words, and is generally inserted after the code on the same line. Next to comment variable and dataset labels also are a source of descriptive information.

Indentation between DATA and RUN statements in a data step is quite common as it is within PROC steps. Indentation is also applicable dependent on the level of conditional IF-THEN-DO-END blocks of code.

Lining up complex bracket structures to avoid errors while coding and to quickly see the meaning of the brackets during inspection is very much recommended. Even better is to avoid complex conditions and bracket structures.

The words “ERROR”, “WARNING” or “UNINITIALISED” should preferably be hidden in the program code such that searching for them in the program code does not find them. E.g. the macro variable ERR will get the value ERROR:

```
%LOCAL Err Warn;
%LET Err      = %SYSFUNC(COMPRESS(E R R O R , %STR( )));
%LET Warn     = %SYSFUNC(COMPRESS(W A R N I N G , %STR( )));
```

Meaningful (macro) variable names should be used as often as possible (see also the paragraph ‘SAS variables’).

While it may be good practice to write SAS keywords in program and macro code in capitals to easily distinguish the code visually from comments (in lower case) it may be good practice as well to use mixed case for all names within the code: a starting capital and a capital for each first character of a subsequent part within a compound name.

Source code statements should preferably be coded on different lines (with proper indentation and so on), but there may be good reasons for exceptions in specific cases, which improve readability even more. These specifically are:

- one-liners: a proc (e.g. SORT), that can be coded within 80 characters completely;
- parts of code, that specifically belong together, such as BY or ELSE after what it relates to, or (multiple) assignments with their accompanying OUTPUT statement;
- repetitive sets of statements that fit on one line per set;
- SAS statements and descriptive comment statements;
- mixtures of normal SAS program code and SAS macro code; and more.

By limiting the number of empty lines by using them only to separate discriminative blocks of related statements, quite a lot of code may become clearly arranged and visible within one screen, or on one printed page. SAS users apply various programming styles, of which there is not one always superior over another. The goal (the intention of the rules) is optimal readability. To SAS itself the readability does not matter at all.

Examples of all these issues regarding the program layout clearly emerge from the other code examples throughout this document.

10) MACRO VARIABLES

Verify that all macro variables, unless explicitly %GLOBAL or macro call arguments, are explicitly %LOCAL and initialized.

GLOBAL macro variables should hardly be used. Macro variables are implicitly local if they are specified as user arguments in the macro call (%MACRO statement without PARMBUFF option). Most, if not all other auxiliary macro variables in the macro are only used within that macro or possibly also in (auxiliary) lower level macros called from there. In order to make sure all internally used macro variable names are local and do not interfere with possibly equally named macro variable names at the higher, calling level (whether global or local there) all internally used macro variables should be specified local explicitly using the %LOCAL statement. This might be extended to specifying each variable name on a single line followed by a short comment on the variable. This looks similar to declaring all used variables which is obliged in several programming languages. For an example see:

```
%*~~~~~;
%* LOCAL macro variables declarations;
%* _____;

%LOCAL    /* Define internal macro variables explicitly local */
I          /* element index */
Id         /* IdList element */
LastId     /* last Id from IdList */
MultRec    /* MRlist element */
[.....]
;
```

If values should be returned to the calling level that can be done in (combinations of) several ways:

- as GLOBAL macro variables (not recommended, because too global everywhere);
- as the only SAS code to generate (no other SAS code should be applied in the macro);
- via a callable macro argument, which names the higher level macro variable to contain the result value.

Return by argument

An example of the third kind is presented below, firstly a piece of the macro header and %MACRO statement:

```
*****;
** SAS macro ChkVar    vs. 1.3.1 ,   9 May 2006, by Jim Groeneveld, Netherlands **;
*****;
**                                                                    **;
*; %MACRO    ChkVar      /* Checking existence of variables within a dataset */
/* ----- */
/* Argument Default      Description and remarks      Required/Optional: R/O */
/* ----- */
( Data      =_LAST_      /* Input data set specification                R */
, VarList =              /* List of variables to be checked for existence                R */
, Result =              /* Name of macro variable in calling environment            R */
/* (without '&' prefix) to add up number of errors                */

```

Note the macro argument (variable) Result. The following code contains the macro call to ChkVar and is part of the parameter checking example at the calling level discussed with fool proof testing. The argument ErrCount is the local macro variable at the calling level that is specified as the Result parameter:

```
%DO; %* Check for non-existing variables within dataset, this is calling code;
    %ChkVar (Data=&Data, CalledBy=&MacName, VarList=
        &Group1 &Group2 &Group3 &Group4 &Group5 &Group6 &Group7 &Group8 &Group9,
        Result = ErrCount); %* adding up non-existent variables ;
%END; %* ChkVar is an external auxiliary macro which checks variable existence;
```

The code below is the most essential code of the macro ChkVar where either the Result or Check macro variable is being processed. After several tests, if a variable does not appear to exist in the dataset the ErrCount macro variable at the calling level has to be increased by 1:

```
%LET &&Result = %EVAL ( &&Result + 1);
```

This statement directly increases the ErrCount macro variable from the higher level because it is being resolved as;

```
%LET ErrCount = %EVAL ( &ErrCount + 1);
```

This is a very nice method to use if more than one value has to be returned to the calling level or if only one value has to be returned, but the macro also generates various SAS code which can not be used as a return value.

A warning in this case must be that the macro variable name used as a parameter at the calling level (ErrCount in the example) never may be equal to any macro variable within the macro, whether a macro argument or LOCAL macro variable name. The explanation is that the macro variable from the calling level should be available within the macro. The macro explicitly should check for such instances and react appropriately (it knows its own local macro variables).

Return by SAS code

If only one macro value (or a list of values) has to be returned from a macro that entirely consists of macro code and thus does not generate, return SAS code the method as under b. above is very much applicable. An example is: suppose the value to return is 1 and it is stored in macro variable ToReturn. Then amidst all macro code just a single SAS statement is essential:

```
&ToReturn /* no semicolon! The resolved value is the only returned SAS code */
```

This way of returning single values is also the way to develop and use function like macros. After all the value on itself is not a valid statement in the calling SAS or macro code that calls the concerning macro (function). It should be part of a statement, e.g. an assignment statement in the calling code (suppose macro Function):

```
RtnValue = %Function (...parameters...); * (SAS code);
%LET RtnValue = %Function (...parameters...); *% (macro code);
```

Instead of returning a single value one might also return a list of values with specific, unique separator characters. The list may be built most easily by the macro using PROC SQL or CALL EXECUTE. The list may be analyzed in the calling code using the %SCAN or SCAN functions. Sample code is:

```
* SQL generated variable name list into macro variable;
PROC SQL;
SELECT NAME INTO : VarList SEPARATED BY ' ' FROM SASHELP.VCOLUMN
WHERE LIBNAME EQ "&LibName" AND UPCASE(MemName) EQ &Data;
QUIT; /* resulting list is &VarList;

* CALL EXECUTE generated variable list into macro variable;
%LET VarList = ; /* empty;
PROC CONTENTS DATA=&Data OUT=&Contents (KEEP=Name) NOPRINT;
RUN;

DATA _NULL_;
SET &Contents;
CALL EXECUTE ( '%LET VarList = &VarList ' || Name || ' '; ');
RUN; /* resulting list is &VarList;

* %SCAN generated variable list in macro variable;
%LET I = 1;
%LET Variable = %SCAN(&VarList,&I,%STR( ));
%DO %WHILE (&Variable NE);
  /* do something with &Variable here.....;
  %LET I = %EVAL( &I + 1 );
  %LET Variable = %SCAN(&VarList,&I,%STR( ));
%END; /*=== While;

* SCAN generated variable list in macro variable;
DO WHILE (SCAN("&VarList", I+1, ' ') NE ' ');
  Variable = SCAN("&VarList", I+1, ' ');
  * do something with Variable here.....;
  PUT Variable=;
  I + 1; * sum statement increments I and RETAINS it with initialization 0;
END;
```

Differences

The advantage of methods b) and c) (return by argument and return by SAS code) over method a) (return by global) is that the return values only exist at the higher level where the user wants them and as long as the user wants them to exist. They do not exist at even higher levels during the whole SAS session where the user has insufficient control over macro variables left over from previous macro calls. For small, but complete examples of both methods b) and c) see under the paragraph 'SAS variables'.

11) SAS VARIABLES

Verify that all macro created temporary SAS variables in (temporary) datasets have unique names (stored as macro values).

Anywhere one likes one may and is encouraged to apply stand-in local macro-variable names for dataset variables. Stand-in names (aliases) internally may be chosen completely freely, while externally (outside the macro) they represent names following the macro specific naming conventions for scratch datasets and variables. The advantage is to be able to use all allowed (8 or 32) characters of a (SAS) name to be meaningful; an additional advantage is the possibility to easily change the external name of an entity at one place if necessary.

An example: suppose one needs an additional variable to count something in a dataset. As one doesn't know anything yet about those datasets some name pattern has to be chosen that is very unlikely to occur in datasets in

PhUSE 2006

general. E.g. names starting with an underscore, or names with all kinds of characters, not necessarily just letters, digits and the underscore in the form 'Any Name'n. Let say we choose the name ____Freq. Then we declare:

```
%LOCAL CountVar;  
%LET CountVar = ____Freq;
```

and subsequently refer to the variable as &CountVar, while the macro variable name is unique by definition within the macro. It is even better to generate a unique name, e.g. using the current time of the day and to check firstly whether that name does not already occur in the dataset. If it does a new one should be generated and checked for its existence. Example code to generate unique SAS names is presented below under the paragraph 'Datasets'.

12) DATASETS

Verify that all macro created temporary datasets have unique names (stored as macro values), not interfering with others and verify that they explicitly are deleted after use (also in case of errors).

Stand-in names should preferably also be applied for temporary SAS datasets. As with SAS variables it is even better to generate a unique name, e.g. using the current time of the day and to firstly check whether a dataset with that name does not already exist in the work space. If it does (coincidentally) a new name should be generated and checked for its existence.

Unique, random names

The following two examples (both pilot macros to be developed further) generate a random SAS name for use as a dataset name or, if modified, as a variable name. The first example uses the return value method Return by Argument and the second one the method Return by SAS code, both discussed in the paragraph 'Macro Variables':

```
/* Example of Return by Argument;  
/* ===== GenerateUniqueDatasetName.sas =====;  
%MACRO GenerateUniqueDatasetName (Libname=, Prefix=_, DatasetName=);  
  /* Requirements (no built-in checks yet);  
  /* - Valid LibName;  
  /* - Prefix starting with letters or _;  
  /* - Valid macro variable name as dataset name;  
  %DO %UNTIL (NOT %SYSFUNC(FEXIST(&Libname..&&DatasetName)));  
    /* Add (pseudo-random) time to Prefix and change . into _;  
    %LET &DatasetName = %SYSFUNC(TRANSLATE(&Prefix.%SYSFUNC(TIME()),_,.));  
  %END;  
%MEND GenerateUniqueDatasetName;  
/* =====  
  
/* Application of GenerateUniqueDatasetName;  
/* Store unique name in macro var Dataset,  
/*   which already must exist at the calling level;  
%LET Dataset=;  
%GenerateUniqueDatasetName (Libname=WORK /* or empty */, DatasetName=Dataset);  
%PUT Dataset=&Dataset;  
  
/* Example of Return by SAS code;  
/* ===== UniqueDatasetName.sas =====;  
%MACRO UniqueDatasetName (Libname=WORK, Prefix=);  
  /* Requirements (no built-in checks yet);  
  /* - Valid LibName;  
  /* - Prefix starting with letters or _;  
  %LOCAL DatasetName;  
  %DO %UNTIL (NOT %SYSFUNC(FEXIST(&Libname..&DatasetName)));  
    /* Add (pseudo-random) time to Prefix and change . into _;  
    %LET DatasetName = %SYSFUNC(TRANSLATE(&Prefix.%SYSFUNC(TIME()),_,.));  
  %END;  
  &DatasetName /* no semicolon, pass on to calling code;  
%MEND UniqueDatasetName;  
/* =====  
  
%LET Dataset = %UniqueDatasetName (Libname=WORK /* or empty */);  
%PUT Dataset = &Dataset; DATA &Dataset; RUN;
```

Both macros generate unique dataset names; however these are the long forms. If one would adapt such a macro to generate random variable names (by checking for existing variable names) and if one uses SAS 6.12 or SAS 8.x or 9.x with the OPTION VALIDVARNAME=V6 then variable names may not be longer than 8 characters. These premature, example macros still should be adapted for that limitation too (see also the sub paragraph SAS names under the paragraph 'Optional features' below).

Furthermore these macros still should be extended with code that, next to check for existing dataset or variable names, should also check for already generated random names which are not yet in use as a dataset or variable

PhUSE 2006

name. As one can see there is still a lot to do on these example macros before becoming actually useful, but their only purpose here is to demonstrate elementary random name generation.

Cleanup

Deletion of work datasets should always explicitly occur, because the macro(s) are not exclusively used in a batch environment, but also in a single interactive environment (in which SAS is only started and initialized once). They should also be deleted (as far as created yet) in case of unexpected SAS errors or macro generated errors (as a result of erroneous user input). So they should always be deleted once the macro terminates and jumps to a label near the end of the macro, of which the following code is an example:

```
%*~~~~~;
%* Process error conditions ;
%* _____;

%Aborting:

PROC DATASETS NOLIST;          %* Delete scratch datasets;
  %IF (%SYSFUNC(EXIST(&_Sorted1))) %THEN DELETE &_Sorted1;;
  %IF (%SYSFUNC(EXIST(&_Sorted2))) %THEN DELETE &_Sorted2;;
  %IF (%SYSFUNC(EXIST(&KeepVars))) %THEN DELETE &KeepVars;;
  %IF (%SYSFUNC(EXIST(&_Extreme))) %THEN DELETE &_Extreme;;
  %IF (%SYSFUNC(EXIST(&Contents))) %THEN DELETE &Contents;;
  %IF (%SYSFUNC(EXIST(&IndexFrq))) %THEN DELETE &IndexFrq;;
RUN; QUIT;

%IF ( &ErrCount GT 0 ) %THEN
%DO;
  %IF (%SYSFUNC(EXIST(&Out)) AND
      %FirstCh(&OverWrit) EQ Y) %THEN %DO;
    PROC DATASETS; DELETE &Out; RUN; QUIT;
  %END;
  %LET SYSLAST=&Data; * reset to original SYSLAST value;
  DATA _NULL_;
    ABORT /*ABEND*/;
  RUN;
%END;
%ELSE
  %LET SYSLAST=&Out; * reset to explicit SYSLAST value;

%Finish:

%*****;
%MEND MR2RM; * -----end-of-macro-MR2RM-----;
%*****;
```

13) TITLES AND FOOTNOTES

Check for sufficient flexibility, allowing the user to specify titles and footnotes outside the macro (instead of fixed macro generated title and footnotes).

Macros may or may not automatically generate TITLES and FOOTNOTES. In both cases it is important that the user is able to specify his own TITLES and FOOTNOTES. One construct to enable both possibilities is to let the macro know of the user's TITLES and FOOTNOTES. These need not to be specified after a PROC and before a RUN statement, but may be specified long before the PROC where they belong to, even if there are data steps and other (data manipulating) PROCs in between. TITLES and FOOTNOTES are active until redefined by the same or lower numbered TITLES or FOOTNOTES. E.g. TITLE5 is redefined by a new TITLE5 and reset to nothing by one of the TITLES 1 u/i 4. The macro does not need to know of the TITLE or FOOTNOTE contents itself, but it just should know from which numbered TITLE or FOOTNOTE it can generate one or more itself. To that purpose a dedicated argument can be added to the %MACRO statement specifying the TITLE (or FOOTNOTE) number to start with from within the macro. The user then has the starting numbers up to that number to his disposal. Specifying 0 (zero) disables the macro's feature to generate TITLES (and FOOTNOTES). The argument in the %MACRO statement is coded as:

```
, Titles =0          /* show two standard titles from 1..9, 0=no titles 0 */
and in the macro code:
%IF ( &Titles_ NE 0 ) %THEN /* define TITLE(s) here */ ;
```

14) SYSTEM OPTIONS

Verify saved state of SAS system options (GETOPTION) if options are being temporary changed in the macro, and reset OPTIONS at the end of the macro.

A macro may use SAS system options and set them to specific values irrespective of the values up to the macro call. examples are: MPRINT, MERROR, SERROR, MLOGIC, SYMBOLGEN, MACROGEN, MISSING, LINESIZE, PAGESIZE and many more. This on itself is no problem at all. But at the end of the macro all these options have to be

PhUSE 2006

reset to their initial values, because the user, the programmer of the calling program does not know about these changing values and he should not bother about them, but he should be able to expect that they are still the same as before the macro call.

To that extent the macro, before setting, changing an OPTION, should store the initial value of that option in a macro variable using the SAS function GETOPTION, e.g. storing the current LINESIZE option:

```
%LET LS = %SYSFUNC(GETOPTION(LS));
```

Then the macro may freely change the LINESIZE. At the end of the macro the original LINESIZE has to be reset by:
 OPTIONS LINESIZE=&LS; /* reset original options;

15) OUTPUT LAYOUT

Verify readability, layout of readable output (tables, any results), possibly standard footnotes on each output page with date, SAS version, macro name, version., etc.

Macros may produce any kind of result, just for what they have been developed. Most commonly they produce either datasets with numerical results (calculated, restructured, aggregated, etc.) or they produce output for visual inspection like listings, tables and figures. Just like with usual dedicated program code a macro should take care to produce that output in (at least) the same quality as the dedicated program would have produced it. That may be a nice generally useful standard of layout, but it may as well be a company layout style, possibly optionally selectable. In any case the output should meet common standards with respect to readability and structure. It is up to the validating person to judge whether the output quality is sufficient.

It is quite common to add footnotes to output with information on the date of production, the SAS version, the current program name and its author and possible the macro name (and version). Hence, it should be possible that either the macro optionally produces such footnotes or that the user can program it outside the macro. All this determines the judgment of a macro in its application environment. A partial example of a nice, readable table structure is:

Table 1.2.3 Number (%) of subjects with clinically significant pre-existing medical conditions by system organ class and medical condition, within and across treatment groups, All-Subjects-Treated group

		Treatment group as treated					
		Product A (N=146)		Product B (N=120)		TOTAL (N=266)	
MedDRA primary SOC	MedDRA low level term	n	%	n	%	n	%
Blood and lymphatic system disorders	Spherocytosis			1	100	1	100
Cardiac disorders	Anginal pain	1	17			1	14
	Atrial fibrillation	1	17			1	14
	Bradycardia	1	17			1	14
	Mitral insufficiency	1	17			1	14
	Mitral valve incompetence			1	100	1	14
	Palpitations	1	17			1	14
	Valvular heart disease NOS	1	17			1	14
Congenital, familial and genetic disorders	Congenital jaw malformation	1	100			1	100
Ear and labyrinth disorders	Conductive hearing loss			1	33	1	20
	Ear drum perforation	1	50			1	20
	Hearing loss unilateral	1	50	1	33	2	40
	Vertigo			1	33	1	20

Study_name / Program_name / Date: 31 July 2006 / SAS version: 9.1.3 / by: Jim

PhUSE 2006

This (partial) table has been produced by the macro by performing most, if not all of the calculations with PROC FREQ, writing them to datasets, combining those in data steps and presenting the results using PROC REPORT.

16) LOG FILES

Check log files for easy reading macro log part, incl. macro header in log and verify presence of review of user specified parameters in log files.

Limiting log

Especially in case of problems, errors and bugs the log of a macro should be readable quite well and contain a sufficient amount of information to help catching the problem. If a macro has proved to function quite well and does not show any problems at all anymore, the author may decide to limit the amount of log information from the macro by setting specific system OPTIONS (see also paragraph System Options). He may especially set OPTIONS NONOTES to suppress most of the normal data step and procedure log and he may also use ERRORS=0 if these are not to be expected, though that may be disputable. On the other hand the macro should generate clear and appropriate errors and warnings itself if it encounters them while checking input data and user specified parameters in the macro call or if later in the macro process problems emerge. Under the paragraph Fool Proof Testing some of these error messages can be seen in the presented macro code.

Header

Generating a macro starting header in the log is very much recommended. The header may indicate the macro name, the version number, its version date, the author's name and a brief description of the macro's goal. And it is also advisable to generate a complete list of the macro's arguments and its (user specified) parameter values. In case of (unexpected) problems it can easily be seen which macro call caused the problems and what the macro parameters were. An example of log output is:

```
| ~~~~~ |
| Macro Crosstab, vs. 0.9.1s, by Jim Groeneveld, 19 Oct 2005 |
|          9-dimensional crosstabulation with percentages   |
| ~~~~~ |
```

```
Crosstab: Review of argument specifications (incl. defaults)
Crosstab: Data      = Test
Crosstab: Group1    = I
Crosstab: Format1    =
Crosstab: Missing1  = "Missing"
Crosstab: Total1    = "TOTAL"
Crosstab: Values1   =
Crosstab: Group2    =
```

and so on. The code to produce this kind of information is quite simple, just a series of PUT statements.

17) DEBUGGING

Verify optional debugging system options for feedback in log file.

During the process of developing or debugging a macro it may be desirable to increase the amount of information in the log to normal or even above normal. And next to logging information it may also be desirable to PROC PRINT temporary datasets as intermediate results of processing steps. Instead of having embedded PROC PRINT statements within the macro code, which normally have been commented out, it would be much nicer and much faster to generate all kinds of debugging information with just 'one press on the button', or in this case via a single, extra macro argument Debug. Its default value could be 0:

```
, Debug = 0 /* Debug level; for testing, debugging purposes only 0 */
```

At macro invocation the Debug value could be set to 1 or whatever value to be used within the macro. Dependent on specific values all kinds of OPTIONS to generate much more log output can be set, intermediate results can be printed or breakpoints indicated (using the PUT statement). Example code is:

```
%IF (&Debug EQ 1 OR &Debug GT 1) %THEN %DO;
  OPTIONS MPRINT MERROR SERROR MLOGIC SYMBOLGEN MACROGEN;
%END; /* in the beginning of the program;

%IF (&Debug > 10) %THEN
%DO;
  PROC PRINT DATA=&AggrData; TITLE "AggrData after forcing values in 1st loop"; RUN;
%END; /* TITLE may disturb user titles, but only during the debugging process;

%IF (&Debug EQ 0) %THEN NOPRINT;; /* NOPRINT option with PROC FREQ;

%IF (&Debug EQ 2) %THEN PUT "Debug breakpoint A: before first SORT";;
```

PhUSE 2006

```
%* At the macro end (or even unconditionally) reset original options to stored state;  
%IF (&Debug NE 0) %THEN OPTIONS &Mprint &Merror &Serror &Mlogic &Symbolgn &MacroGen;;
```

Note the two successive semicolons in the examples above: the first one ends the macro statement (whether the condition was true or not) and the second one ends the SAS statement. Thus if the condition is not true then the PUT statement is not generated, but the ending semicolon is; it then just is an empty statement without any consequences.

18) AUXILIARY MACROS

Verify presence of auxiliary macros (separate or appended in same file).

A list of called auxiliary macros should be part of the description with per macro an indication of:

```
%* = internal: - CompBloc *;  
%* = attached: - Spacing0 *;  
%* = external: - ChkVar PreZeros FirstCh ChkList CrossChk *;
```

Three types of auxiliary macros are being discriminated:

- a. **Internal** macros are macro which are defined and compiled a run time by the main macro or one of its auxiliary macros. Internal macros thus may vary in their definition. This kind of **nested** macros should not be used if possible, but can not always be avoided. An example is an alternative for a complex macro variable which value is conditionally dependent on several situations, macro CompBloc:

```
%*~~~~~;  
%* Runtime definition of macro CompBloc ;  
%* _____;  
  
%* Macro variables Across, Indent_, Grp#Var, Grp#Type, Group#N, Grp#Fmt,  
%* LblAsVal local to parent;  
  
%MACRO CompBloc ;  
  %LOCAL Nr; %* Group number 1 or 2, dependent on Across macro variable;  
  %IF ( &Across EQ 0 ) %THEN %LET Nr = 1;          %* Process Across into Nr;  
  %ELSE %LET Nr = 2;  
  %IF ( &Indent_ NE ) %THEN  
    %DO;                                     %* If indentation specified ;  
      COMPUTE BEFORE &&Group&Nr;             %* Define compute block before;  
      LENGTH &GrpValue $100;  
      &GrpValue = PUT (&&Group&Nr, &&Grp&Nr.Fmt);  
      %IF (%FirstCh(&LineSpac) NE S) %THEN LINE @1 ' '; %* optional empty line;  
      LINE @1 &LblAsVal &GrpValue $CHAR100.;  
    ENDCOMP;  
  %END;  
  [.....]  
%MEND CompBloc;
```

This macro is being used, called later in the macro in the same way and where otherwise the complex macro variable would be specified to perform the macro statements and to generate the conditional SAS code.

- b. **Attached** macros have a fixed definition, are called only from the main macro and are specific for that main macro. They are defined in the same macro (.SAS) files as the main macro (with different, discriminative names) after the code of the main macro. If the main macro is called from a SAS program for the first time the SAS AUTOCALL facility finds the macro as part of the file MacroName.sas. Once that file is read not only the main macro has been defined in the program, but the auxiliary macro(s) (if any) have been read and defined as well and can immediately be used to be called by the main macro by their own name. So they are not part of their 'own' .SAS files. These may be used as a subroutine by the main external macro. Therefore they can not be always avoided. They make macro programming easier and more transparent, but macro processing by the compiler more complex and slower. Some balance might be considered when choosing between repetitive inline code and a macro subroutine.
- c. **External** macros are macros defined separately and are contained in .SAS files with their macro name. They actually are macros like the main one, and might be called by any program. but they are only or mainly called by main macros as auxiliary macros, some kind of subroutines or macro functions within macros.

19) MACRO CALL

Check method of (compiled) macro storage and autocall in main program, preferably not via %INCLUDE.

Macros may be called, found and run from a main, calling program in several ways:

- a. as an uncompiled macro found using the AUTOCALL facility (also external auxiliary macros)
- b. as a compiled macro found using the AUTOCALL facility (also external auxiliary macros)
- c. as an explicitly %INCLUDEd macro, which is not the most elegant and preferable way

PhUSE 2006

- d. as an auxiliary macro, defined and compiled when a main macro with that macro as an attached one in the same .SAS file was found, defined and compiled; generally “not known” outside the scope of the parent macro
- e. as an auxiliary macro, defined and compiled when a main macro built, defined that macro like a complex macro variable; such an internal or nested macro could be called from elsewhere but that would not make sense; besides, the “elsewhere” should not know of that macro at all
- f. as an inline coded macro in the main program, only recommended for dedicated one-time macros;

Finding general (incl. external auxiliary) macros using the AUTOCALL facility is most preferable. Next to validating the macro on itself it is also worthwhile to see how it will be called by other programs if it is possible to observe that by the validating person. It should make no difference whatsoever which method is applied. The macro should perform identically in all situations.

20) VALIDATION PLAN

Verify presence of a test plan or procedure for SAS and macro code validation or write it.

A macro validation plan, to prepare in advance of the actual validation, can be part of the whole validation process. It is especially recommended if it is to be expected that the validation process may be a complex process with many tests to perform. This most often applies to large and complex macros. A test plan should describe the tests and checks to perform with regard to some or all aspects of macro validation that are mentioned here. It should at least describe:

- a. aspects of white box testing, parameters to use;
- b. aspects of black box testing, parameters to use;
- c. testing measure of fool proof, erroneous parameters to use, whether macro checks for valid parameter values and explicitly existing or not existing datasets and variables;
- d. way to judge source code with regard to algorithms, header info, documentation, history, layout and comments;
- e. verify presence of version control;
- f. checks on declaration and initialization of macro variables;
- g. checks on uniquely named temporary SAS variables in (temporary) datasets;
- h. checks on temporary datasets, unique named and deleted at termination;
- i. checks on the use of SAS system options, change and reset;
- j. judgment of readability of produced output;
- k. judgment on readability of log files (macro header, user feedback, macro generated errors);
- l. judgment of debugging possibilities;
- m. checks on internal and attached auxiliary macros.

All these aspects should return in a validation report at the end of the validation process.

21) VALIDATION REPORT

Write validation report, both in case of approval and disapproval.

While testing a macro with regard to various aspects the validating person should of course keep track of his findings. In order to officially declare a macro validated and approved at the end of the process he should at least summarize those findings in a report and make this report widely available. Such a report may contain the following paragraphs:

- a. White box test results, with default parameters if possible;
- b. Black box test results;
- c. Results of all kinds of parameters with all macro arguments, incl. erroneous ones, verification of checks in macro on valid parameters, but also on explicit existence or non-existence of datasets and variables;
- d. Judgment of programming algorithms in source code;
- e. Judgment of quality and quantity of macro identification and description in header and documentation;
- f. Judgment of version control system;
- g. Judgment of history description;
- h. Readability of source code with regard to layout and commenting (incl. spelling errors);
- i. Declaration and initialization of macro variables;
- j. Verification of uniquely named SAS variables in (temporary) datasets;
- k. Verification of uniquely named temporary datasets and deletion of them at termination;
- l. Judgment of flexibility to specify user define titles and footnotes;
- m. Review of SAS system options that are changed and reset at termination;
- n. Judgment of readability and layout of produced readable output (tables. listing);
- o. Readability of log files with regard to macro header, user feedback and macro generated error reports;
- p. Review of debugging options (partially visible in log file);
- q. List of called auxiliary macros, both internal (part of main macro), attached (outside, after main macro but part of macro file) and external (separate macros);
- r. Description of way(s) to call the macro from a main program;
- s. Final critics that should be made with regard to the macro;
- t. Final judgment about the macro, approved or disapproved (with reason(s)).

The final judgment may very much depend on whether the macro is capable of producing numerically correct results. It is as important whether the macro interferes with the data to be processed or not (unintentionally overwriting data by non-unique dataset or variable names). The way the macro results are output as dataset(s) or readable output (layout) is somewhat less important in the overall judgment of a macro. Robustness of arguments against erroneous parameters, measure of fool proof, is also rather important, but not essential.

22) ELECTRONIC SIGNATURE

Optionally sign the validated macro electronically (e.g. using PGP or similar software).

Next to the validation report, in which the validation process has been described and in which its successful result has been reported, there should preferably be some electronic indication as close as possible linked to a macro about its certification. Of course the macro header may report its validated and certified status, along with the name of the person who did the validation and the date, but to a distrusting user that may not be enough. Who guarantees such a user that the macro yet hasn't been changed since the validation, or that this is the macro version that has been certified? Using an associated, but separately stored electronic signature of the certifying person on that macro may provide sufficient additional trust about the status of the macro.

The electronic signature should be created from the macro source code by the certifying person using his own personal identification code (the private key). The user can verify the valid status of the signature by checking it with the macro source code against the known public key of that person, who himself should be trustworthy. A software system that supports these kinds of private and public signatures is the commercial PGP, or its free counterpart GPG. The signature file (MacroName.sas.sig) should preferably be kept in the same directory as the macro source code file (including possible auxiliary macros) and be accessible to users, who may check the signature at any time (using the concerning software). It is also possible to electronically sign a complete, compiled macro catalog file if validated.

23) OPTIONAL FEATURES

The following features are less important to macro validation, but may nevertheless add value to a macro:

NoTable

The NoTable macro argument is being used to specify the text (within quotes) that should be displayed on a page (below the titling) in case a table is lacking. This may be due to an empty dataset, especially after a where condition has been applied. It clearly displays an appropriate message in the output. The macro argument is specified as:

```
, NoTable =/* empty or */ "Dataset $Data empty, no table to present!" /* or */
/* "Dataset $Data where $Where is empty, no table!" */
/* Text to present with empty table, none if empty, 0 */
/* If text not between single quotes then $ char. */
/* replaces any argument by its parameter value */
```

and the code to process this argument later in the macro is:

```
DATA &XtabData (KEEP=&GrpList &ByGroups);
* Code to execute _before_ the dataset is being read;
%IF (%STR(&NoTable) NE ) %THEN %DO; /* only if NoTable parameter has been specified;
* Investigate whether 0 obs (after WHERE selection);
IF (_N_ EQ 1) THEN &WhereObs = 0; /* initialize;
CALL SYMPUT ("WhereObs", PUT(&WhereObs, BEST12.)); * for every observation;
%END;
SET &Data %IF (%QUOTE(&Where) NE ) %THEN (WHERE=(&Where)); ; /* read dataset;
* Remaining of data step will not be run if there are no observations;
%IF (%STR(&NoTable) NE ) %THEN &WhereObs + 1;; /* increment observation counter;
.....
```

The code above counts the number of observations resulting from the WHERE option. The NOBS dataset option can not be used for that purpose because it always returns the number of observations in the dataset at compile time, before the WHERE condition. The remaining processing and presentation code is:

```
/* ----- Process empty dataset -----;
%IF (%STR(&NoTable) NE ) %THEN %DO;
%IF (&WhereObs EQ 0) %THEN %DO;
%IF (%FirstCh(&NoTable) NE %STR(%')) %THEN
/* If no single quote then change $ to & in order to resolve the macro variable */
%LET NoTable = %QSYSFUNC ( TRANWRD ( %QUOTE(&NoTable), $, %STR(&) ));
DATA &Dummy;
Message = %UNQUOTE(&NoTable);
RUN;

PROC REPORT DATA=&Dummy; RUN;
%GOTO Exit;
%END;
%END;
```

CalledBy

The CalledBy macro argument is especially used with auxiliary macros being called from the main macro:

```
, CalledBy=&MacName /* Macro name of macro or program for error report 0 */
```

PhUSE 2006

Its purpose is to be given the name of the main macro calling it. In case the auxiliary macro writes an (error) report to the log it can be seen by which main macro (or program) it has been called when the event happened, because the auxiliary macro writes the CalledBy parameter value to the log as well. Instead of or next to a macro name (stored as the value of a macro variable MacName) any information may be given to it, e.g. a different value per call, to recognize the proper auxiliary macro call in the log reporting the event.

ForMyUse

The ForMyUse macro argument is intended for the author / developer of the macro only. Via that argument he is able to force specific actions, which are not used during normal operation. The only action that has been applied until now is to force compiling the macro by way of the parameter value Compile:

```
, ForMyUse=DO_NOT_USE /* argument for the author's use only, do not use it ! */  
and within the macro code:
```

```
%IF ( %UPCASE(&ForMyUse) = COMPILE ) %THEN %GOTO Finish;
```

This does nothing than skipping over almost all of the macro code after it has been loaded. The macro option STORE option still has to be set 'by hand', that is temporary edited into the source code (activating the commented out code) in order to force compiling.

Compiling / encrypting

A macro may be compiled. Compilation is not a guarantee against revealing the source code. Even with the debugging macro options a compiled macro may show the resolved macro and SAS code, though not the original source code. Embedding fixed options like NOMPRINT into the macro do not prevent breaking the code if one wants to. It would go far beyond the scope of the subject of validation to discuss the details of this kind of alleged security and (tricky) ways to break the security precautions here.

Since SAS version 9.2 there is the macro option SECURE, that offers real security to a compiled macro. It can be run, it may generate output and messages (PUT) to the log, but it does not show its code in any way, not even the resolved code. This means that quite some macro options have no use (like MPRINT).

Validation should be done on the original macro source code. A compiled macro is absolutely unacceptable for that purpose.

SAS names

It has already been discussed that the length of macro generated dataset and variable (and other SAS) names are dependent on the SAS version and the option VALIDVARNAME. It is possible to let a macro automatically check the current SAS version and the VALIDVARNAME option, and to find out which maximum lengths for these SAS names apply. Sample code for variable name and label lengths is:

```
/*~~~~~  
/* Adaptation of variable name and label length limits to SAS version (6,7,8,9);  
/*  
~~~~~  
%IF (%SUBSTR(&SYSVER, 1, 2) EQ %STR(6.)) %THEN  
%DO; /* vs. 6.xx;  
%LET MaxName = 8;  
%LET MaxLabel = 40;  
%END;  
%ELSE %IF (%SUBSTR(&SYSVER, 1, 2) EQ %STR(8.)  
OR %SUBSTR(&SYSVER, 1, 2) EQ %STR(9.)) %THEN  
%DO; /* vs. 8.x.y, 9.x.y;  
%LET ValidVar = %UPCASE(%SYSFUNC(GETOPTION(VALIDVARNAME)));  
%IF (&ValidVar EQ V6) %THEN  
%DO;  
%LET MaxName = 8;  
%LET MaxLabel = 40;  
%END;  
%ELSE /* V7 or UPCASE or ANY */  
%DO;  
%LET MaxName = 32;  
%LET MaxLabel = 256;  
%END;  
%END;  
%ELSE  
%DO; /* any other vs.;  
%LET MaxName = 8;  
%LET MaxLabel = 40;  
%END;
```

The MaxName and MaxLabel macro variables, which should have been declared LOCAL in advance, can be used later in the code to generate valid random SAS names and/or to check user parameters for legal lengths.

PhUSE 2006

As with many pieces of code that are present and do the same thing in many macros, a separate auxiliary macro checking the SAS version and its consequences for all kinds of SAS names can be written and called. A pilot macro SAS_name (still under construction) is available on request.

Recursion

Recursive macros may be written and work well, but the developer has to keep in mind that the maximum level of recursion is limited and depends on many circumstances of which the amount of values to store and the available internal computer memory are the most important ones. It varies from a few tens to a few hundreds of levels deep. Recursion should preferably be limited to not more than a few levels if it can not be avoided. A most elementary macro and a few macro calls to test the maximum level of recursion is:

```
%MACRO Recurse (Value); /* Returns a positive value;
  %PUT Value=&Value;
  %IF (&Value GE 0) %THEN %Recurse (%EVAL (&Value - 1));
%MEND Recurse;

%Recurse (10);
%Recurse (100);
```

This macro is as well an example of a principle that preferably should not be used in practice: solving a problem by looping through many recursion levels. During validation any recursive functioning should be observed and judged.

E. CONCLUSIONS

During the development and validation process the macro developer may or preferably should stick to quite a number of guidelines on several aspects of macro development. These have to do with the purpose of the macros, their user friendliness and last but not least their reliability. The validating person should verify the main issue whether the macro offers all the features that it pretends to offer. All kinds of aspects of a macro that are discussed before may be applicable and should be validated with success. The more aspects the better in order to be able to judge a macro as an entire piece of code with specific functionality. Only then a macro can be trusted to do what it pretends to do in a justified way. Only then a user may use a macro with sufficient confidence.

F. ACKNOWLEDGEMENTS

The contents of this paper are based on many years of experience with developing, testing and validating SAS code and macros and discussing all aspects of SAS programming with many people. I would like to thank everyone without naming them, especially on the internet distribution list SAS-L, or the newsgroup comp.soft-sys.sas .

CONTACT INFORMATION

Y. (Jim) Groeneveld
Statistician, SAS consultant
OCS Consulting Benelux
PO BOX 490
5240 AL ROSMALEN
THE NETHERLANDS
Office: +31 (0)73 523 6000
Fax.: +31 (0)73 523 6600
JimG@OCS-Consulting.com
www.ocs-consulting.com
home.hccnet.nl/jim.groeneveld