

SAS Macro Version Control

Jim Groeneveld, OCS Consulting, Rosmalen, the Netherlands.

ABSTRACT

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 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. In case a current macro version is not fully backwards compatible 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. A version control system within the macro code is presented which allows the user to specify which (compatible) version of a particular macro is to be used. If the current macro version is known not to be compatible with an older, specified version it refuses to run, generates an appropriate message and stops. The user then should take care for replacing it by the appropriate older macro version. Alternative solutions that don't stop the code from running and that allow forcing of a specific version of a macro to run, are being presented.

INTRODUCTION

CHANGE CONTROL

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 original SAS and macro code can be used for reproduction purposes. They may already have been validated in combination, just like other versions. This is the regulatory aspect of version control.

BACKWARD INCOMPATIBILITIES

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. There are, however, instances where a newer macro may not be fully backwards compatible. Examples of such instances are:

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

VERSION CONTROL

In case a current macro version is not fully backwards compatible it should not be used for reproduction purposes with SAS code that used to call the older macro version. Instead the older macro version has to be used. A version control system within the macro code is presented which allows the user to specify which (compatible) version of a particular macro is to be used. If the current macro version is known not to be compatible with an older, specified version it refuses to run, generates an appropriate message and stops. The user then should take care for replacing it by the appropriate older macro version. Alternative solutions, that don't stop the code from running and that allow forcing of a specific version of a macro to run, are being presented.

STOPPING VERSION

The version control system consists of an additional parameter Version in the macro call and of macro code interpreting the Version value. It has a default value of the current version number, allowing running it normally. If the user explicitly specifies an older, backwards incompatible version number it will not perform its normal action, but notify the user to use the older version. Possibly incompatible or at least different results are thus being avoided. If the user specifies another, but backwards compatible version number, the newest macro allows running itself. The code that interprets the version number is:

```
%IF (&Version NE AND &Version NE &MacroVs) %THEN /* if Version not default and not empty */
%DO;
  %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.

This simple method to prevent a macro from running at all (at least regarding the intended functionality) should at least be built into any general macro that isn't entirely backwards compatible with any previous version(s). It should also be used in combination with running alternatives, as discussed below.

RUNNING VERSIONS

An alternative action of an incompatible newer macro may be to reproduce the deviant behaviour of the older version in case the older version number is specified while calling the macro. Another alternative is to yet always, independent from the specified version number, allow currently undocumented, outdated parameter names and argument values as long as these can be interpreted unambiguously.

BACKWARD SUPPORT

The main and most logical method of supporting previous versions is to still support previous functionality, defaults, arguments and dependent on the specified version number perform the functionality of the older version(s). Validation is also necessary to check whether the previous functionality still works completely correct like before.

For example: an old macro version knew the outdated parameter name `SourceDs`, indicating an input dataset. In a newer macro version the parameter name has been changed to `Data`, indicating just the same. The old macro parameter was:

```
( SourceDs= _LAST_ /* Input data set specification R */
```

and the corresponding one in the newer macro is:

```
( Data = _LAST_ /* Input data set specification R */
```

while the backwards compatible parameter in the newer macro is:

```
, SourceDs= /* old version's Data , one specification allowed - */
```

Only one of the equivalent parameters may be used at a time, not both; that would induce discrepancies.

The internal macro code that processes both alternative parameters is:

```
%IF (&Data NE AND &Data NE _LAST_ AND &SourceDs NE) %THEN %DO;
  %PUT *** &MacName *** &ERR: Both old SourceDs and new Data names specified;
  %PUT . macro &MacName will abort;
  %LET ErrCount = %EVAL ( &ErrCount + 1);
%END;
%ELSE %IF (&SourceDs NE) %THEN %DO;
  %LET Data = &SourceDs; /* Data not specified;
  %PUT *** &MacName *** &Warn: Parameter name 'SourceDs' extinct, use 'Data';
%END;
```

The above example has been taken from macro `MR2RM.SAS`, version 2.7.3k. It has more similar backward support. See <http://home.hccnet.nl/jim.groeneveld/software/SASmacro/MR2RM.zip> (or with version number in its name).

At some time in the future, e.g. after 15 years, the currently required storage period for original data (after study termination) in clinical research, the support for outdated, old-fashioned, antique functionality may be removed in yet newer versions.

BACKWARD CALL

Instead of supporting backward functionality a newer macro should pass control to an older version that exists separately once that version has been specified in the macro call. In order to be able to do so the newer macro should support all older arguments anyway. It can pass control to the older macro by `%INCLUD`ing the older one or by calling it via the `AUTOCALL` facility. The older macro in that instance can not have the same name anymore, as a macro can not redefine itself. It should exist under a different name (e.g. original name combined with version number), whether or not located in the same directory.

Once the newer macro gets outdated by another yet newer one its backward calling functionality becomes redundant as that should be supported by the newest one always. Though the idea is good and would work well the method is complicated and makes re-editing the newer macro necessary once a still newer one has been developed. This method thus won't be worked out in more detail and will not be recommended. Instead this method serves as the basis for yet another method described below, with which each macro version only supports its own parameters and functionality at all times and which is much easier to maintain.

VERSION DISTRIBUTION

A version distribution system consists of a series of two or more independent backwards incompatible macros with their own parameters and supported functionality, each without any support for other versions. These macros, though actually different versions of the same macro should be given different file and macro names, e.g. the original name combined with a version number (consisting of legal file and macro name characters). The only version control these separate versions should have is the above described stopping support as a safeguard against running while not justified. Next to these and serving as the macro to be called, there is a version independent main macro with the name of the original macro.

The main macro supports all possible parameters of the newest and older versions, determines the version to be called and passes control over to that version (with its full macro call). That way any version can be called at any time. The main macro does not need to explicitly support all possible parameters and their values; it can suffice with the complete set of unnamed parameters using the PARMBUFF macro option. The whole user specified macro call is then stored within the (macro quoted) automatic macro variable SYSPBUFF (that can be redefined) that should be passed along to the appropriate macro version call. The advantage of the unnamed macro call is that unspecified user parameters are not passed on (while keeping their default values), while when applying named parameters one does not know which ones were specified by the user and which ones not. (A solution to that problem might be a unique default value of all parameters, like DEFAULT, that would signal unspecified parameters not to be passed on.) More advantages will become clear and will be summarized later.

A most simple basic example: suppose a macro called VERSION, version 1 consisting of (without stopping control here):

```
%MACRO Version (Text=Default text of macro Version1, Version=1);
  %PUT This is macro version 1 (One);
  %PUT &Text;
%MEND;
```

and suppose another macro called VERSION, version 2 consisting of (also without stopping control to keep it simple here):

```
%MACRO Version (Text=Default text of macro Version2, Version=2);
  %PUT This is macro version 2 (Two);
  %PUT &Text;
%MEND;
```

Rename both macro versions and their file names to resp. VERSION1 and VERSION2 and create a main macro VERSION:

```
%MACRO Version / PARMBUFF;
  %LOCAL I Argument Parameter Version Macro;
  %* Determine version number from user specified arguments;
  %IF (%LENGTH(&SYSPBUFF) LE 2) %THEN %LET SYSPBUFF = ; %* ();
  %ELSE %DO;
    %* Remove parentheses around macro arglist;
    %LET SYSPBUFF = %QSUBSTR(&SYSPBUFF,2,%LENGTH(&SYSPBUFF)-2);
    %LET I = 1;
    %LET Parameter = %QSCAN (&SYSPBUFF , &I, %STR(,));
    %DO %WHILE (%LENGTH(&Parameter));
      %LET Argument = %SCAN (&Parameter, 1, =);
      %IF (%UPCASE(&Argument) EQ VERSION) %THEN %DO;
        %LET Version = %SCAN (&Parameter, 2, =);
        %LET I = 9999; %* to jump out of %DO %WHILE loop;
      %END;
      %ELSE %LET I = %EVAL ( &I + 1 ); %* increment pointer by 1;
      %LET Parameter = %QSCAN (&SYSPBUFF , &I, %STR(,));
    %END;
  %END;
  %* Call appropriate macro version with complete user specified arguments;
  %IF (NOT %LENGTH(&Version)) %THEN %LET Version = 2; %* default;
  %LET Macro = Version&Version;
  %INCLUDE "&Macro..sas"; %* or use the AUTOCALL option;
  %&Macro (%UNQUOTE(&SYSPBUFF));
%MEND;
```

The main macro VERSION obtains the full user specified parameter call, incl. the parentheses around them via the automatic macro variable SYSPBUFF. The surrounding parentheses are being removed and the call is split into separate parameters with their specified argument values. These are scanned for the parameter name VERSION (in any case) which eventually yield the user specified version. If there is no Version parameter or the argument is empty the newest version number, known to the main macro serves as the default value. Dependent on the version number the appropriate macro version is called with the complete user specified argument list. The concerning macro is called with the list (%UNQUOTE (&SYSPBUFF)); without removing the parentheses the list might have been specified as %UNQUOTE (&SYSPBUFF), but that results in SAS errors as a macro definition requires explicit parentheses, not (delayed) resolved ones. The rest of the list and the macro name may be obtained from resolved macro variables as is demonstrated.

Instead of the code to “call appropriate macro version with complete user specified arguments” (below the comment) one may program somewhat more extensive code if the dependent macro version name can not be easily deduced from the original macro name and the version number, e.g. if the version number contains periods or other illegal characters for macro or file names. An example of somewhat more dedicated code, that needs to be extended once another, newer macro emerges, is:

```
%IF (&Version EQ 1) %THEN %DO;
  %INCLUDE 'Version1.sas'; %* or use the AUTOCALL option;
  %Version1 (%UNQUOTE(&SYSPBUFF));
%END;
```

```

%ELSE %IF (&Version EQ 2) %THEN %DO;
  %INCLUDE 'Version2.sas'; /* or use the AUTOCALL option;
  %Version2 (%UNQUOTE(&SYSPBUFF));
%END;

```

Finally here are some calls to various versions of the same macro, which may occur all of them within the same program:

```

%Version (Text=This is the transferred text2, Version=2);
%Version (Text=This is the transferred text1, Version=1);
%Version (Text=This is the transferred text0);
%Version ();
%Version;

```

Additional advantages of the version distribution system are:

- main macro should have name of macro that it drives the versions for;
- so one main version independent macro per set of different versions of the same macro;
- some intermediate versions may be mutually backwards compatible, only the main macro needs to “know” about that;
- when adding another backwards incompatible version the main macro only has to be slightly (or even not at all) adapted to support that version number;
- specified, but not existing or historically existent, but not involved version numbers may be interpreted as either the closest newer version or as a specification error to which the main macro should respond by refusing to pass control to any version of the macro;
- the user always gets the very newest (validated) version if (s)he does not specify a version number, with its new features and bugs fixed, that is not possible if the user would directly call an older macro version;
- some macro versions may be backwards compatible with regard to the immediately preceding version, thus it is not always necessary to call a specific version. Only a few different versions, with which their next newer version is not backwards compatible, but which themselves are backwards compatible with one or more preceding versions, need to be supported by the version distribution system. While a user might specify a version number that does not have a concrete macro version associated, the main distributing macro may give the user a newer version that is backwards compatible with the specified version.

Needless to say that the best method of version control, the best way to avoid change control is to avoid backwards incompatibility completely. Yet in such cases original macro versions, whether compiled or not, have as well to be archived together with dedicated SAS code calling the macros and with which they have been validated, to ensure program code integrity once the programs have to be rerun at a later stage.

OTHER SYSTEMS

Another simple version control system that is often applied, consists of macro versions with always different names (involving version numbers), just like with this version distribution system, which, however, are called directly from the dedicated application program by the user. The version distribution system can be regarded as a general extension to such systems.

There may be more version control systems invented, some of them using fixed version numbers for specified macros (while the system described here uses dynamically specifiable version numbers in individual macro calls). Some of these may apply AUTOCALLED search directories with a specified search order. In that case older versions of more macros in those directories actually are determined as a related set of macro versions.

REFERENCES

SAS Institute Inc. 2004. Base SAS® 9.1.3 Procedures Guide. Cary, NC: SAS Institute Inc.

This paper is a more detailed elaboration of the paragraph “Version numbering and control” in: Jim Groeneveld, SAS macro validation criteria. PhUSE 2006, Dublin, October 9-11, 2006, <http://www.lexjansen.com/phuse/2006/ra/ra04.pdf>

CONTACT INFORMATION

Y. (Jim) Groeneveld
OCS Consulting Benelux
PO BOX 490
5240 AL ROSMALEN
THE NETHERLANDS
Office: +31/0 73 523 6000
Mobile: +31/0 650 880 934
Fax.: +31/0 73 523 6600
Jim.Groeneveld@OCS-Consulting.com
www.ocs-consulting.com
home.hccnet.nl/jim.groeneveld