Paper 047-29

Guidelines on Writing SAS® Macros for Public Use

Frank Ivis, Canadian Institute for Health Information, Toronto, Ontario, Canada

ABSTRACT

The usefulness of SAS macros is without question. But when macros are shared with others, their value increases immeasurably. Consider the time saved when one programmer makes available a solution to an entire team. The effectiveness of such a macro does not simply depend on whether it accomplishes its intended goal, but on how well the macro can be understood and used. If it is inadequately documented or difficult to use, its utility is limited, regardless of how powerful or inventive the code is. Poor design often stems from the fact that the author originally intended the macro for personal use only and did not consider the issues involved when making it publicly available. The purpose of this paper is to describe some guidelines for writing good public macros, i.e. macros designed and intended for use by multiple users. This paper is aimed at the beginning to intermediate SAS user who has some familiarity with the macro language.

INTRODUCTION

The macro language is undoubtedly one of the most useful and powerful components of the SAS System. Although ostensibly a method of text substitution, macros can be used in an infinite number of ways across various modules of the SAS System. From simplifying repetitive tasks to complex self-contained applications, the macro language knows no bounds. Its utility and flexibility are key features. Once you become adept with the macro language, you find that it is usually much more efficient to create a macro for any complex piece of code that will likely be used more than once in different situations. More importantly, a properly written macro can also be shared with colleagues, be they in your own office or other parts of the world. Much has been written on the logic and syntax of macros. Less frequently discussed, however, are the necessary attributes of a public macro versus one intended only for private use. I define a public macro as any macro that the original author makes available to other users. Numerous issues arise with public macros because the context of their use is removed. For example, public macros may be used with various operating systems, some of which may be quite different from what the original author had intended. Potential users may also have varying levels of experience with using macros, so clarity and proper documentation is vital. In most cases, you should be able to use any public macro with a minimum of fuss. Ultimately, the goal is to ensure that public macros can be used by all who need them. The most creative, powerful macro will not be of much use to anyone if it is poorly documented and complicated to use.

The purpose of this paper is to summarize some key components that should be included in a macro intended for public use. Many of these techniques are not new. In some cases, they are simply adaptations of good programming practice (see McConnell, 1993). However, based on my experience, public macros are often lacking in certain attributes. This may be due to a number of reasons. If my early experience with macros is at all generalizable, it may simply be due to the fact that beginning programmers are often satisfied when a macro simply works, and do not concern themselves with questions of robustness and usability. Furthermore, like others, I learned much from the examples in books and the internet, where the focus is typically on the syntax and logic of the code. Appreciation of the more subtle design issues is usually acquired gradually with experience. The guidelines that I present are simply suggestions that I have found useful in the past several years of using and writing public macros. They may not be appropriate for all situations. At a minimum, I hope that these guidelines make you reconsider how you write public macros.

DOCUMENTATION

Although documentation is important for any type of program, it becomes especially critical in public macros. The potential user needs to know what the macro does and how to invoke it properly. To begin with, a documentation template is very helpful in providing all essential details in a standard format. Styles and level of detail will vary, but information about the author, purpose, scope and design of the macro should be included. The following is an adaptation of a format suggested by Whitney (1996).

```
/*************************************************************************

Macro name:       In addition to the file name, include the full path of the
                  directory in which it resides.

Written by:       Your name, as well as relevant contact information (email,
                  telephone number).

Creation date:    The date of original program.
```

```
As of date:        The date of the most recent revision.

SAS version:       The version and platform used to write the macro.

Purpose:           Clearly and concisely describe what the macro actually does.

Method:            Describe the steps necessary to achieve the purpose. Additional
                   comments may also be included in the code.

Format:            List the components of the actual macro call. For example:

                   %print_obs (data =
                             , keep =
                             , nobs =
                               )

                   An added benefit of this is that users can simply cut and paste
                   the above code when they invoke the macro.

Required
Parameters:        Define all the required parameters needed for the macro
                   to run properly, e.g.

                   data - Name of SAS data set.

                   keep - List of variables that you want printed.

Optional
Parameters:       Define all the optional parameters and any default settings.
                   For example:

                   nobs - Number of observations to be printed. Default is 50.

Sub-macros
called:            Are any additional macros called within the current macro. If so,
                   list them and their purpose.

Data sets
created:           List the library and name(s) of any data sets created by the
                   macro, if applicable.

Limitations:       Discuss any limitations of the macro.

Notes:             References, background information or rationale of the macro.

History:           List bug fixes, coding changes, etc.

Sample Macro
call:              What the actual macro call would look like, e.g.

                   %print_obs (data =mydata
                            , keep = var1 var7
                            , nobs = 100
                              )

                   You may also want to consider including sample data/output to
                   fully demonstrate how the macro should work.

*******************************************************************************/
```

The above headings were listed separately for clarity. In actual practice, they could be combined or subsumed under larger headings where appropriate. Topics that are not applicable can simply be omitted.

Although the above comments are useful in describing the purpose and design of the macro, you should also strive to make the actual macro code self-documenting. This can be accomplished by using clear, descriptive naming conventions. To start with, the name of the macro should give a clear indication of what it does. For a macro that creates a listing of duplicate records, use a name like PRINT_DUPS rather than CHECKDATA or DUPLICATES. Descriptive keyword parameters also aid in understanding the macro. I find that naming the macro parameter after the SAS statement or option to which it corresponds to be particularly useful. For example, I always use DATA for the macro parameter relating to the SAS data set to be read in and OUT for any created data sets. Other common examples include KEEP, WHERE and NOBS. Whichever naming system you adopt, make sure that it is consistent. The macros will be easier to use if they share a common naming convention. Lastly, use descriptive names in the actual macro code. This will help you in understanding and debugging the code at a later date. For example, instead of using generic names like I or J as counters, use a name that describes its purpose, such as YEARS or VISITS.

Another consideration when documenting macros is the use of options and commands that print information to the LOG. There are three macro options that relate to documentation: MPRINT, MLOGIC and SYMBOLGEN. The first, MPRINT, will generate the SAS code created by the macro. MLOGIC is most useful for evaluating any conditional logic or %DO loop processing. SYMBOLGEN will print the resolution of all macro variables every time they are resolved. Of the three options, I typically use only MPRINT, so that the user will have a record in their LOG of what the macro did. The other two options often create too much information, which detracts from the readability of the LOG. They are most useful when creating and debugging macros. A good alternative, however, is the judicious use of %PUT statements. In addition to printing specific macro variables and their values, other options include _USER_ , _LOCAL_ , _GLOBAL_ , _AUTOMATIC_ and _ALL_. Additional statements can also be included for clarity. For example, to confirm the user-defined macro variables, you could use:

```
%put You defined the following macro variables:  ;
%put _user_ ;
```

There are many other possible uses for %PUT statements such as explanatory notes, listing of default options or warnings. In addition to the above suggestions, you can also insert additional comments to document the actual macro code, where appropriate. In addition to the */* message */* style, the *%* message ;* style is specific to macros. Do not use ** message ;* ,which is intended for open code.

For very large or complex macros, additional supporting documentation may also be required. In such cases, it is usually preferable to create a separate document, either some type of text or html file. Typically, these types of documents go into detail regarding the theory, algorithm, usage or output. Remember to include a reference to this document in the macro. If you have created a collection of macros, include a *readme* file that summarizes what each one does and how to call them. This is particularly important if you are unsure of who the potential users are or their familiarity with macros.

ERROR CAPTURE

Any macro will produce errors if it is not invoked properly. How you deal with these errors can make a big difference in terms of usability. On one hand, you can take a passive approach and let the macro processor run its course, generating its own error messages. On the other hand, you can take a more active approach and write your own error capture code to catch common errors. This approach is preferable for two reasons. First, it allows you to write clearer error messages or warnings that are more helpful in debugging than the often cryptic messages generated by SAS. New users unfamiliar with your macro will especially appreciate this. Second, you can avoid any unnecessary processing by terminating the macro as soon as an error is encountered. This is especially useful when working with large data sets or intensive computations. The first part of the macro may require considerable processing, but may bomb later on due to a missing parameter value.

Although it is difficult to predict and capture every type of error that a user may make, there are a number of standard checks that are relevant to many macros. One is to check whether all the required macro parameters are specified. This can be accomplished with the macro function %LENGTH. For example:

```
%if %length(&data) = 0 %then %do ;
    %put ERROR: Value for macro parameter DATA is missing ;
    %goto finish ;
%end ;
/* other macro statements */

%finish:
```

By definition, a macro variable with a length of 0 is null. If this is the case, the above code will print an error message and end processing. The judicious use of a %GOTO statement will neatly avoid the rest of the macro code and quietly end the program. Error processing is a good example where %GOTO statements can be quite useful (McConnell,1993). Note that when you begin your message with ERROR, the entire phrase will appear in red in the LOG.

Another good check is to verify whether a required entity exists. For example, if an input data set is a required parameter, its existence will be necessary for the macro to execute. Combining the EXIST function with %SYSFUNC makes it easy:

```
%let error = 0 ;
%if %sysfunc(exist(&data)) = 0 %then %do ;
    %put ERROR: data set &data does not exist ;
    %let error = 1 ;
%end ;

/*  other error checking code */

%if  &error = 1 %then %goto finish ;

/* macro code */

%finish:
```

The above code takes a slightly different approach to error processing. A macro variable, ERROR, is used to keep track of any errors. Only at the end, when all errors are checked, is the variable evaluated.

The examples presented so far are rather simple. However, depending on the macro, error capture can get rather complex. Other checks might examine whether a variable is character or whether a libname or format is a valid SAS name. If you begin to notice that your macros share the same type of code, consider creating error checking sub-macros. These are short utility macros that can be defined within another macro. For example, you could define a standard sub-macro to check for the existence of a data set using the code above. This will ensure consistency and significantly reduce the amount of visible code in the macro. The important point is to capture the most common errors and provide clear explanations of them.

Another useful approach to reducing errors is to allow for some variation in the way that the values for macro parameters are specified. For example, rather than depend on the user to specify values in uppercase, it is easier for the macro itself to do it. This is accomplished simply with the %UPCASE function. The example below ensures that the value of DEBUG is uppercase, regardless of how it is entered.

```
%let debug = %upcase(&debug) ;
```

If you prefer, you could also extend the functionality by accepting a whole word, even when only a single character is required. In the following example, the value of the macro variable DEBUG should be either *Y* or *N*. But a *Yes* or *no* could also be accommodated.

```
%let debug = %upcase( %substr(&debug,1,1 ) ) ;
```

Obviously, you will want to avoid this practice if there is any possibility of ambiguity or confusion with other parameters. Incidentally, this is similar to the convention SAS uses to accept misspelled commands that are close to correct syntax. For example, I sometimes mistype DATA as DATDA in DATA steps or procedures as in the following example.

```
datda test ;
   a = 1 ;
run ;
```

This DATA step will still be processed, but with the following message in the LOG :

```
WARNING 14-169: Assuming the symbol DATA was misspelled as datda.
```

DO NO HARM : REDUCING THE POTENTIAL FOR UNINTENDED CONSEQUENCES

A macro may not always work exactly the way a user would like, but at the very least, it should not interfere with other programs. There are a number of ways in which this could possibly occur. If your macro creates a data set in the course of its operation, there is the possibility that it could overwrite an existing data set of the user. One common convention is to use an original, descriptive data set name proceeded with an underscore. Although not entirely foolproof, it greatly reduces the chances of using a common data set name. To be completely secure, one could check for the existence of the data set before proceeding. If such a conflict were detected, the macro could be stopped and the user alerted of the issue. Any temporary data sets created by the macro should also be deleted as soon as they are not required. In addition to freeing-up disk space, it will also reduce the chance for file conflicts.

One of the trickier problems concerns conflicts between global macro variables. Briefly, global macro variables are available to the user at any time during the session, including open code. Local variables, on the other hand, exist only for the duration of the macro in which they are created. The key, therefore, to avoiding conflicts is to use only local variables in your macros. A short example will illustrate the problem.

Suppose a user has the following assignment statement in open code:

```
%let x = 5 ;
```

And then calls your macro, CHECK:

```
%macro check ;
    %* %local x ;
    %let x = 1 ;
%mend check ;

%check
%put x = &x ;
```

To the user's dismay, the global macro variable X has been changed from 5 to 1. This occurs because during macro execution, the macro processor checks whether X already exists in the global symbol table. Since it does, it updates it accordingly. You can see the difference by uncommenting the above code so that macro variable X is defined as local, using the %LOCAL statement. To be safe, all variables created within a macro should be defined as local, including those derived from CALL SYMPUT, the INTO: clause of PROC SQL and index variables used in %DO ...%END loops. If you must use global variables, note them in the documentation, use unique names (with underscores) and/or test whether they are pre-existing (using the new %SYMEXIST function, available in SAS 9.1). Consult SAS documentation for more details on the rules governing local and global macro variables.

In the event that your macro changes any system options, be sure to return them to their original setting before the macro ends. This is easy to do with the GETOPTION function. For example, I like to use the MPRINT option in my macros. To preserve the user's original setting, I use:

```
%macro test ;
%local option ;
%let option = %sysfunc( getoption( mprint, keyword ) ) ;
options mprint ;

/* macro code */

options &option ;
%mend test ;
```

The original setting, either MPRINT or NOMPRINT, is captured by the macro variable OPTION. It is then reinstated before exiting the macro.

Lastly, if your macro uses any titles or footnotes, be sure to turn them off before ending the macro. Simply using the TITLE or FOOTNOTE statement alone will do the trick. For precise control of titles and footnotes, the dictionary table DICTIONARY.TITLES (or SASHELP.VTITLE) allows you the possibility to restore previously defined titles that may be changed by the macro. For example, before changing a title within your macro, you could use the following code below. First, assume that TITLE1 is already defined by the user. It does not have to be, but it will illustrate the point more clearly.

```
title1 'Original title';

/* Next, within the macro, capture the title number and text of the title. */

%macro test ;
proc sql noprint;
    select number, text into: num ,:text
    from dictionary.titles
    where number = 1;
quit ;
%let num = &num;
%let text = &text ;

/* The above %let statements are simply a convenient method for trimming the macro
variables. Next comes the new title defined in the macro. */

title1 'Macro title';

/* But before existing macro, restore the original title: */

title&num "&text";
%mend test;
```

EASE OF USE

The macro facility has a number of different options that impact how a macro is invoked. Some are more user friendly than others. In the case of macro parameters, there are two styles: keyword and positional. The difference is that keyword uses an = after the parameter name when the macro is defined. Consider the following macro called RECODE, which categorizes values into distinct groups. Both methods, positional and keyword, are demonstrated below.

```
/* positional */ %macro recode (data, out, var, groups );

/* keyword    */ %macro recode (data =, out =, var =, groups = );
```

However, the important difference is apparent when the macro is invoked:

```
/* positional */  %recode(sales, salesr, revenue, 5)

/* keyword    */  %recode(data = sales, out = salesr, var = revenue, groups = 5 )
```

The keyword style offers several advantages over the positional style. As is demonstrated from the above example, keywords are self-documenting. It is clear that SALES is a data set and REVENUE is a variable and not vice versa. There is little chance of error in assigning the wrong value, unlike with positional parameters, where the order is important yet there are no visual clues as to the parameter. An additional benefit of keywords is that they allow for the specification of default values. If for example, five groups were considered a reasonable value for most situations, one could simply revise the initial macro statement to:

```
%macro recode(data=, out=, var, groups = 5) ;
```

By doing this, only the first three parameters would be required to invoke the macro, assuming the default value was acceptable.

One practice that you definitely want to avoid is having the user edit the macro and change values in %LET statements. There are several serious problems with this. First, this practice precludes the possibility of having a truly public macro since it would no longer be possible to store the macro in a single location. Changing %LET statements could have disastrous consequences for other users. The other problem is simply the necessity of editing the macro, which could result in errors or unintended changes.

Regardless of the style used, one should try to keep the number of parameters to a minimum. Too many parameters can make a macro difficult to use. How many is too many? It is hard to say, as it will often depend on the nature and purpose of the macro. As a very rough rule of thumb, you may want to reconsider the design if there are more than seven required parameters. If you feel that too many parameters are making your macro difficult to use, there are a few things you can do. First, check with users or potential users about the functionality that they require. Perhaps some of the original parameters that you originally envisioned do not add appreciably to the usefulness of the macro. If all the parameters are deemed useful, consider setting some reasonable defaults so as to reduce the number of required parameters. This allows the user to get started quickly with the macro without having to fuss over all the options. Many of the SAS Procedures are good examples of this. Few PROCs actually require more than three or four mandatory parameters, yet they are extremely powerful. Of course, there are usually numerous options for the users with specific needs. I find the same approach also works well with macros. Another useful practice is to imbed comments within the macro call to remind the user of what they represent. For example:

```
%print_vars (data =    /* name of input data set */
          ,   var =    /* list of variables to print */
          ,  nobs =    /* number of observations */
          , byvar =    /* name of by variable */
             )
```

One simple way to spare users unnecessary grief is to avoid the use of statement-style macros. In addition to being rarely used, making them unfamiliar to the majority of SAS users, they are difficult to identify precisely because they lack the % sign. They also slow processing due to the extra resources required to scan for the macro names (SAS Institute Inc., 1999). Consider the following short example:

```
options implmac ;

data test ;
    a = 1 ;
run;

%macro printdat(data) / stmt ;
    proc print data = &data ;
    run;
%mend printdat ;

printdat test ;
```

Within the context of a larger program (and without the actual macro code embedded), it is not immediately obvious, especially for inexperienced users, that the last line above is invoking a macro. It is safer to stick with the name-style (i.e. using the % sign).

TRADEOFFS AND FINAL CONSIDERATIONS

When creating public macros, you should consider potential tradeoffs in design that will affect functionality and usability. This can often occur due to differences in versions of SAS or components installed. For example, Version 9 of SAS contains many new functions, commands and procedure options. However, it does not make much sense to use them in your macros if some of your users are still using Version 8. In most cases, existing solutions are preferable whenever possible, even if they take longer to code. For example, the following is a new command available in Version 9:

```
call symputx ('count', count) ;
```

It will automatically left justify and trim a macro variable. However, the standard solution is compatible at least since version 6:

```
call symput ('count', trim(left(put(count,8.)))) ;
```

If you are used to having access to all components of SAS, be aware that others may not have this luxury. SAS/STAT®, SAS/GRAPH®, SAS/QC®, SAS/IML®, SAS/ETS®, SAS/OR®, among others, are all separate components. Selecting random samples is now easy with PROC SURVEYSELECT, for example. But this procedure is only available in SAS/STAT. In this situation, it may be preferable to use only the functionality of Base SAS to

write a macro to draw random samples if you think that some of your users may not have access to SAS/STAT. Of course, in many cases, the specialized procedures available in certain components are critical for a particular type of analysis. Just be sure to alert the user of the requirements when documenting the macro.

Another important issue is that of robustness. Writing a comprehensive macro that can be used in any environment, under any circumstances is a worthwhile goal. However, incorporating such portability and robustness can require considerable time and increase the complexity of the macro. Ultimately, the correct approach will vary depending on the situation. This issue can often crop up in some unexpected ways. Hamilton(2001), for example, has demonstrated that calculating the number of observations in a data set is far from a simple task. The NOBS= option on the SET statement will only work properly on a simple, unedited SAS data set. It does not work with edited data sets, views or external files. Consequently, if you plan to write a macro that in part requires the number of observations in a data set, you must consider the context in which the macro will be used and whether the added robustness will be required. If you decide that it is not practical for the macro to handle all potential situations, be sure to list the limitations in the documentation.

Some final suggestions:

- Maintain consistency in documentation standards, naming conventions, macro variables, etc. The particular style or conventions that you use are generally less important than implementing them consistently. This is particularly important if more than one person is writing public macros within the same group or section. Try to agree on a common style template so that all macros, regardless of the author, have a similar look and feel.

- Test your macro thoroughly before release, not simply for bugs, but for functionality and efficiency. In one case, I used PROC FREQ in one part of a macro to count values and it appeared to work fine during testing. However, when I eventually used it on a file with several million records, the macro bombed. I subsequently re-wrote that section using PROC SQL. Try to get others to help you with testing.

- If you know who your users might be in advance, consult with them before, during and after creation of the macro. This helps to ensure the macro has the required functionality. There is no point in creating macros for people who will never use them.

- Store public macros in a common, read-only drive. This ensures no inadvertent changes are made and that everyone uses the most current version of the macro.

- Review your macros periodically. As your knowledge of the macro language increases, you may find new ways to improve and fine-tune those old macros.

RECOMMENDED RESOURCES

Given the current space limitations of this paper, many of the topics were discussed only briefly. For additional information, consult the following resources:

- *SAS Macro Language: Reference* (1999) - One needs to be familiar with the intricacies of the macro language in order to implement many of the ideas presented in the paper. Of particular interest is Chapter 11, Writing Efficient and Portable Macros, which goes into some detail about host-specific issues - an important consideration if your macros are to work properly across different platforms.

- *Code Complete: A Practical Handbook of Software Construction* (1993) by Steve McConnell. This is an excellent reference to best practices related to software design, applicable to any language. Among the many topics discussed are error processing, documentation, local vs. global variables and naming conventions.

- http://support.sas.com - Check here for many examples of macros incorporating the guidelines discussed in this paper. The statistics section has a particularly good collection of more complex macros.

- As might be expected, the internet is a good resource for SAS macros of varying quality. In addition to picking up new techniques, it is often useful to critically evaluate other macros. Try to spot weaknesses in the design and devise improvements or alternative solutions.

CONCLUSION

This paper began with a discussion of public macros and went on to outline a number of important topics including thorough documentation, error capture, eliminating unintended consequences and usability. The work required in implementing these ideas may seem daunting at first, but will ultimately save time and effort in the end.

Although this paper makes a distinction between public and private macros, there is no reason why these ideas cannot be applied to any macro, regardless of the intended audience. Any large, complex project, even if it involves a single programmer, can run into problems if macros are not carefully designed and documented. Even for the simplest macros, proper design and documentation force you to think carefully about the underlying logic and construction of your program. This in turn will prepare you for more complex macros in the future and ultimately make you a better programmer.

REFERENCES

Hamilton, J. (2001). *How Many Observations Are In My Data Set?* Paper presented at the SAS Users Group International 26, Long Beach, CA.

McConnell, S. (1993). *Code Complete: A Practical Handbook of Software Construction*. Redmond: Microsoft Press.

SAS Institute Inc. (1990). *SAS Guide to Macro Processing, Version 6.* (2nd ed.). Cary: SAS Institute Inc.

SAS Institute Inc. (1999). *SAS Macro Language: Reference Version 8.* Cary, NC: SAS Institute Inc.

Witney, C. M. (1996). Taming the Chaos: A Primer on the Software Life Cycle and Programming Standards. *Observations*, Fourth Quarter, 15-21.

ACKNOWLEDGEMENTS

I would like to thank Michael Mace and Helen Carey for their helpful comments on an earlier draft of this paper.

CONTACT INFORMATION

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

Frank Ivis
Canadian Institute for Health Information
90 Eglinton Avenue East, Suite 300
Toronto, Ontario M4P 2Y3
Canada
Work Phone: 416 481 - 2002
Fax: 416 481 - 2950
Email: fivis@cihi.ca
Web: www.cihi.ca