Paper CT10

Standardised SAS Macro Considerations for Submission-Ready Code

Will Greenway, Plus-Project Partnership, Leeds, UK

ABSTRACT

Standardised macros can be great but can often overload a user's log. Not all macro comments and processing need to be shown to a user where only the end result really matters. Giving users what they need, when they need it, can result in better quality macros, downstream processes and clearer logging.

This paper is based on learnings from a "submission-ready code" process which outputs resolved code (without macro calls). The original suite of standardised macros generated resolved code containing meaningless processing steps and orphaned comments that were at best irrelevant, and at worst confusing or misleading. I intend to cover my experience with various SAS options and methods for controlling log and MFILE output, debugging considerations, and the impact of different SAS comment types, all in the context of "submission-ready" code. And, as always, there are hidden obstacles along the way.

INTRODUCTION

During the process of submission to various agencies, programs used to create key deliverables are often required. The FDA's Study Data Technical Conformance Guide [1] states:

> *"Sponsors should provide the software programs used to create all ADaM datasets and generate tables and figures associated with primary and secondary efficacy analyses. … Sponsors should submit software programs in ASCII text format. Executable file extensions should not be used."*

The main purpose, per the same document, of requesting the submission of these programs is to understand the process by which the variables for the respective analyses were created and to confirm the analysis algorithms and results.

One of the common obstacles faced when submitting SAS® programs is that there is often heavy use of macros within the programs. These macros are often proprietary code stored in a central location to support in-house processes across a range of studies and compounds. As such, they can be quite complex to ensure that they are flexible enough to deal with a range of scenarios.

There are multiple ways that these programs can be made available, including submitting the macro code, however, as mentioned these are often proprietary, sometimes with considerable time and effort to create and not something that companies often like to share. Another obstacle is that again, being often complex programs, sharing the code alone might not really help a reviewer understand which parts of a macro actually run for a particular output. Another alternative is sharing secured macro catalogues, but although it means the programs could be theoretically run, macro catalogues often do not transfer across computing environments easily, and as the code is hidden, it may still not be obvious what processing is taking place.

As sharing the macro code or catalogues is often not ideal, another option often taken is to create "Submission-ready" programs. These are the original programs, but with macro calls replaced with the resolved macro code. There are again a few ways to create these and some extra thought needs to go into the macro development to ensure code is useful and easily understandable. This paper will discuss these considerations and offer some best practices for global macro development.

CREATING SUBMISSION-READY CODE

One of the most common methods for creating SAS programs with resolved code is the use of SAS options MFILE [2]. Using this option requires a few steps to setup and then to create the final resolved code.

First, a `filename` statement should be written with a set fileref of `mprint` pointing at the output file name and location, for example, and noting that per the FDA requirement executable file extensions (e.g. ".sas" which may be setup to automatically run in some batch systems) should not be used:

```
filename mprint "&programloc.\submission_ready_code\adsl_sas.txt";
```

Once the file reference for `mprint` has been created, the SAS options `mprint` and `mfile` should be activated:

```
options mprint mfile;
```

The `mprint` option is often used in debugging SAS macros and allows SAS to send resolved macro code to the SAS log window for the user to see easier what code was actually submitted by the macro processor for running, while the `mfile` option is used here to also direct this resolved code to the referenced file `mprint` created earlier.

One of the limitations of the `mfile` options to be aware of is that it will only output resolved macro code. Therefore, if you have macro calls interspersed between non-macro code, only the macro code will be output. To get around this, placing the entire code within a macro will ensure that all code is output to the same resolved code.

That is, change from something like the below where some of the statements are in open code, with macro steps in between:

```
data adsl_1;
  ...;
run;

%popn(in=adsl_1, out=adsl_2);

data adsl_3;
  ...;
run;
```

To a program where all code is within a single overall macro call, or series of macro steps with no open code in between:

```
%macro adsl;
  data adsl_1;
    ...;
  run;

  %popn(in=adsl_1, out=adsl_2);

  data adsl_3;
    ...;
  run;
%mend adsl;
%adsl;
```

This will ensure that all code associated with this output is directed to the referenced file `mprint`, not just the code for some steps. There are many statistical computing environments that already use a process of driver and macro processing, where a top-level program (a driver), will always run an autoexec or setup program followed by various macros/scripts for each output. This type of system design works well when using SAS and outputting resolved code using `mfile` as all code is already inside one or more macros called by the driver program, with no open code that would be left out.

The final step, once your program is entirely contained within one or more macros, is to run the program. Ideally, if the above steps are proactively followed prior to a final run, then the resolved code can be generated at the same time as the final outputs. However, if this is needed retrospectively then extra caution should be taken not to overwrite previously generated outputs and logs. Additionally, some lines generated by macros, such as a date/time stamp at the bottom of a table, will be different when generating the submission-ready code retrospectively, rather than at the time of the final run.

TIDYING UP

Using `mfile` to produce the resolved code can unfortunately be a little untidy, for example not following good programming practice of indentation of code. I would therefore always advise some post-processing to adjust code indentation and ensure the program is readable.

As a quick solution, various versions of SAS Enterprise Guide have a "format code" option that can be quickly utilized to re-indent code and these can be customized in the Enhanced Editor Options menus. However, if this is needed for a large number of programs then investing time in creating a method to post-process these files using other methods may be easier in the long run.

Aside from this easy-to-spot issue, there may be other more subtle issues with the resolved code that it is good to be aware of. For example, some important macro processing may be hidden from view, or code that isn't really needed could be spamming the output, generating needless lines of code that were only used for internal macro processing. The below section discusses some of these other considerations and some good programming practices for global macros in general.

SAS OPTIONS

SAS options are often very important in the processing of macros. Having different options for `validvarname` for example, can determine what comes out of a **`proc transpose`**. Other options such as `notes`, `mprint` and `source`, which will be discussed later, impact lines sent to the logs, output and also the `mprint` file. As such, during the processing of a macro, options may be adjusted away from the default, or those set by a user, to ensure the macro runs as expected and, as you'll see later, to only show the user what is relevant.

However, as options settings remain until changed, adjusting options can lead to issues or impacts on other programs if options are not correctly reset. Extra care should therefore always be taken to return the options settings back to how they were prior to the macro executing. There are multiple ways to do this and I will discuss two of them below:

RE-SETTING OPTIONS – METHOD 1) PROC OPTSAVE & PROC OPTLOAD

One of the easiest ways to reset all options in one go is to simply save the settings of all options at the start of a macro call, and then read that back in later to reset all options to the same settings.

To save the settings of all options at the start of processing, the **`proc optsave`** procedure can be used, which will create a SAS dataset (in the below example called __options) containing the settings of all SAS options:

```
proc optsave out=__options;
run;
```

Once the macro is ending and in the "wrap up" phase, the counterpart **`proc optload`** procedure can be used to read in the previously saved SAS dataset and reset all options.

```
proc optload data=__options;
run;
```

The main disadvantages of this method are that the SAS options need to be saved at the starting settings, so, if `mprint` was already activated then the **`proc optsave`** line statements will appear in the final resolved code. To avoid this by specifically setting `options nomprint` prior to this procedure would mean that the setting of `mprint` prior to running the macro has been lost and would require other methods to reset.

However, as a catch-all for macros that might change several SAS options, this trade-off of having the procedure in the final resolved code isn't too bad.

RE-SETTING OPTIONS – METHOD 2) %SYSFUNC(GETOPTION( ))

A slightly more focused method of resetting SAS options is by creating local SAS macro variables with values created from `%sysfunc(getoption( ))` [3] for example:

```
%let optnotes=%sysfunc(getoption(notes));
```

This will create macro variables `optnotes` macro variable with the value of the option `notes`, i.e. values of `notes` or `nonotes`. This macro variable can then be used to reset the option whenever you need, e.g.

```
options &optnotes;
```

You can add as many of these `options` as you like into one macro variable to store and reset multiple settings at the same time, for example:

```
%let optSettings = %sysfunc(getoption(notes)) %sysfunc(getoption(source))
                   %sysfunc(getoption(mlogic)) %sysfunc(getoption(symbolgen))
                   %sysfunc(getoption(mprint));
```

Although the above is generally sufficient, there are some options where the value returned by `getoption` by default is not suitable for use within the `options` statement as-is. There are numerous ways around this but one of the simplest is to use the `keyword` argument added to `getoption`, for example considering the commonly used linesize (`ls`) option:

```
%let optls1=%sysfunc(getoption(ls));
%let optls2=%sysfunc(getoption(ls,keyword));
%put &optls1;
%put &optls2;
```

SAS Log:

```
30         %put &optls1;
132
31         %put &optls2;
LS=132
```

You can see above that `&optls1` which was created using the default behaviour of `getoption` generated just the value of `ls` (which was 132 at the time of running) and would be no good if put inside a `options` statement. Whereas the value for `&optls2` which used the `keyword` argument generated a statement that could be used directly in an options statement to reset the value of `ls`. Specifying `LS=` when either setting or resetting the value of `ls` can also be done but this may complicate things if you have a mix of options. Also if you use a macro variable named the same as the option (in this case `ls`) then you can also use `&=ls` but that may not always be feasible.

Although this method of resetting options requires more work to re-set all options that may have been adjusted, it does mean that select options could be set and re-set independently throughout the program, rather than Method 1 which would reset all options each time.

Whichever method is chosen, care does however need to be taken to ensure that sub-macros using the same methods do not override each other's previous settings. This can be done by defining the scope of the macro variables using `%local` or defining macro-specific names for the macro variables or options dataset.

OPTIONS NOTES

Ensuring the SAS options for `notes` is active during most processing is essential as it can show when steps executed, how long they took (if you are looking for efficiency gains), and other statements of interest. However, writing notes to the log is time-consuming for SAS, and when considering complex global macros, often there is some processing specific to the inner workings of the macro itself that may not be useful for a user to know. Therefore in some circumstances switching between `notes` and `nonotes` during various steps can not only save time in processing, but it may also be used to hide unimportant lines from the user and let them focus on what they need to know.

Using an example that I presented at a previous PHUSE EU Connect [4] of a macro to read in XLSX files by treating them as zipped XML files (which they are), is it useful for a person running the macro to know that the macro extracted 4 XML files that make up an XLSX and the detail of how it stitched these together? Or is it enough to simply know the end result, i.e. how many variables and observations were ultimately read and what dataset was created. Using the `notes` option sensibly we can remove the noise out of the SAS log and present the user with only the key details, for example:

```
%readxlsx(xlsxfile=&labloc.\lab.xlsx, outdset=lab);
```

SAS Log:

```
NOTE: [READXLSX] Reading: \\projects\project\study\data\labs\lab.xlsx.
NOTE: [READXLSX] The data set WORK.LAB has 370 observations and 22 variables.
```

This was done by saving the value of the `notes` option at the start of the code, then setting `options nonotes` for most of the processing. At the end, once a final dataset has been created, `options notes` is set in order to output the NOTE: messages shown above informing the user about the successful (or not) creation of a dataset. Finally the value of the `notes` option is reset to its value at the start of the macro. Note that setting `nonotes` will not hide SAS `ERROR`s or `WARNING`s, so alongside good error handling rules issues encountered can still be flagged to a user for fixing, with context around them as needed. However, careful consideration is still needed when supressing messages from the log to ensure that there are no hidden issues being supressed.

OPTIONS MPRINT

While `notes` controls output to the log, `mprint` is its equivalent for output to the `mprint` file and can be used in much the same way.

Consider the scenario of a macro that runs over all datasets in library. There are again multiple ways to derive this, for example within **`proc sql`** querying the `dictionary.tables`, or a user may have manually written each one. Exactly **how** this list of datasets is generated may not be important, but simply **what** the contents of the list is, i.e. which datasets it ran over. Using a simplified example below of a macro for reading all SDTM datasets into work. The macro may look something like this:

```
%macro sdtm2wrk;

  proc sql noprint;
    select memname into :dset1-
      from dictionary.tables
      where libname='SDTM';
    %let lastsdtm=&sqlobs;
  quit;

  %do d=1 %to &lastsdtm;
    data &&dset&d;
      set sdtm.&&dset&d;
    run;
  %end;

%mend sdtm2wrk;
%sdtm2wrk;
```

mprint:

```
proc sql noprint;
select memname into :dset1- from dictionary.tables where libname='SDTM';
quit;
data AE;
set sdtm.AE;
run;
data CM;
set sdtm.CM;
run;
...
```

In the resulting mprint file above, the section of code highlighted in yellow is no longer really needed in our final resolved code. If this resolved code was run, the highlighted code would not actually do anything since the macro variables dset1 and onwards have already been resolved when creating the loop of data steps below it. As such, this highlight code section is now redundant. If this step was more complex it could also take some considerable time working out to determine what it is and what it is doing, which isn't needed to understand the actual output produced.

Instead sensible setting of mprint and nomprint options around relevant parts of the macro can result in the final mprint file reduced to only lines of real use:

```
%macro sdtm2wrk;
  %let optmprint=%sysfunc(getoption(mprint));
  options nomprint;

  proc sql noprint;
    select memname into :dset1-
      from dictionary.tables
      where libname='SDTM';
    %let lastsdtm=&sqlobs;
  quit;

  options mprint;

  %do d=1 %to &lastsdtm;
    data &&dset&d;
      set sdtm.&&dset&d;
    run;
  %end;

  options &optmprint;
%mend sdtm2wrk;
%sdtm2wrk;
```

mprint:

```
options nomprint;
data AE;
set sdtm.AE;
run;
data CM;
set sdtm.CM;
run;
...
options MPRINT;
```

Setting nomprint around such macro processing prevents showing the internal workings of **how** the lines were generated, while setting mprint during the processing of each dataset in the loop allows the user to still see which datasets were processed, which is the important part.

However, one further consideration when switching between mprint and nomprint is that when these options are set, SAS has a delay of 1 "word" before starting/stopping the output. This can mean something as mundane as extra unnecessary semi-colons, or it could mean code that would error upon running. Take the following example, where initially the following options are active: options nomprint nonotes nosource

```
%macro mprint;

  ** Toggle to show processing **;
  options mprint notes;

%mend mprint;
%mprint;
```

mprint:

```
;
```

During the macro processing `mprint` was activated and the next word (`notes`) was skipped but the rest of the line after that (containing just a semi-colon) was printed. This is annoying, but not a major error. However, to see how this might become more important see another example below where more than one additional option is being set:

```
%macro mprint;

  ** Toggle to show processing **;
  options mprint notes source;

%mend mprint;
%mprint;
```

mprint:

```
source;
```

The output again skipped the next word (`notes`) but again the rest of the remaining line did not skip, so the word `source` and onwards is now appearing in our code. If this resolved code was now run we would likely get an `ERROR: Statement is not valid or it is used out of proper order.` in our SAS log, which is now more important to resolve.

The simple solution for this is to set `mprint` at then end of the options line:

```
%macro mprint;

  ** Toggle to show processing **;
  options notes source mprint;

%mend mprint;
%mprint;
```

mprint:

Here, `mprint` has been put as the final option to be set. The word after that, the semi-colon, has been ignored and as that is the end of the statement line, nothing from this line is actually output to the **`mprint`** file, which is ideally what we would want. Therefore when toggling `mprint` active, I would suggest always adding this as the last option to be set which will avoid anything, including a stray semi-colon being sent to the **`mprint`** file.

The same is true for `nomprint`, for an example where the following options are initially active: `options mprint notes source`, consider the following code:

```
%macro mprint;

  ** Toggle to hide processing **;
  options nomprint nonotes;

%mend mprint;
%mprint;
```

mprint:

```
** Toggle to hide processing **;
options nomprint nonotes
```

Note that the `options nomprint` is shown here regardless, as is the next word after `nomprint` which in this example is `nonotes` however the words after that (just a semi-colon) have **not** been included in the output. This can cause issues when running this code as the `options` statement has not been closed with a semi-colon.

Instead, putting `nomprint` as the final option to be set, the word after that (the semi-colon), has been included in the **`mprint`** file and so, although we ideally wouldn't want this line to show at all, at least it is a complete `options` statement that will not error if we run the resolved code.

```
%macro mprint;

  ** Toggle to hide processing **;
  options nonotes nomprint;

%mend mprint;
%mprint;
```

`mprint:`

```
** Toggle to hide processing **;
options nonotes nomprint;
```

COMMENTING

Similar to options, when considering macro code within the context of submission-ready code, the type of comment you use may actually impact more than you realise. You may wonder why SAS has a few options for SAS comments, and although there are a few reasons, their interaction with the SAS macro facility is one of the reasons why one comment would be better than others. I will only consider the 3 recommended types of comments below, ignoring others.

/* BLOCK OR IN-LINE COMMENTS */

Comments starting with `/*` and ending with `*/` are the most common form of comment used. They can be any length and a single comment can span multiple lines containing semicolons and unmatched quotation marks. Their main advantage is that they can be used within statements or anywhere a single blank can appear in SAS code, however one drawback is that they cannot be nested, i.e. a comment started with `/*` will end at the next occurrence of `*/`, regardless of whether sub-comments using `/*` were started inbetween. For example:

```
/* /* Nested comment 1 */ Nested comment 2 */
```

Nested comment 2 in the above example is shown as being outside of any comment.

When used with `mprint` and `mfile` they do **not** appear in the final code. Therefore, these may be useful for commenting macro processing (e.g. adding a comment to explain a `%do` loop or `%if` condition, or other code that is purely to support macro processing, such as those that create macro variables for use later in the code), however this comment form should not be used to comment programming lines that should appear in the final resolved program (the **`mprint`** file).

* STATEMENT COMMENTS;

Comments starting with `*` and ending with `;` are the other common method of commenting used. Again, they can be any length, however each comment must be it's own separate statement and they cannot contain internal semi-colons. Additionally a macro statement or macro variable reference that is contained inside this form of comment will be processed by the SAS macro facility.

When used with `mprint` and `mfile` they **do** appear in the final code. Therefore, these may be useful for commenting programming lines that will end up in the final resolved program, but should not be used for commenting macro processing, otherwise the final code will have a lot of meaningless comments.

%* MACRO STATEMENT COMMENTS;

Comments starting with `%*` and ending with `;` are a half-way house between the above two types of comments. They are similar to statement comments, except, like block or in-line comments, they are hidden from the macro processor. Therefore, these may be useful for commenting macro processing, however this comment form should not be used to comment programming lines that will end up in the final resolved program.

SUMMARY OF COMMENT FORMS

To demonstrate these comments, here is a program that uses all 3 types mentioned above, and the output in the final program.

```
%macro comments;

  /* Block-comment */
  * Statement Comment;
  %* Macro Statement Comment;

%mend comments;
%comments;
```

mprint:

```
* Statement Comment;
```

As demonstrated above, only the Statement Comment form made it through to the final resolved code.

Comment Type	Commenting Macro Processing (e.g. `%do` loop or `%if` condition)	Commenting Resolved Code (code output to the `mprint` file)
`/* Block-comment */`	✓	✗
`* Statement Comment;`	✗	✓
`%* Macro Statement Comment;`	✓	✗

DEBUGGING

Debugging is another area that can be impacted when considering the information above. If you hide too much from a user, how can they work out what is going wrong to fix it? Therefore any macros that start hiding things from a user should have some good debugging options in effect.

DEBUG PARAMETER

I would always recommend for complicated macros to include a `debug` macro parameter that accepts values of `Y` or `N` as input, defaulting at `N`. This parameter can then be used within the start of the program to activate, or prevent deactivation, of options such as `mprint`, `mlogic` and `notes`.

PREVENTING NOTE/MPRINT FROM DEACTIVATING DURING PROCESSING

There may also be parts of your code that for toggle between settings, for example `mprint` and `nomprint`, or `notes` and `nonotes`. Setting a macro variable to the value of either `mprint` or `nomprint` based on your debugging macro variable can ensure that a macro doesn't mistakenly change the `mprint` setting part way into a program to show, or hide, what it wasn't meant to. This is a code example:

```
%if %upcase(%substr(&debug.,1,1))=Y %then %do;
  %* For DEBUG=Y notes are never turned back off **;
  %let nonotes=notes;
  %* If DEBUG is on, show code, notes and macro logic **;
  options mlogic symbolgen notes source mprint;
  %put NOTE: [READXLSX] Debugging mode is active.;
%end;
%else %do;
  %* For DEBUG=N notes are always turned back off **;
  %let nonotes=nonotes;
  %* If DEBUG is off, hide code, (most) notes and macro logic **;
  options nomlogic nosymbolgen nonotes nosource nomprint;
%end;
```

later in the code...

```
options notes;
%put NOTE: [READXLSX] The data set &in. has &obs. observations and &vars. variables;
options &nonotes;
```

ADDING EXTRA NOTES

Another good way of adding good debugging options is to add additional output to the SAS Log when you debug mode is activated, or preventing a complete clean-up of temporary datasets controlled via the `debug` macro parameter:

```
%* While in debug mode, inform user of how many Sheets the XLSX file contained **;
%if &debug=Y %then %do;
  options notes;
  %put NOTE: [READXLSX] There are &lastsheet. sheet(s) found in &infile..;
  options &nonotes;
%end;

...

%* Remove all the temporary datasets **;
%if &debug=N %then %do;
  proc datasets lib=work nolist nodetails;
    delete __xlsx:;
  quit;
%end;
```

CHECKING MACRO PRE-REQUISITES

Another way to help with debugging is to check any pre-requisites of the macro in advance. For example, if a macro variable should contain the name of an input dataset then before the macro starts its main processing, code can be added to check that the macro variable is not blank, and its value is the name of a dataset that exists.

```
%macro prereq(indset=);
  %* Check INDSET value is a valid SAS dataset **;
  %if %nrbquote(&indset)= %then %do;
    %put %str(ERR)OR: [READXLSX] Required Macro Parameter INDSET is blank.);
    %goto macend;
  %end;
  %else %if not %sysfunc(exist(&indset)) %then %do;
    %put %str(ERR)OR: [READXLSX] Dataset used by INDSET does not exist: &=indset..);
    %goto macend;
  %end;

... /* Main macro processing goes here */ ...

  %macend:
  %* Add some macro clean up steps before exiting the macro **;
%mend prereq;
```

These types of checks prior to the main macro allow you to check for errors in a controlled way before they happen, flag the issue to the user using a message that should be meaningful to them, and also to exit the macro again in a controlled way that for example could re-set options, or clean up any temporary datasets created.

CONCLUSION

Using `mprint` part-way into an `options` statement or deciding which is the most suitable comment type is not something that we often need to put a great deal of thought into during everyday programming. However, understanding how these interact with the macro processor can greatly improve the flexibility of any macro that may be used to generate submission-ready code.

Many programmers will be familiar with writing complex SAS macros, however when considering that these may contribute towards submission-ready code produced via `mprint` and `mfile` there are a host of additional considerations that should be taken into account to ensure that any resolved code used for a submission requires minimal changes post-production, is (theoretically) executable, or syntactically correct, and as clear as possible to ensure the main processing can be understood by a reviewer.

In the project I was involved with, existing macros were generating hundreds of lines of useless comments, random extra semi-colons, and lines that would cause errors if executed. Considerable time was devoted to tidying these up for a submission. It is hoped that, armed with the above knowledge, future macros can instead be developed in such a way that it is easy to produce submission-ready code seamlessly.

Having a process in place to ensure that any resolved code actually runs and creates the same outputs as the original should also be a key consideration and would catch a lot of these issues, however this is outside of the scope of this particular paper.

REFERENCES

[1] The FDA's Study Data Technical Conformance Guide, March 2022
https://www.fda.gov/regulatory-information/search-fda-guidance-documents/study-data-technical-conformance-guide-technical-specifications-document

[2] SAS® 9.4 and SAS® Viya® 3.5 Programming Documentation - MFILE Macro System Option
https://documentation.sas.com/doc/en/pgmsascdc/9.4_3.5/mcrolref/p0j3zpinabl49ln13q5f5r1t46mg.htm

[3] SAS® 9.4 and SAS® Viya® 3.5 Programming Documentation - GETOPTION Function
https://documentation.sas.com/doc/en/pgmsascdc/v_031/lefunctionsref/n096vc8shtf7mzn1rlvx6003ayho.htm

[4] CT06: No XSLX Engine in SAS? No XLSX Problem, PHUSE EU Connect 2021, Will Greenway
https://www.lexjansen.com/phuse/2021/ct/PRE_CT06.pdf

ACKNOWLEDGMENTS

I would like to acknowledge and thank Elizabeth (Liz) Meeson, and Tony Cooper for their review of this paper and the associated presentation.

CONTACT INFORMATION

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

Will Greenway
Plus-Project Partnership
Suite L3CA, No1,
Booths Park,
Chelford Rd,
Knutsford,
Cheshire,
WA16 8GS

Email: will.greenway@plus-project.co.uk
LinkedIn: https://www.linkedin.com/in/willgreenway/
Web: https://www.plus-project.co.uk/