SC14

Controlling Macro Output or, "What Happens in the Macro, Stays in the Macro"

Frank DiIorio, CodeCrafters, Inc., Chapel Hill, NC

ABSTRACT

One of the hallmarks of a well-designed macro is that it creates only those artifacts that it was supposed to. That is, once it terminates, the only changes to the SAS environment are the macro variables, data sets, files, *et al.* that were identified in the macro's documentation. Macros that create unwanted data sets, reset system options, assign or deallocate librarires and the like are annoying at least and even worse, potentially harmful to the calling environment.

This paper describes some tools and programming techniques that can be easily adopted to prevent unwanted changes and in turn lead to better, more reliable macro design.

INTRODUCTION

Imagine the following apocryphal-yet-realistic scenario. Rather than reinvent a particular programming "wheel," you use a generalized program, or "utility," to perform the task. The utility creates the data set that you expected, but it also makes unexpected and unwanted changes in the SAS environment: options are reset; macro variables and data sets are created; and titles and footnotes are changed. Clearly the utility was helpful, but equally clearly, it left a bit of a mess in its wake.

The kindest interpretation of the utility's behavior sees these unexpected items as a minor annoyance. It isn't hard to make the leap from annoying to problematic and harmful. The utility could have overwritten `WORK` or other data sets, reset options that subsequent code expected, and so on.

Clearly, the criterion for a useful utility should be to create what it promised. Equally important, the utility should *not* alter the SAS environment in any other way.

This paper will outline some simple, effective techniques for ensuring that utilities written in the SAS macro language are well-behaved. We demonstrate the techniques by presenting a poorly written macro, describing some design principles, then performing a code "makeover" that results in a revised functional, well-behaved utility.

PART I: BAD, SLOPPY CODE

Data to some regulatory agencies such as the FDA may be in the form of SAS data sets meeting certain criteria: no user formats, use specific variable names and data types, etc. Our example utility creates two items: a format that identifies values as SAS or user-written formats, and a macro variable that contains the count of user-written formats.

The output format can then be used by data cleaning and quality assurance programs. The initial Version 9 program follows. (A discussion of the `SASHELP` views of SAS metadata is interesting, but beyond the scope of this paper. See "Recommended Reading," below, for resources).

```
%macro userFormat;
       data temp;
            fmtname = '$ufmt';
            set sashelp.vformat(where=(fmtype='F')) end=eof;
            start = fmtName;
            if source = 'C' then do;
               label = '_U_';
               nUser + 1;
               end;
               else if source in ('U', 'B') then label = '_S_';
               else                              label = '?';
            if eof then call symputn('nUser', nUser);
       run;

       options nocenter;
       title "$UFMT, identifying &nUser. User-written formats";
       proc format cntlin=temp fmtlib;
       run;
%mend;
```

The macro does what was expected of it, and creates format $UFMT. It can be used as shown below:

```
… DATA steps, PROCs, OPTIONS statements …
%userFormat;
data errorListing;
     set proj.metadata;
     if put(upCase(format, $ufmt.)) = '_U_' then do;
        set message text, write to ERRORLISTING
        end;
```

As we noted in the Introduction, the macro's ease of use comes with a price. Specifically:

- If data set TEMP existed prior to the call to %userFormat, it will be overwritten.
- Data set TEMP is not needed once PROC FORMAT runs, yet it remains in the WORK directory once the macro terminates.
- Macro variable nUser may replace/overwrite a like-named global macro variable.
- The CENTER option is set to NOCENTER, possibly changing the setting prior to the macro call.
- The title text is set, thus clearing the entire title "stack" (TITLE1 through TITLE10) in effect prior to the macro call.
- If the OBS option in effect before the macro call is less than the number of observations in SASHELP.VFORMAT, PROC SQL will not be able to read to the end of the view, and will produce incorrect results.

It's important to realize that the only items we wanted from the program were the format and the macro variable. Anything else (data sets, options, title) that changes is unexpected, unwanted, and indicates bad design. How, then, to control the macro's impact on the calling program's environment? How do we ensure – to paraphrase the Las Vegas ads – that "what happens in the macro, stays in the macro?"

These problems stem from both a lack of control and communication. The programmer has the ability to control the attributes (name, scope, etc.) of any item in the macro. Likewise, the programmer has the obligation to communicate (via comments and other documentation) to the user what, exactly, transpires in the macro (the program's user is not held harmless here, and must *read* the documentation in order to avoid surprises). This section broadly describes the principles of control and communication. The principles are put into practice in the next section ("Part II: Good, Clean Code").

DESIGN PRINCIPLES: CONTROL

Options. Macros often require specific option settings (OBS, XWAIT, etc.) Rather than set them, and possibly changing their values prior to the macro's invocation, it's better to capture the settings at the start of the macro, then reset them before macro termination. Tools to accomplish this include PROC OPTSAVE and PROC OPTLOAD for more complex applications and the GETOPTION function for smaller, simpler grabs of a small number of options. The generalized use of this technique is shown below:

```
%macro example;
       %local optSettings;
       %let optSettings = %sysfunc(getoption(obs, keyword))
                          %sysfunc(getoption(center, keyword));
       options obs=max noCenter;
       … other macro statements …
       options &optSettings.;
%mend;
```

Scope. All variables created by the macro should be explicitly scoped as global or local. This avoids unintended insertion or replacement of a value into the calling program's macro table. Consider the macros defined in the following example:

```
%macro m1;
       %do i = 1 %to &n.;
           %m2
       %end;
%mend;
```

```
%macro m2;
       %do i = 1 %to 10;
           … macro statements …
       %end;
%mend;
```

The example is, of course, contrived, but illustrates the problem: variable I is referenced in both %M1 and %M2. Results are unpredictable, but likely dire, in %M1 because %M2 is continually resetting I. The problem is solved by making both I's scope local, as shown below. Note, too, that more appropriate naming would also address this problem.

```
%macro m1;
       %local i;
       %do i = 1 %to &n.;
           %m2
       %end;
%mend;
```

```
%macro m2;
       %local i;
       %do i = 1 %to 10;
           … macro statements …
       %end;
%mend;
```

Naming. Collisions between data sets, macro variable, and other entity names in "caller" and "called" macros is a problem that is both pervasive and easy to deal with. Simply prefix items with a string that is unique to the macro. This is shown in the before and after program fragments below:

Before (no prefixes)	After (use prefixes)

`%let count = %dscount(data=t1);`
`data t2;`
`  set master.t1;`
	`%let _PR_count = %dscount(data=t1);`
	`data _PR_t2;`
	`  set master.t1;`

Macro variables and temporary data set names unique to the macro are readily identified as those beginning with `_PR_`. This technique, of course, requires coordination among programmers writing macros in a library, so as not to chose duplicate prefixes. Ultimately, it's a small matter to attend to, and it pays off handsomely.

Cleanup. In most cases, the macro should end with a series of commands that remove items that are not expected or used by the calling program. These items include data sets, macro variables, temporary files, `FILENAMEs`, and `LIBNAMEs`. One approach is shown below:

```
%symdel _PR_count _PR_list _PR_listQ n;
proc delete data=_PR_t1 _PR_cnltin _PR_optvals;
run;
filename _PR_temp;
```

It's reasonable to say "if these names are unique and not likely to be referenced by the calling program, why not let them be deleted at the end of the program? `FILENAME`s will be cleared, `WORK` data sets will be deleted, and global macro variables will likewise disappear." This is true, but does not address the larger issue of the macro's impact on the calling environment. For example: disk space is unnecessarily used by the data sets; counts of data sets would be inflated because the calling program was not aware of the macro's mechanics.

Another, more powerful, technique requires some development effort. A utility macro could allow wildcarding of data sets and macro variables. Rather than manually specify items in `%SYMDEL` and `PROC DELETE` statements, we can use a single call:

```
%delItems(macvar=_PR_* n, data=_PR_*);
```

The exact syntax of `%delItems` is beyond the scope of this paper. It is, however, simple to write: parse each parameter's value, identifying wildcarded values (ending in `*`), building lists using SAS Dictionary Tables, and then creating `%SYMDEL` and `PROC DELETE` statements as needed. Notice the power of consistent naming in this example: simply specifying a root value of `_PR_` in the macro calls will remove all macro variables and data sets beginning with `_PR_`. You don't have to know each value. All you *do* have to know is that you want all these items deleted.

Design. A certain predictability of macro structure is desirable, particularly when you have to maintain a system of potentially dozens of macros. One general structure that readily supports the design hints noted above is shown below:

```
/* header describes macro purpose, inputs, outputs, usage notes, examples,
   maintenance history */
%MACRO statement
```

- Keyword parameters control, among other features, naming of output entities.

```
Initialization
```

- Capture current values of options that will be re-set
- Initialize global macro variables produced by the macro
- Check validity of parameters. Branch to bottom of the macro if there are problems.

```
"Core" processing
```

- Perform the core tasks of the macro, creating macro variables, data sets, etc as needed. Branch to the bottom of the macro if a condition arises that prevents further execution (e.g., an intermediate data set has no observations).

Termination

- Reset options to their original values
- Delete global macro variables that are no longer required
- Delete data sets, clear LIBNAMEs and FILENAMEs as needed
- Optionally, write messages to the SAS Log identifying items created by the macro.

DESIGN PRINCIPLES: COMMUNICATION

The importance of communication with the macro user cannot be overemphasized. This takes several forms, and in several locations in the program. The macro header is a communication of programmer intent, while run-time messages communicate what actually happened when the macro was used.

Macro Header. A well-written macro header can be used as both user documentation and programming reference. The text should include, at minimum:

- Description of macro parameters (name, permitted values, interactions with other parameters, whether required or optional);
- Outputs (macro variables: name and expected values; data sets: name, variable name, type, length, content) ;
- Examples of use;
- Usage notes: comments on processing, including entity prefixes (if used), whether the macro can/must be used in open code; items that are changed by the macro and cannot be reset. (See the notes in the macro header in the next section's example).

Run-Time Messages. Simple utilities may require little or no communication to the user or calling program. A macro that needs to start or end a timer has few "moving parts," and may not require Log messages. However, a complex utility that combines data sources and reports results using many formatting options has many "moving parts" and, in turn, more places where things can go wrong. `%PUT`, `PUT`, and `PUTLOG` statements can be used to inform the user of the macro's activites. Messages can inform the user about:

- Checkpoints. *"Successfully read all required metadata data sets" "Begin processing 23 transport files"*
- Results. *"Created global macro variables COUNT, LIST, and LISTQ"*
- Unexpected conditions "*Could not locate variable COMMENTS in data set VALUELEVEL. It will be omitted in the report."*
- Error conditions. *"Subsetting PROD.FY2006 resulted in 0 observations. Execution terminating."*

One can argue for the inclusion or exclusion of all these types of messages, save for, perhaps, error reporting. In the final analysis, it is probably most prudent to allow the user to decide whether to see messages. One simple and very effective method for doing this is shown below:

```
%macro buildMst(data=, out=, print=yes);
       %local _p;
       %if %upcase(&print.) = NO %then %let _p = *;
       %&_p.put Use var/dataset prefix _BLD_;
       … statements omitted …
       %&_p.put Created intermediate data set _BLD_PRETRANS;
       … statements omitted …
       data _BLD_tran2;
            set _BLD_pretrans nobs=n;
            &_p.put "NOBS in _BLD_pretrans = " n;
       … statements omitted …
%mend;
```

Local macro variable `_P` defaults to null. If `PRINT=NO`, `_P` is set to `*`. Its insertion into `%PUT` and `PUT` statements turns the statement into a comment if `PRINT=NO`. That is,

```
       %&_p.put Use var/dataset prefix _BLD_;
```

becomes:

```
       %*put Use var/dataset prefix _BLD_;
```

at run time. The macro writes a *comment* statement for SAS to execute, rather than a *`%PUT`* statement.

PART II: GOOD, CLEAN CODE

Pretty Good, Clean Code. With the above in mind, let's take a first pass at rewriting our example macro (shading identifies new statements):

```
* … some header content omitted …
  Output: Format $UFMT, mapping upper-cased format names
                  (including $ as needed, but not including
                  width and decimal point).  Mapped values are:
                  _S_ (SAS System format)
                  _U_ (User format)
                  ?   (Not recognized as a format name)
          Macro variable NUSER: contains number of user formats
          Contents of format library, written to default output
                destination (Output window, LST file, etc.)
    Notes: Overwrites current title
          Uses prefix _UF_ for macro variables and data sets
;
%macro userFormat;
       %local opts;
       %let opts = %sysfunc(getOption(obs)) %sysfunc(getOption(center));
       options nocenter obs=max;
       %global nUser;

       data _UF_temp;
            fmtname = '$ufmt';
            set sashelp.vformat(where=(fmtype='F')) end=eof;
            start = fmtName;
            if source = 'C' then do;
               label = '_U_';
               nUser + 1;
               end;
               else if source in ('U', 'B') then label = '_S_';
               else                              label = '?';
            if eof then call symputX('nUser', nUser);
       run;
       %put &nUser. user formats were found.;

       options nocenter;
       title "$UFMT, identifying &nUser. User-written formats";
       proc format cntlin=_UF_temp fmtlib;
       run;

       options &opts.;
       proc delete data=_UF_temp;
       run;
%mend;
```

Youll note that the revised macro uses the design suggestions from the previous section. Specifically:

- The header comment clearly describes the intended outputs (the format and the macro variable). The "Notes" section identifies the prefix used for data sets and macro variables. It also lets the user know that there is one item that can't be reset upon termination: the title used by `PROC FORMAT` will overwrite all titles in effect when the macro was called[1].
- The scope of macro variables `NUSER` and `OPTS` is controlled by `%LOCAL` and `%GLOBAL` statements.
- Values of the `OBS` and `CENTER` options are stored in variable OPTS, then altered, then reset to their original values at the bottom of the program.
- The temporary data set used by `PROC FORMAT` is given a unique prefix to avoid overwriting. The data set is deleted at the end of the macro
- A message to the Log lets the user know how many user formats were found.

[1] Readers familiar with SAS Dictionary Tables may rightly suggest that titles and footnotes could be saved, then restored by using the `TITLES` table. Since the table stores only text and not rendering information – height, color, justification – this approach is not reliable.

***Really* Good, Clean Code.** And then, of course, things change as a result of user feedback, data issues not revealed during testing, and the like. Let's imagine a few modifications, adding keyword parameters to the macro to allow the user to specify the names of the format and macro variable (shading identifies new statements).

```
* … some header content omitted …
  Input:  FMTNAME  Name of the output format. Must follow the
                   naming rules of SAS character formats.
                   Optional.  Default is $USERFMT
          COUNTER  Name of global macro variable containing the
                   count of user-written formats.
                   Optional.  Default is NUSER.
  Output: Format $UFMT, mapping upper-cased format names
                 (including $ as needed, but not including
                 width and decimal point).  Mapped values are:
                 _S_ (SAS System format)
                 _U_ (User format)
                 ?   (Not recognized as a format name)
          Macro variable NUSER: contains number of user formats
          Contents of format library, written to default output
                destination (Output window, LST file, etc.)
   Notes: Overwrites current title
          Uses prefix _UF_ for macro variables and data sets
          Does not create the format or macro variable if FMTNAME
             and/or COUNTER are syntactically invalid.
;
%macro userFormat(fmtname=$userfmt, counter=nuser);
       %put Create format &fmtName. and macro variable &counter.;
       %local opts;
       %let opts = %sysfunc(getOption(obs)) %sysfunc(getOption(center));
       options nocenter obs=max;
       … checks for validity of FMTNAME, COUNTER go here …
       %if parameterErrors %then %do;
           %put Format and macro variable will **not** be created.;
           %goto bottom;
           %end;
       %global &counter.;

       data _UF_temp;
            fmtname = "&fmtName.";
            set sashelp.vformat(where=(fmtype='F')) end=eof;
            start = fmtName;
            if source = 'C' then do;
               label = '_U_';
               nUser + 1;
               end;
               else if source in ('U', 'B') then label = '_S_';
               else                              label = '?';
            if eof then call symputX("&counter.", nUser);
       run;
       %put &counter. user formats were found.;

       options nocenter;
       title "&fmtName., identifying &&&counter. User-written formats";
       proc format cntlin=_UF_temp fmtlib;
       run;

       %bottom: options &opts.;
                %if %sysfunc(exist(_UF_temp) %then %do;
                    proc delete data=_UF_temp;
                    run;
                    %end;
%mend;
```

The revised program is better for the programmer using the utility. It provides more control over naming, thus completely eliminating unintended overwriting of items in the calling environment. Notice, too, that it creates a fair amount of work for the programmer:

- The header comment describes the parameters and the impact of their misspecification.

- The program must ensure that the parameters' values are valid. If they are not, the program branches to the termination section. The earlier version of the program simply flowed from top to bottom. Since the new version considers the possibility of interrupted flow, we must insert a label (`%bottom`) to jump to. Notice, too, that we print a message saying both the format and macro variable will not be created.
- Termination becomes more complicated. Since `_UF_temp` might not have been created due to parameter errors, we must verify its presence before deleting it.

The extra code is worth the effort. The macro produces the intended outputs and carefully cleans up after itself, resetting system options and conditionally deleting the temporary data set used by `PROC FORMAT`.

EXTENSIONS

SAS software contains few built-in diagnostic tools. This puts development effort in the hands of the user community. While a detailed description of these tools is beyond the scope of this paper, it is worthwhile to take a quick look at what might help the macro developer. The functionality of the tools gives some insight into factors that need to be considered when developing a library of generalized programs. The short list of desirable utilities might include:

- Delete macro variables. Test to see if the macro variable exists before attempting deletion. Allow wildcarding.
- Delete data sets. Test to see if the data set exists before attempting deletion. Allow wildcarding. An alternative design is to call the macro at the beginning of a program or macro, then at the end, delete all `WORK` data sets created between calls, with an optional `KEEP` parameter.
- Check a value for syntactical correctness as a macro variable, data set name, or format. Create a macro variable with a value of 1 if valid, 0 otherwise.
- Capture, then restore a set of system options. Call the macro at the start of a program, passing the list of options to save, then call again at macro termination, restoring the original settings.
- For development: identify changes in the SAS environment made between calls to a "what changed" macro. The first call to the macro uses SAS Dictionary tables to save lists of data sets, macro variables, catalog entries, etc. The next call to the macro captures the same information, then compares it to the values at the beginning of the macro. The well-behaved macro will change nothing except what was described in its header comments.
- For documentation: read the headers of all macros in a library (directory, PDS, etc.) and write them to a plain text file and/or HTML page. This places documentation for all utilities in a single location.

CONCLUSION

The viability of a macro, along with and the credibility of its programmer, is diminished when it makes undocumented changes to the calling program's environment. This paper describes a set of simple practices that give both the programmer and user greater control over what is produced by the macro. The list presented here is hardly complete, but should give the macro developer a starting point for methods to control what goes on in the macro and what leaves its grasp.

ACKNOWLEDGEMENTS

I would like to thank Laurie McLennan of Rho, Inc. for her comments on an early and less-readable version of this paper. As always, you, the reader, are having an easier time dealing with my writing thanks to my wife, April "The English Major" Sansom.

And lest we forget: SAS and all other SAS Institute Inc. product or service names are registered trademarks or trademarks of SAS Institute Inc. in the USA and other countries. ® indicates USA registration.

CONTACT

Your comments are valued and encouraged. Please direct correspondence to the author at Frank@CodeCraftersInc.com.

RECOMMENDED READING

Books and Articles

Carpenter, Art, "Carpenter's Complete Guide to the SAS Macro Language, Second Edition," SAS Press, Cary NC, 2004.

DiIorio, Frank and Jeff Abolafia, "Dictionary Tables and Views: Essential Tools for Serious Applications," www.codecraftersinc.com/pdf/DictionaryTables.pdf

DiIorio, Frank, "Dictionary Tables Reference Card" www.codecraftersinc.com/pdf/DictionaryTablesRefCard.pdf

DiIorio, Frank, “Rules for Tools - The SAS Utility Primer” www.codecraftersinc.com/pdf/UtilityPrimerWebFormat.pdf

Online Resources

listserv.uga.edu/cgi-bin/wa?S1=sas-l

SAS-L archives. Use search terms such as “macro” or “macro design”. Also search for postings by Ron Fehd and Ian Whitlock.

www.lexjansen.com

This site has almost 5,000 papers from the SUGI, NESUG, and PharmaSUG conferences, going back as far as 1996. Click on the conference of interest, then use the search tool.