Paper TU03

Building the Better Macro: Best Practices for the Design of Reliable, Effective Tools

Frank DiIorio, CodeCrafters, Inc., Chapel Hill NC

Abstract

The SAS® macro language has power and flexibility. When badly implemented, however, it demonstrates a chaos-inducing capacity unrivalled by other components of the SAS System. It can generate or supplement code for practically any type of SAS application, and is an essential part of the serious programmer's tool box. Collections of macro applications and utilities can prove invaluable to an organization wanting to routinize work flow and quickly react to new programming challenges.

But the language's flexibility is also one of its implementation hazards. The syntax, while sometimes rather baroque, is reasonably straightforward and imposes relatively few spacing, documentation, and similar requirements on the programmer. In the absence of many rules imposed by the language, the result is often awkward and ineffective coding. Some amount of self-imposed structure must be used during the program design process, particularly when writing systems of interconnected applications.

This paper presents a collection of macro design guidelines and coding best practices. It is written primarily for programmers who create systems of macro-based applications and utilities, but will also be useful to programmers just starting to become familiar with the language.

Introduction

Let's start with two possibly painfully familiar scenarios. In the first, we use a general-purpose macro to count observations in a dataset. Here is an excerpt:

```
%do i = 1 %to &nDatasets.;
    %let dset = %scan(&dsList., &i.);
    %dsetN(data=&dset., count=nobs)
    %if &nobs. > 0 %then %do;
        … ODS, PROC REPORT statements …
        %end;
%end;
```

The program runs for minutes instead of seconds, repeatedly printing the first dataset in `&DSLIST.` We eventually trace the reason to macro `%dsetN`, which set the value of variable `i` to 1. Thus, when control returned to the calling macro, `i` was always 1, rather than the expected progression of 1, 2, …, `&nDatasets`. We tried to do the right thing (use general-purpose macros to reduce code volume) but instead became another example of the adage "no good deed goes unpunished."

The second scenario has a better, but still imperfect, outcome. We use an application to create a SAS dataset, then use the output dataset in PROC REPORT.

```
%missVars(data=clin.ae, out=ae_miss, keep=usubjid)
proc print data=ae_miss;
     id usubjid;
     title "Variables in CLIN.AE that had all missing values";
run;
```

The macro ran successfully and created the dataset as described in the user documentation. What `%missVars` also did, however, was leave several datasets and global macro variables in the SAS session. These unwanted artifacts were not problematic in this use of the macro, but it is easy to see how unexpected datasets, option settings, and the like could have negative consequences.

This paper presents a collection of macro design guidelines that prevent these and similar scenarios. The list does not pretend to be exhaustive, but it *does* give the macro programmer a set of tools to work with. We will see that subservience of individual style to a rigid set of guidelines is not necessary, and that all that is required by the reader is an awareness of what the guidelines are attempting to accomplish (and prevent!)

Keys to Good Design

To understand why good design is important and how guidelines should be structured, think of a craftsman's work shop. It is a well-organized space, filled with specialized tools. Some of these are used every day. Others, while rarely used, are lifesavers when needed, filling a specialized requirement. They can make a nearly impossible task appear simple.

So it is with the programmer's virtual workshop. We need specialized tools so we can build general-purpose applications. While our choice of tools is not completely encompassed by the macro language, macros are a

significant asset in most SAS programming shops. Learning how to build them properly and how to use them effectively is an important step for any would-be master programmer.

The collection of guidelines presented below attempts to present generalities rather than specifics. The goal throughout is to guide the reader's thinking about how to write professional tools that reduce program volume, increase program reliability, and make life in the virtual workshop more enjoyable. The list is not exhaustive, and some of the points are arguable. Simply review them with an open mind and with an eye toward how they might complement and improve your programming.

1. Know When a Macro *Is* and Is *Not* Necessary

Macro programmers, especially those who have just learned the language, would do well to remember the adage, "to someone with a new hammer, everything looks like a nail." In some situations the macro language is the only possible clean solution. In other situations, the macro language is used unnecessarily, and therefore needlessly complicates a program. To identify these, it's worthwhile to identify the macro language's strengths:

- Repetitive tasks
- Conditional execution of program fragments
- Tasks that have varying execution paths and thus need to be parameterized

Why is it even necessary to discuss this? Because simple code is always easier to understand and maintain than complex code, and the macro language lends itself to complex (sometimes *overly* complex) coding. If you can create an effective solution without the macro language, do so. Also, recognize that there are situations where the macro language may, after some reflection, not be required. Some of these situations include:

- A simple, one-time program
- A straightforward program with a small number of parameters (macro variables) where no conditional execution is needed
- A DATA step using hash object or arrays
- DATA steps using CALL EXECUTE rather than macro callls
- Use of SQL or other programmatically generated statements
- Macro variables and functions used outside a macro (`%sysfunc`, `&sysDate`, *et al.*)

The list could go on and on. The important point here is that the macro language should not be the first and only tool you pick up as you enter the programming workshop. Being aware of SQL, DATA step, ODS, and other components of the language may result in a non-macro solution, and that's fine. All you do have to be is aware and informed. If, after evaluating the programming task at hand you decide that a macro-based solution is appropriate, that's fine too. (And it makes the rest of this paper worth reading!)

2. Conceptually Separate Utilities and Applications

Even though the overall structure of macro utilities and applications is identical, it is helpful to conceptually separate them. Granularity of the programs' scope of work is the key difference. Some programs are narrowly focused and perform work that is independent of data and projects. The scope of other programs is broader, and possibly unique to a project or sub-system. The former are utilities, the latter are applications.

Examples of macro utilities:

- Count the number of observations in a data set
- Convert one or more macro variables to upper case
- Create a count and list of distinct values of a data set's variable
- Quote each token in a macro variable
- Check for the presence of one or more variables in a data set

Examples of macro applications:

- Print the first "n" observations for every data set in a library
- Create a set of HTML files that display attributes for data sets and variables in a library
- Check the contents of one or more data sets for compliance with internal (corporate) or external (legal, industry) standards

The distinction is clear. Utilities perform generalized, tightly defined activies, while applications have a much broader range of activity. As a rule of thumb, the name of the utility macro should be a near-complete description of what it does. If `%quoteList` has code for both counting the number of items in a list and quoting the items, it may be better to isolate the item counting in a separate macro, leaving the bulk of the `%quoteList` program to focus on quoting. Since the nature of utilities is their potential for use by many applications, it is *imperative* that they "play well with others," doing what was expected by the calling program, but no more. This point is developed at length later in this paper.

This is not an artificial distinction. The development mindset is different for each, and they may have different validation and documentation requirements. Other features that vary between utilities and applications are the amount of error checking, and optimization of execution speed and other resources.

It is worth briefly noting that the size of a macro, particularly utility macros, is irrelevant. Rules of thumb that say a macro should be "x" (60, 100, other) statements ignore a reality of macro construction: macros should liberally use other macros to get their work done. Placing a limit on size is arbitrary, counterproductive, and artificial. Most importantly, limits ignore the real-world reality that some things simply need a lot of code to get done correctly.

3. Clearly Document the Macro

Documentation is written for two audiences: users of the macro and programmers who perform macro maintentance. Even the barest-boned macro must contain this essential content.

3.1 User Documentation. This is usually placed in a comment at the beginning of the macro. Note that this is the minimal documentation set. External documentation – web pages, sample libraries, and the like – frequently complement the internal comments. The amount and content of the header documentation vary, of course, by the complexity of the macro. This list identifies some common features:

- **Macro Name.** File name and/or complete, relative path.
- **Description.** Brief summary of the macro's functionality.
- **Input.** Datasets, with access restriction if appropriate; parameters, clearly identified with their status as optional or required or acceptable values, default values.
- **Outputs.** Complete description of output files, data sets, macro variables and other artifacts that are produced by the macro, distinguishing, if appropriate, between successful and unsuccessful or incomplete execution. This list, *and nothing else*, is how the user should expect the SAS session to change once execution is complete. Validation of the macro should confirm that all expected and no unexpected output is produced.
- **Processing.** Identify the names of any intermediate SAS data sets, macro variables, and other artifacts that are produced by the macro.
- **Execution.** Requirements such as running in open code, using a particular version of SAS, etc. The first executable statements in the macro should be those that test for these conditions. The macro should issue error messages and terminate if any of the requirements are not satisfied.
- **Examples of Use.** Start with simple calls to the macro, taking as many default values as possible, followed by as many other, more complex examples as necessary.
- **Error Conditions.** Items that would cause the macro to fail during:
  - Initialization: parameter values or inconsistencies; required resources that were not found
  - Execution: insufficient number of observations in a dataset; a variable that has no non-missing values
- **History.** Identify the programmer name, the date of the change, a revision code used throughout the program so the "touches" can be easily identified, and a short description of the change. The history comments should also identify changes associated with versions and maintenance releases.

The exact contents and format of the header may be prescribed by corporate or other standards. The contents will evolve over time, not just due to enhancements and fixes, but also due to usage. As end-users and programmers use the macro, it is likely that gaps, unclear wording, and the like will surface. As they emerge, they should be remedied.

Creating the header comment can be at least somewhat automated. The UltraEdit text editor, for example, allows definition of templates. They may contain boilerplate text as well as special text sequences for insertion of the file name, extension, and date. This is shown below. Other text editors have similar capabilities. They should be utilized whenever possible.

Template (prefilled items in bold)	Inserted text (resolved text in bold)
*	*
Name: **[FILE_NAME]**	Name: **quoteList.sas**
Description:	Description:
Input:	Input:
Output:	Output:
Usage Notes:	Usage Notes:
References:	References:
History: Date Init Comments	History: Date Init Comments
[DATE_USER]yyyy'-'MM'-'dd[DATE_USER_END] FCD Initial release	**2007-10-22** FCD Initial release
*/	*/

3.2 Programmer Documentation. The other audience, the macro programmer, is interested in aspects of the program not covered by the user-oriented instructions found in the header comment. In addition to being concerned with the inputs and outputs described in the header comment, the programmer also needs to know how processing

takes place. This need is at least partially satisfied by inserting comments throughout the program. Ideally, a programmer who has never seen the macro could review it and see the following at a glance:

- The start of major processing blocks
- Discussions of sections that were problematic to write or required non-standard or non-obvious algorithms
- Revision codes (these are discussed at length in Guideline 8, below)

These uses of comments are demonstrated below:

```
/***** Verify parameters *****/
%if condition %then %do;
    %let OKflag = f; %put error message ;
    %end;

/***** Per-dataset loop *****/
%do i = 1 %to &dsetN.;
    %let dataset = %scan(&datasetList., &i.);

    /* Use ANYMISS to identify obs with all missing values */
    %anyMiss(data=&dataset., missing=allMissing)

    %if &allMissing. ^= %then %do;  /* [U03] Add test for ALLMISSING */
        … write to PDF …   /* Important! PDF hyperlink workaround per
                              track # 5607646 (2007/11/10) */
        %end;
%end;

/***** Clean up *****/
proc delete data=datasetList;
run;
```

Just as there is no "right" or "wrong" header comment, so too is there no single "correct" form of the programmer's documentation. The objective is simply to make the macro as readable as possible. Judicious use of comments, together with external documentation such as flow charts and other data diagramming techniques, should give the programmer a clear idea of the code's structure and functionality.

4. Use Keyword Parameters Consistently

In macro design, as anywhere else in programming, designing for usability benefits anyone touching the program: users, developers, validation programmers, maintenance programmers, and so on. One of the simplest ways to make the macro user-friendly is using keyword parameters.

4.1 Parameters versus Keywords. Consider this macro call:

```
%xpt(ae cm dm, 100, , delete)
```

Somehow the user knows that `%xpt` should process data sets `AE`, `CM`, and `DM`, that the resulting files should be broken into 100MB pieces, and that preexisting versions of the files should be deleted at the start of processing. The user also knew that the deletion option is the fourth parameter, necessitating the use of a placeholder comma. It is a lot to ask of the user. No matter how functional and helpful it may be, the macro's invocation is decidedly "user-unfriendly," since no one wants to remember the order or meaning of parameters. Its design imposes requirements that encourage incorrect usage. Consider this alternative for the keyword parameter:

```
%macro xpt(data=, split=50, report=no, delete=no)
```

These calls to the macro are equivalent to the earlier, positional parameter version. All have the advantage of being more readable:

```
%xpt(data=ae cm dm, delete=yes, split=100)
%xpt(delete=yes, data=ae cm dm, split=100)
```

The user doesn't have to worry about parameter order, and the program becomes somewhat self-documenting by having parameters and their values present in the call to `%xpt`.

4.2 Naming and Values. Just as the presence of keyword parameters makes the macro more usable, so does consistency of how they are named and given values within a system of macros. Parameters that identify similar actions or content should be named consistently. Parameter values should also be coded consistently. Consider these syntactically correct macro calls:

```
%createSpec(datasets=ae cm, output=aecmspec.pdf, sortBy=pos, msg=yes)
%xpt(data=ae cm, msg=t, rpt=aeCMxpt)
```

In only two macros we have managed to lay waste to a good portion of usability. We specify a list of data sets with the `DATASETS` parameter in `%createSpecs`, but use `DATA` in `%XPT`. We specify an output file by `OUTPUT` in

`%createSpec,` but `RPT` in `%xpt`. Notice, too, that we need to specify the full file name in `%createSpec` but not in `%xpt,` where, presumably, the macro adds `.PDF` to the RPT parameter value. Finally, we muddy the waters by having a variety of ways to say "yes": in `%createSpec, MSG=yes,` and in `%xpt, MSG=t`.

Adopting a single set of parameter names and values across the system of macros improves usability. With some standards in place, some of which may require rewriting part of the `%createSpec` macro, we change the calls to:

```
%createSpec(data=ae cm, output=aecmspec, sortBy=pos, msg=yes)
%xpt(data=ae cm, msg=yes, rpt=aeCMxpt)
```

5. Use Consistent Program Structure

Just as we saw benefits from consistency of keyword parameter usage and naming, we also see benefits from using similar structure for all macros in a library. Once the structure is specified and its implementation becomes standard practice, the programmers who maintain and enhance them can more easily locate relevant portions of the program. A broad outline of sections and their functionality follows. Bear in mind that the structure may be overkill for some purposes and absolutely essential for others. The broader the scope of the macro, the more likely the benefit of following this structure.

In general terms, a macro should consist of four sections:

- **Header Documentation** (as described in the "Documentation" section, above)
- **Initialization.** Performs housekeeping activities such as: verifying correct execution environment; checking parameters and resources; capturing values of options that will be reset; defining local and global macro variables; and converting paramters to upper case. If all parameter and other tests pass, execution continues. Otherwise, it branches to the termination section.
  A useful error-checking technique is to set a macro variable flag (e.g., `%let bad = t;`) and print an error message. Once all checking is complete, print a final message and branch to the termination section if the flag was turned on. This allows display of all problematic conditions at once, and is demonstrated in the example in the last section.
- **Core processing.** This is the "meat" in the "sandwich" of program structure. Once the Initialization section is complete, the program says, in effect, "if I'm still executing it must mean that parameter values were valid and I can do the work that was described in the program header." Here, and throughout the macro, processing consists of original, macro-specific coding as well as calls to utility macros. If a condition arises during this processing that prevents continued execution, control branches to the Termination section, described next.
- **Termination.** Regardless of success or failure, almost every invocation of the macro should end in a termination section. Anthropomorphizing again, this is where the macro says "I'm done, so now let me check to make sure I'm not leaving anything behind that wasn't what I promised in the header documentation." This checklist includes, but is not limited to: deleting temporary macro variables, files, and data sets; resetting options back to their original values; and writing messages to the SAS Log describing output items' names and values.

The last section is vital. Without it, the macro has the potential to do its required tasks (create data sets, write a report, quote tokens in a list, etc.) *and* leave behind temporary data sets, global macro variables, system options and the like that were different than they were before the macro executed. Here, as in the practice of medicine, the principle is "first, do no harm." Altering the macro user's environment is not only harmful, it leaves the impression that the macro coding is sloppy and unprofessional. Cleanup is key.

6. Emphasize User, Program Communication

It is appealing to think of a macro as a nicely behaved black box that receives input, processes according to the documentation, and creates the expected output. This usually turns out to be the macro's minimal set of actions. Effective long-term use of both utility and application-class macros requires messaging in different forms. Let's look at the two principal recipients of macro messaging, the user and other programs.

6.1 Communication with the user. A well-designed and well-written macro should provide the option to the user of being able to view messages. These include the macro executing at one or more check points, displaying parameters it received, displaying data set counts, macro variable values and other information useful to the user as well as the macro developer, in case debugging is required. Messaging can be toggled by a macro parameter and handled throughout the program using a coding technique shown below:

```
%macro test(msg=yes, other parameters);
%local prt;
%if %upcase(&msg.) = NO %then %let prt = *;
%&prt.put test-> Dataset contains &dsnCount. observations;
data _null_;
     set rpt;
     &prt.put dsn= vname= status=;
run;
```

This method is both simple and powerful: if parameter MSG resolves to YES, macro variable PRT is null. If it is NO, PRT becomes *. References to PRT throughout the program result in it becoming a comment statement or an executable PUT or %PUT statement. This enables us to turn the messaging on or off with a macro parameter, rather than manually changing all the affected statements.

Some messages will be unconditional, and not subject to control by MSG-like parameters. Warning and error messages should always be displayed. Less essential, but still helpful, are messages about new functionality. These can be controlled via an automatic macro variable, as shown below:

```
%if %sysfunc(juldate("&sysdate9."d)) < %sysfunc(juldate("01apr2008"d))
    %then %do;
    %put;
    %put Version 4.00 effective March 1, 2008:
    %put;
    %put Changes from Version 3.x:;
    %put list of changes goes here
    %put See documentation location for details;
    %put;
    %end;
```

Say we moved the macro into a general-use / production area on March 1, 2008. This code fragment will automatically display messages about new functionality for a month.

A final note about messages to the user. When programming them, take the time to make them informative. A series of well-crafted messages from the macro should tell the story of its execution, with text accompanying values. Which would you rather see in a SAS Log?

```
2 dsns
AE
DM
```

or

```
After filtering, process 2 eligible data sets:
1 of 2: AE
2 of 2: DM
```

Here, as with other practices described above, spending a little extra time makes the macro output appear more professional, and raises the user's comfort level.

6.2 Communication with other programs. A less visible but equally important form of communication takes place between programs in the form of return codes. These are entities, usually macro variables, that tell the calling program how the macro completed. By convention, success could be 0, incomplete execution -1, and so on. The important point here is that the calling program needs to know the range and meaning of the return codes. This, of course, could be documented in the macro's header comments. Suppose macro %obsCount had these lines in its documentation header:

```
Return Codes: 0 = Data set located, observations counted successfully
              1 = Data set located, but could not count observations
              2 = Data set could not be located
              3 = Parameter errors or incorrect calling environment
```

Programs using %obsCount could use its return codes as shown below:

```
%obsCount(data=mast.rand, rc=randN)
%if &randN. ^= 0 %then %do;
    %put Could not determine obs count in mast.rand.  Execution terminating;
    %goto term;    /* Jump to program termination section */
    %end;
```

This and other techniques already shown require more programming by the macro programmer. Here, too, we have a better program: if something was amiss with dataset mast.rand, the calling program can fail gracefully rather than littering the Log with SAS-generated error messages. Handling return codes requires more programming, but results in a cleaner, more informative, and more professional-appearing Log.

7. Control Macro Variable Scope

Recall the first scenario at the beginning of this paper. A utility used by a macro set variable I to 1. This disrupted execution of the calling program, which also had a macro variable named I. Besides demonstrating a breathtaking lack of variable-naming imagination, the example also highlights the need for awareness of macro variable scope.

This is an important topic, worthy of attention in an entire paper. Here, however, we'll focus on a few Best Practices surrounding scope. First, be aware of global and local scopes: a macro variable defined in open code or explicitly, via a %global statement, has global scope; it is available to be read or written anywhere in the program. A locally

scoped macro variable is defined in a macro and is available to all macros invoked by the macro. Let's look at an example of the dangers of uncontrolled scoping:

```
%macro outer(list=);
       %let upper = %sysfunc(count(&list., %str( ));
       %do i = 1 %to &upper.;
           %let token = %scan(&list., &i.);
           %print(data=&token.)
       %end;
%mend;

%macro print(data=);
       %let i = %index(&data., .);
       %if &i. = 0 %then %let data = work.&data..;
       … other statements not shown …
       proc print data=&data.;
            title "&data.";
       run;
%mend;
```

Variable `I` was defined in `OUTER`'s symbol table and so was in `%print` and other subordinate macros' symbol tables. The change to `I` in `%print` meant that `I`'s value was altered once execution returned to `OUTER`, making its execution unreliable at best. There are several practices that help avoid this variable collision.

7.1 Explicitly Identify Local Variables. Use `%local` statements to create copies of `I` in each macro:

```
%macro outer(list=);
       %local i;
       %do i = 1 %to &upper.;
           %let token = %scan(&list., &i.);
           %print(data=&token.)
       %end;
%mend;

%macro print(data=);
       %local i;
       %let i = %index(&data., .);
       %if &i. = 0 %then %let data = work.&data..;
       … other statements not shown …
%mend;
```

7.2 Chose Unique Variable Names. Select a consistent prefix for each macro, using a value that will not be used by other macros in the system. Identify the prefix in the program's header comment.

```
%macro outer(list=);
       %local OUTi;
       %do OUTi = 1 %to &upper.;
           %let token = %scan(&list., &OUTi.);
           %print(data=&token.)
       %end;
%mend;

%macro print(data=);
       %local PRi;
       %let PRi = %index(&data., .);
       %if &PRi. = 0 %then %let data = work.&data..;
       … other statements not shown …
%mend;
```

Using `%local` statements and/or naming conventions ensures a variable's scope is limited to that macro. If a macro variable needs to be shared between macros, it should explicitly be declared as global. Ideally, *all* variables used in a macro will be identified in `%global` or `%local` statements, even if your knowledge of the macro symbol table hierarchy makes you feel this is unnecessary.

8. Implement Diagnostic and Debugging Code

No matter how good you, the programmer, feel about yourself and your abilities, chances are your macro will break at some point. Stated more gently, chances are your macro will need attention to deal with unanticipated situations. Diagnostics and coding techniques associated with debugging are different than the communication issues dealt with in the earlier sections. We will discuss two of them here: debug parameters and revision codes.

8.1 Debug Parameters. Conceptually, this is not very different than the impact of MSG=YES discussed earlier. In this context, however, the only consumer of the information is the macro developer. A macro parameter can be used to display the amount of %PUT, PUT, and PROC statements that are executed. This is output over and above that produced by a "normal" execution of the macro, and provides information that the programmer may find useful when ferreting out anamalous (or nonexistent) output. The parameter can take on different values so that the larger the number, the greater the amount of output. This is shown in the following example (only debugging-related code is shown):

```
%macro rpt(debug=0, other parameters);
%if &debug. > 0 %then %do;
    %put Global macro variables at start of execution:;
    %put _global_;
    %end;
%if &debug. >= 1 %then %do;
    proc freq data=_TMP_2;
         tables grp1-grp&n.;
         title "Grouping variables from transposed master data";
    run;
    %end;
```

The result is as effective as it is simple. DEBUG defaults to 0, which produces no extra output. If its value is greater than 0, we display all global macro variables. If it is 1 or greater, we also get a frequency distribution of variables that we know will be helpful diagnosing the problem. Just how much output is produced by which levels of DEBUG is a matter of experience, and this experience is usually gained during the initial development of the macro. As time goes on and the macro's features are enhanced, the amount and control of the diagnostic output can be easily altered.

8.2 Revision Codes. A useful utility or application will, over the course of its life, need maintenance and encourage enhancement. This, in turn, means that the original program will be altered. The usual way of doing this is a line in the header comment:

```
History:  2007/11/23  FCD  Initial version in autocall library
          2007/12/22  FCD  Add handling for empty datasets
```

Presumably the initial version of the macro assumed the input data would always have a non-0 number of observations to process. In reality, there were situations where this was not the case. In the above example, rather than have the macro "die noisily," programmer FCD made the necessary changes just before leaving for a well-earned Christmas vacation.

The question then becomes what, exactly, was changed in the program? If the modification was not sufficient or if similar changes need to be made in other macros, it would be nice to quickly locate the affected statement(s). Revision codes solves this problem. First, let's revisit the header comment:

```
History:  2007/11/23  FCD  Initial version in autocall library
          2007/12/22  FCD  [U01] Add handling for empty datasets
```

We use the convention [Unn] ("update number nn") to identify the change. Then throughout the program we reference it:

```
%dsetCount(data=pass1, count=np1)
%if &np1. < 1 %then %do;  /* [U01] */
    %put First filtered pass resulted in an empty data set;
    %put Execution terminating.;
    %goto term;
    %end;   /* [U01] end */
```

We mark off the portion of the program affected by the change with comments, adding an additional note that the %end statement was the last location where the change took place. This way, the programmer can easily locate all related changes simply by searching for the text string U01. It's not unlike Hansel and Gretel leaving a trail of bread crumbs in their walk through the woods so they can find their way home (perhaps not the best analogy, since birds ate the crumbs and the children became lost; however, the general point is sound).

As with messaging and diagnostics, it requires additional programming. Indeed, in this case the extra effort does not even add any executable statements to the program. The improvement in maintainability is marked. All that is required is a standard to adhere to and a little discipline when making changes.

9. Use Built-In Macro Tools

Rather than reinvent the wheel and possibly end up with a less-than-round result, you should take advantage of the macro tools that come with SAS. The brief listing that follows identifies language features that aid program development, debugging, and distribution:

9.1 Automatic Macro Variables. SAS automatically creates and maintains a host of automatic macro variables. Some of their functionality can be duplicated by DATA step or other coding. Others are unique to the macro language. In either case, it pays to know what is available.

In the following example, a "macro unaware" solution needlessly uses a DATA step to create a macro variable, while the "macro aware" code simply accesses the SYSDATE9 variable. The amount of code is reduced and makes the program more readable.

Unaware of Automatic Macro Variables	Utilizes Macro Variables

`data _null_;`
`  x = today();`
`  call symput('date', put(x, date9.));`
`run;`
`footnote "Run date: &date.";`
	`footnote "Run date: &sysdate9.";`

The previous example showed how automatic variables can replace cumbersome coding. Some variables make an otherwise impossible or incredibly convoluted task straightforward. The variable SYSINDEX holds the number of macro executions that have been made so far during a SAS session. Normally, this is a "so what" item. Recall, however, the programmer workshop metaphor we used at the start of the paper. SYSINDEX is a prime example of the obscure, seldom-used tool that becomes a lifesaver. Consider macro %ISO, which requires an array of date and time variables to assemble an ISO 8601-compliant date-time variable. Since different date components might be used in several calls to %ISO in the same DATA step, we cannot use the same array name. Adding the SYSINDEX value to the array ensures that multiple %ISO calls will generate multiple, and unique, array names.

```
%macro iso(date=, out=);
       … statements not shown …
       array dtParts&sysindex.(6);
       … statements not shown …
%mend
data ae:
     set clin.ae;
     %iso(date=onset)
     %iso(date=term)
run;
```

The first %ISO call might generate array DTPARTS45, the second DTPARTS46. The automatic macro variable guarantees that the array names will be unique, and is a good example of the payoff from browsing the help files even if you think you've "seen everything."

9.2 Autocall Libraries. Recall the design goals described earlier in this paper: we document the macro, provide clear messages to the user, and take other steps to make it as user-friendly as possible. The macro itself must be easily made accessible to the user. This is handled by the SAS autocall facility, as shown below:

```
options sasautos=('path1', 'path2', sasautos) mautosource mrecall;
```

The effect is similar to paths for formats and ODS item stores: a library of tools is made available to the user with a single option statement. Programs using these options do not have to use %include or other cumbersome constructs. A cautionary aspect of autocall usage is knowing where a macro is coming from; SAS will use the first instance of a macro as it searches the autocall path. Using the above example, if %FREQS is defined in both path1 and path2, SAS will use the copy found first, in path1. The MAUTOLOCDISPLAY system option displays the source of macros as they are used. Like MPRINT and various other options, it can be toggled on and off via debugging parameters (see "Options," below).

9.3 Functions. Just as you should use automatic variables to reduce code volume and make programs more readable, so should you use macro functions. Functionality that is not addressed by native, autocalled macro functions is usually found in other Base SAS functions. These are made accessible via the %sysfunc and %qSysfunc functions. Given the relatively few limits on these functions' use within a macro, there should be relatively few instances of a DATA step being used solely to produce a macro variable. This is shown below; the DATA step, while clever, is unnecessary. The %sysfunc-aware code is more succinct, executes faster, and is easier to maintain.

Unaware of %sysfunc	Uses %sysfunc

`data _null_;`
`  call symput('found', 'n');`
`  set master.clin;`
`  call symput('found', 'y');`
`  stop;`
`run;`
`%if &found. = y %then %do;`
`  … statements not shown …`
`  %end;`
	`%if %sysfunc(exist(master.clin)) %then %do;`
	`  … statements not shown …`
	`  %end;`

9.4 Options. As mentioned in the abstract, the macro language has tremendous flexibility as well as the potential to produce chaos if programmed incorrectly. Fortunately, the variety and number of macro-related system options increases with each release of the SAS System. You do not *have* to use `MPRINT`, `MAUTOLOCDISPLAY`, `MLOGIC`, and `MPRINTNEST` in every program, but you should at least be aware of their existence and purpose. Be aware that some of these options, while providing essential information for debugging, can clutter the SAS Log to the point of being unreadable. One way to control the volume of output is an extension of the debugging parameter technique shown in Section 8, above. Notice that we follow the standard good practice of preserving option values, setting them to meet the desired debugging level output, and then reverting them to their original values in the termination section.

```
%macro rpt(debug=0, other parameters);
%local opts;
%if &debug. > 0 %then %do;
    %let opts = options %sysfunc(getoption(mprint))
                %sysfunc(getoption(mautolocdisplay))
                %str(;) ;
    options mprint mautolocdisplay;
    … other DEBUG > 0 actions …
    %end;
/* termination section */
&opts.
```

10. Build the Other Tools You Need

There is, sadly, a dearth of macro tools built into the SAS System. Macro-related options abound, and many are helpful, but they can be verbose, cluttering the SAS Log to the point of becoming unreadable. Sometimes there are diagnostic tools that you would like but that simply do not exist. One of the hidden costs of having a robust library of macro utilities and applications surfaces when we look at home-grown tools that need to be developed. This section describes two such tools. One is presented in full, while the second is described only at a high, non-coding level. Both should give a feel for the kind of supplemental tools that are helpful and the programming effort involved.

Tool 1: Variable Display. During macro development and debugging it is often helpful to display a list of all global macro variables and their values. Any one who has used `%put _global_;` for this purpose is aware that it is both alluring in its simplicity and disappointing in its output. The values come out in an order that is, to put it kindly, not discernable to the naked eye. A simple macro, shown here without comments for the sake of space, follows:

```
%macro printMacvars;
%local _opts;
%let _opts = %sysfunc(getoption(mprint)) %sysfunc(getoption(notes));
options nomprint nonotes;
proc sql noprint;
     create table _macvars_ as
     select *
     from dictionary.macros
     where offset=0 and scope='GLOBAL'
     order by name
     ;
quit;
%if &SQLobs. = 0 %then %do;
    %put AllMacVars-> No global macro variables matched search criteria;
    %goto bottom;
    %end;
data _null_;
     set _macvars_ end=eof;
     file log notitles;
     if _n_ = 1 then
        put / 'Macro Variable' @34 'First 50 Characters' /
              32*'=' +1 50*'=' ;
     put name $33. value $char50.;
     if eof then put   32*'=' +1 50*'='
                     / "# of variables = " _n_
                     / 83*'='
                     ;
run;
%bottom: proc delete data=_macvars_;
         run;
```

```
          options &_opts.;
  %mend;
```

The utility is simple and powerful, and introduces the use of SAS metadata (dictionary tables) as an adjunct to tool development.

Let's look at the macro in action. If we defined two global macro variables, `global1` and `testmacvar`, a call to `%printMacvars` would produce the following output:

```
  Macro Variable                 First 50 Characters
  ============================== ==================================================
  GLOBAL1                        G1
  TESTMACVAR                     tmv
  ============================== ==================================================
  # of variables = 2
  ================================================================================
```

The display is easy to read and clearly labeled. The alphabetical ordering of the variables doesn't suggest a huge improvement over `%put _global_` in our simple test case, but it doesn't take much imagination to see how this presentation would be helpful when dozens of variables are involved.

Tool 2: Header Web Page. Section 3 discussed the merits and importance of the header comment at some length. What was not addressed was the difficulty of locating the macros and having a quick way to read the headers. When all macros are in a single directory, and when there are relatively few of them, printing them or opening them (as read only!) in an editor is not too tedious. Consider, however, that a mature system of macros which includes utilities and a reasonable range of applications can easily approach 100 individual files. In this scenario, printouts and text editors are not a match for the complexity of the system.

One approach is to exploit the similarity of the macros' structures. Suppose each macro header is a single comment, such as:

```
  /*  xxx.sas
     … header comments …
  */
```

We can exploit this consistency by writing a program that does the following:

- Identifies all directories in the autocall library path
- Reads each *.sas file in these directories. For each file, store the complete path name and write an HTML file with the program name. The HTML contains the first line in the source file up to and including the first line containing a */ (the end of the header comment). The HTML also contains a hyperlink to the source file.
- Writes other HTML files so that the final product is a frame set with a navigation pane.

The output from this tool is shown on page 16, below.

11. Adopt the Software Development Mindset

This last set of guidelines develops a best-case scenario. You have constructed a library of utility macros that are liberally used by macro-based applications. These applications are well-received by end users, creating demand for more options and, eventually, completely new applications. As a developer, you are also an end user of your utility macros. As such, you see the occasional need to add parameters to the macros and to build new utilities to meet the expanding demands of applications.

Consciously or not, you have become a software developer.

If your programs were entirely self-contained (no `%include` statements or macro references) and if you were the program's only user, then change is not an issue. You might make a backup copy of the program, then alter the program and use it. The change in functionality or method of invocation is irrelevant because it is immediately known to the program's entire user community (i.e., you). If a change did not work as planned you can simply tweak the program and rerun it until it meets your requirements.

However, once programs are dependent on utilities and other users have developed expectations about an application's input and output, you have moved into an entirely different realm of programming. Some of the key coins of this realm are: separation of development and production areas, and validation. Each of these is a large topic. We present only a cursory look at them here, limiting ourselves to the macro development context, and are fully aware that these topics are controlled to varying degrees by standard operating procedures in most organizations.

11.1 Separation of Development and Production. Source code is typically stored in a directory structure that reflects the life cycle of the macro. A common organization is to have development, testing, and production directories. Program setup files or macros can take advantage of this structure by prepending development directories to the autocall path. In the following example, `testDir1` and `testDir2` contain macros that are

undergoing development or revision. Invoking %progSetup with DEV=YES inserts these locations in front of the default (production) directories, and ensures us that progam x.sas in the test / prepended directories will be used instead of x.sas found in any of the production directories.

```
%macro progSetup(dev=no);
       %local autoPath;
       %let autoPath = 'prodDir1' 'prodDir2' 'prodDir3';
       %if %upcase(&dev.) = YES %then
           %let autoPath = 'testDir1' 'testDir2' &autoPath.;
       options sasautos=(&autoPath. sasautos) other autocall options;
```

There can, of course, be more elaborate directory structures. The underlying logic for the structure is the same, however: separate the production areas seen by the end users from the testing and development areas, and promote new or upgraded programs to production only after they have been thoroughly reviewed.

11.2 Validation. Any piece of code, macro or otherwise, must be validated before it can be moved to a production area. The resources devoted to the validation effort vary. Typically, the more complex or mission-critical the program, the more elaborate the validation process. When validating *any* macro, the validation checklist should do the following:

- confirm there is adequate header documentation
- verify macro output is limited to what was described in the header comment
- examine naming and default values of keyword parameters, ensuring they match what is described in header comment
- thoroughly test parameter error-checking
- examine the program for appropriate exception / error handling
- confirm that the termination section removes temporary variables, datasets, etc. and sets any re-set options to their original state.

Implementation: An Extended Example [11]

Let's close with a example of a small, complete application that demonstrates many of the guidelines presented in this paper. %attrDiff was born out of need – datasets in a library had to have identical attributes for like-named variables. Patient identifier USUBJID, for example, should always be character, length 15, and have the label "Subject Identifier". The application reads the SAS dictionary table COLUMNS and compares attributes (type, length, label) of all variables in a library. Application output is a dataset containing the dataset name, variable name, and attributes of variables that are mismatched.

Before looking at specifics, it is worthwhile to note that a good portion of the program is taken up by comments and blank space. There are about 150 lines, and only about half are program statements. The rest of the space is used by header comments and a sprinkling of comments intended to help any one trying to read the program. This narrative and blank space (blank line separators, indentation, alignment, etc.) is typical of a well-documented program, and makes it easy to understand and read.

```
/*  attrDiff

    Function:  Identify conflicting attributes of like-named variables
               in a library

               For each parameter, describe content, permissible values, whether required
               or optional, and default value.
    Input:     Parameters (not case-sensitive)
               LIB     LIBNAME of library to examine.
                       REQUIRED.  No default.
               COMPARE Attributes to compare. Order does not matter.
                       Specify any or all of these variable attributes:
                       T - type
                       S - length
                       L - label
                       OPTIONAL.  Default is tsl
               OUT     One or two-level name of output dataset.
                       Cannot be in same library as specified
                       by LIB parameter.
                       REQUIRED.  No default.
               MSG     Write messages to Log?  YES or NO
                       OPTIONAL.  Default=YES
```

[1] The source for %attrDiff source as well as programs demonstrating its use are available at the author's web site: www.CodeCraftersInc.com

```
                Describe output dataset observation content, sort order, and conditions
                that prevent it from being created.
    Output:     Dataset specified by OUT parameter. Sort order is
                name, dataSet.  Variables:
                dataSet      $ 32  Dataset name
                name         $ 32  Variable name (upper-cased)
                type         $  4  Type (if COMP contained t)
                typeFlag        8  Differing TYPE (0 or 1)
                length          8  Length (if COMP contained s)
                lengthFlag      8  Differing LENGTH (0 or 1)
                label        $255  Label (if COMP contained l)
                lablelFlag      8  Differing LABEL (0 or 1)
                The dataset is created with 0 observations if there
                are no attribute conflicts.
                The dataset is NOT created if there are parameter
                errors.
                OUT dataset cannot be located in the same library
                as LIB.

                If we say to run in open code, we should also test for it at the beginning of
                execution.
  Execution:    Run in open code

    Example:    %attrDiff(lib=clinical, compare=ts, out=probs)
                    Compare type and length for datasets in library
                    CLINICAL.  Write output to dataset WORK.PROBS

                Revision code notation is arbitrary.  Just be consistent in referring to it
                throughout the program.
    History:    2007-10-08  JHA  Initial program
                2007-11-15  JHA  [U01] Add OUT parameter
*/
       Use keyword parameters. Ensure default values match what was described in the
       header comment.
%macro attrDiff(lib=, compare=tsl, msg=yes, out=);    /* [U01] */

       Since we said run in open code in the header comment, we need to test it here.
       Branch to the last statement rather than the termination section.
       /* ---------- Be sure we're running in open code ---------- */
       Knowledge of automatic macro variables makes open code test straightforward.
       %if &sysprocname. NE %then %do;
           %put attrDiff-> Must run in open code.  Execution terminating.;
           %goto lastStmt;   /* <<<< <<< << < <<<< <<< << < <<<< <<< << < */
           %end;

       Begin initialization section
       /* ---------- Housekeeping and initial messages ---------- */
       Take explicit control of variable scope.  You can place the %LOCAL statement near the
       statements creating the variables.
       %local opts star;
       Save initial option values before resetting.
       %let opts = %sysfunc(getoption(mprint)) %sysfunc(getoption(notes));
       options nomprint nonotes;

       %if &msg. = NO %then %let star = *;

       Begin writing messages to Log.
       %&star.put;
       %&star.put attrDiff-> Begin. Examine library [&lib.] compare [&compare.] create [&out.];

       Standardization of values makes evaluation easier later on (code is less cluttered due
       to lack of %upcase function references).
       /* ---------- Upper case some parameters ---------- */
                      Use built-in macro functions
       %let lib     = %upcase(&lib.);
       %let compare = %upcase(&compare.);
       %let msg     = %upcase(&msg.);

       Create error flag OK.  As we find problems, set OK to f and write a message.  This lets
       us accumulate errors and report more than one problem at a time.
```

```
/* ---------- Check for parameter errors ---------- */
```

Take explicit control of variable scope. You can place the %LOCAL statement near the statements creating the variables.

```
%local ok outLib;
%if &lib. = %then %do;
    %let ok = f;  %put attrDiff-> LIB cannot be null;
    %end;
```

Use %sysfunc as much as possible to reduce code volume.

```
    %else %if %sysfunc(libref(&lib.)) ^= 0 %then %do;
          %let ok = f;  %put attrDiff-> Input LIBNAME [&lib.] not found.;
          %end;
```

Reference to revision code [U01]

```
%if &out. = %then %do;     /* [U01] */
    %let ok = f;  %put attrDiff-> OUT cannot be null;
    %end;
    %else %do;
          %if %index(&out., .) %then %let outLIB = %upcase(%scan(&out., 1, .));
              %else                  %let outLIB = WORK;
          %if &outLIB. = &lib. %then %do;
              %let ok = f;
              %put attrDiff-> OUT and LIB libraries cannot be identical;
              %end;
              %else %if %sysfunc(libref(&outLIB.)) ^= 0 %then %do;
                    %let ok = f;
                    %put attrDiff-> Output LIBNAME [&outLIB.] not found.;
                    %end;
          %end;
%if &compare. = %then %do;
    %let ok = f;  %put attrDiff-> COMPARE cannot be null;
    %end;
    %else %if %sysfunc(verify(&compare., TSL)) > 0 %then %do;
          %let ok = f;  %put attrDiff-> COMPARE can only contain T, S, or L;
          %end;
```

This is a simple way to avoid bulky macro coding. The alternative would have been: %if &msg. ^= NO & &msg. ^= YES %then %do; The benefit of this technique grows as the number of comparisons increases.

```
%if %sysfunc(indexW(NO YES, &msg.)) = 0 %then %do;
    %let ok = f;  %put attrDiff-> MSG can only contain YES or NO.  Found [&msg.];
    %end;
```

Branch to termination and print message if we found any error conditions.

```
/* If anything was amiss, print a message and branch to bottom */
%if &ok. = f %then %do;
```

We keep the user informed about what's happening and why.

```
    %put attrDiff-> Execution terminating due to error(s) noted above;
    %put attrDiff-> Output dataset [&out.] will NOT be created;
```

Execution is forced to the termination section. This guarantees that any clean up that is required will, in fact, get done. We do *not* use %return or %abort!

```
    %goto bottom;   /* <<<< <<< << < <<<< <<< << < <<<< <<< << < <<<< <<< << < */
    %end;
```

Initialization section is complete. Begin core processing.

```
/* ---------- Create SQL statement fragments based on COMPARE value ---------- */
```

Take explicit control of variable scope. You can place the %LOCAL statement near the statements creating the variables.

```
%local sumOps tf sf lf;
%if %index(&compare., T) %then %do;
    %let tf     = type, count(distinct type)   > 1 as typeFlag, ;
    %let sumOps = , typeFlag;
    %end;
%if %index(&compare., S) %then %do;
    %let sf     = length, count(distinct length) > 1 as lengthFlag, ;
    %let sumOps = &sumOps., lengthFlag;
    %end;
%if %index(&compare., L) %then %do;
    %let lf     = label, (count(distinct label) > 1 |
                  (count(distinct label) = 1 & sum(missing(label) > 0)))
                  as labelFlag;
    %let sumOps = &sumOps., labelFlag;
    %end;

/* ---------- Build the dataset ---------- */
```

```
      If this were a longer, more complicated program, we might have a %put statement saying
      "Step 1: read COLUMNS table, collect variable attributes"
      Since the core processing is basically just a single step, this message is probably not
      necessary.
      proc sql noprint;
           Reference to revision code [U01]
           create table &out.  /* [U01] */ as
           select &tf. &sf. &lf., upcase(name) as name, memname as dataSet
           Knowledge of SAS metadata (dictionary tables) makes creation of &OUT possible in
           a single statement.
           from dictionary.columns
           where catt(libname, memType) = "&lib.DATA"
           group by name
           having sum(0 &sumOps.) > 0
           order by name, dataSet
           ;
           %&star.put attrDiff-> &SQLobs. variables with mismatches.;
      quit;

      Code processing section is complete.  Execution drops into termination section.
      %bottom: %&star.put attrDiff-> Done.;
               %&star.put;

               Only clean up required is setting some options to their original values.
               More complex macros might require deletion of temporary data sets, temporary
               global macro variables, etc.
               /* ---------- Revert to orginal MPRINT and NOTES values ---------- */
               options &opts.;

%lastStmt: %mend attrDiff;
```

Closing Comments

As stated in the introduction, this is a brief review of a large topic. You may have differing opinions about how to approach some of the items that were discussed. Indeed, you may have a list of items that you think should have been discussed but were omitted. For now, however, simply consider the reasons for why the items were included, and how systems of programs would benefit from their use.

And, of course, if you have questions or comments, contact the author:

Frank@CodeCraftersInc.com

References

There are *many* books, papers, and other resources that deal with the macro language. Here are some of the more useful web sites.

SAS Online Documentation — http://support.sas.com/onlinedoc/913/docMainpage.jsp

Find the macro documentation by following "Base SAS" → " SAS Macro Language: Reference"

Conference Proceedings Archives — http://www.lexjansen.com

This site has thousands of conference papers. Use search terms such as "macro language", "macro design".

SAS Support — http://support.sas.com

This site has many sample programs, links to publications, and the same "knowledge base" used by the SAS tech support staff

SAS Community — http://www.sascommunity.org/wiki/Main_Page

A sort of virtual users group, containing blogs, forums, downloads.

SAS-L list Server — http://groups.google.com/group/comp.soft-sys.sas/topics?hl=en

Questions, answers, and opinions dating back to the 1980's. A high-volume and high-quality group. http://listserv.uga.edu/archives/sas-l.html

CodeCrafters, Inc. — http://www.CodeCraftersInc.com

The author's web site, containing SAS-related papers and many other professional and personal links.

Fine Print

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. Other brand and product names are trademarks of their respective companies.

Figure 1: Programmatically-Generated Documentation