

# The Macro Debugging Primer

Frank DiIorio, CodeCrafters, Inc., Philadelphia PA

## Introduction

You know you're in for a rough ride when the company that wrote the software says in its documentation:

Because the macro facility is such a powerful tool, it is also complex, and debugging large macro applications can be extremely time-consuming and frustrating. (SAS 2003-1)

Here are some of the scenarios that might have motivated that mildly chilling sentence:

- The macro produced unexpected output, printing attributes for data sets that were not specified by the macro call. The SAS® Log contained no warnings or errors.
- The macro was run using a new project's root directory as a parameter value. The heretofore reliable macro fails to execute, and creates a host of near-indecipherable SAS error messages.
- The macro produced no errors, warnings, messages, or output.

What do you do? The seasoned programmer's first instinct is to turn on some of the diagnostic tools that come with SAS. Options such as MPRINT and SYMBOLGEN often do the trick, and make the cause of the unexpected behavior obvious.

More often, and certainly when dealing with complex macros, such out-of-the-box tools are *part of* but not the *complete* solution. What's needed is a collection of techniques and applications that supplement Base SAS. This paper addresses this need. Specifically, it:

- Briefly discusses the debugging process
- Outlines several macro design Best Practices in the belief that as the quality of the initial program improves, the need for downstream debugging decreases
- Describes general debugging strategies
- Presents some of the built-in macro-related debugging features available in Base SAS. For each, we present a description, syntax, and scenarios where the feature might be useful.
- Illustrates some simple debugging tools that can be easily created with a minimum of macro coding
- Describes more complex, stand-alone applications that significantly extend the programmer's ability to design and debug macros.

The paper's syntax-specific discussion is based on SAS Version 9.1.3 running in Windows XP. The general comments and programming strategies apply to all versions and operating systems.

## Just What Is Debugging, Anyway?

Although the answer may seem obvious, it is worthwhile taking a moment to consider what, exactly, is meant by debugging. The simple definition goes something like this: "Debugging is gathering information and writing code that fixes a problem in a program." Any programmer with even a modest amount of experience will tell you that if a debugging solution involves writing code such as

```
if usubjid = 'A123' & visitNum in (-1, 0, 3) then aesev = .;
```

it is not really a fix so much as a Band-Aid. Hard-codes such as this or similar, brute-force solutions are not only inelegant but are virtual guarantees that you'll be back, coding exceptions for USUBJID=A124 and who knows how many others.

This leads us to the more robust definition of debugging: "Gathering information and writing code that fixes the *root cause* of problems in a program." This requires a somewhat different mindset than the development programmer and goes beyond a simple "hurry up and fix it" mentality. Rather, it requires abandoning assumptions about how the program works, leaving open all possibilities for why execution failed. This objectivity is sometimes absent in the developer. Debugging also requires some different tools than might be used by a development or validation programmer.

It's important to identify the impact of when during the program's life cycle the error occurred. Any development methodology (SDLC, RAD, XP, *et al.*) will have phases where errors can be introduced. Those created during development are often more readily identified and easily fixed because they were incremental: the problem was introduced and resolved as a result of a specific change (add some conditional coding, use a new version of a format, etc.). Debugging a finished product is often more complex since the advantage of the numerous code-run-debug cycles is lost.

## Why the Emphasis on Macro Debugging?

The macro language has a number of features that make it both powerful and difficult to debug. To be able to unravel what happened in a macro-driven program, it is important to understand some of the ways it differs from the rest of the SAS language. Three of the most important are:

- **Timing.** At the risk of over-generalizing, execution of a macro-less program is reasonably straightforward: statements are executed in the order in which they are encountered in the program. A program employing macros, however, requires distinguishing compile-time and execution (or run-time) activities. Programmer recognition of the macro language's primary role as a text/program generator is vital. This and the next point are closely linked.
- **Program Structure.** Macro programming is characterized by what might be called non-linear structure and execution. Execution flow can be modified by logical conditions evaluated by, among others, %IF-%THEN statements. The structure of a macro-based program is also different than the traditional, macro-less program. Macros can, and should, refer to other macros to perform specialized tasks. This almost guarantees that at some point an error that is manifest on line X of program Y will have its source in line "far-from-X" in program Y or somewhere in program "not-Y." The lack of proximity in a program between error cause and manifestation presents special debugging challenges.
- **Scope.** Macro variables have a well-defined but sometimes overlooked set of rules regarding their scope, or the locations throughout the program in which they will be recognized. Macros that explicitly control scope will seldom exhibit the sometimes erratic behavior of those that defer to SAS's defaults. See Carpenter 2005-1 for a good discussion of this topic.

This brings us to one of the main reasons for this paper: macros are hard to debug, and since they are harder they are usually more interesting than a "pure," macro-less program. The pieces to the programming puzzle become numerous and nuanced, and thus require more tool-building and diagnostic creativity than other parts of the SAS System.

## Good Design Reduces the Need for Debugging

While macro design is not the stated focus of this paper, it cannot be denied that good design prevents errors and so warrants discussion here. Knowledge of the elements of a "good" macro is also useful during debugging, when code can be rewritten to be more robust and reliable.

The following items are taken from *Building the Better Macro* (Dilorio 2008).

- **Know when a macro is and is not necessary.** Code written as a macro can sometimes be replaced by CALL EXECUTE, open-code use of %SYSFUNC, SQL-generated macro variables, BY-group processing or similar, more transparent constructs.
- **Clearly document the macro.** Headers and other comments are the debugging programmer's key to understanding program logic. The debugging programmer can demonstrate an appreciation of programs being living documents by using revision codes. These are discussed in "Building on the SAS Toolset," below.
- **Use keyword parameters and values consistently.** Parameter naming and values should be consistent within a system of macros. Likewise, values should be standardized (e.g., upper-cased) early in the program to facilitate evaluation in %IF and similar statements.
- **Use consistent program structure.** Predictable program structure brings new (non-developer) programmers up to speed more quickly than idiosyncratic construction. This consistency is particularly important during debugging, when the speed of problem isolation is paramount.
- **Know when to redesign.** If a macro becomes extremely large, filled with complex coding constructs and numerous indirect variable references (the dread &&var, &&&var, etc.) then consider taking a different, possibly simpler approach.
- **Emphasize user, program communication.** Control of messages relevant for both the end user and debugging programmer is essential. This can take various forms: routine and error/warning text written to the SAS Log, status variables, error logs, and others.
- **Take control of macro variable scope.** Already noted above. Use of %LOCAL and %GLOBAL statements is simple, and removes any ambiguity about which values are known to which macros. This is especially critical when macros use specialized, single-purpose utilities.
- **Implement diagnostic and debugging code.** At any point in the life cycle, but especially during development, it is helpful to have a means to control the amount of debugging and diagnostic output produced by the macro.
- **Use built-in macro tools.** Be aware of what built-in tools SAS has to offer, and be prepared to ...
- **Build the other tools you need.** See the "Home-Grown Tools" section, below.

## Debugging Strategies and Considerations

The following general debugging strategies are taken from *The SAS Debugging Primer* (Dilorio 2004).

- **Debug incrementally.** In addition to the immediate, problem-resolution role of debugging, it should also be a learning experience. If you make three changes to a program and it runs correctly, you may have fixed it, but you *don't* know if all three changes were required. You learn best when you make a minimal set of changes, run the program and evaluate results.
- **Debug from top to bottom.** One small syntax or logic error at the top of a program can spawn many downstream errors. Code the fix for the first error, then rerun the program. Note that this is yet another argument for debugging incrementally.
- **Look for alternative coding methods.** If the code in question seems unreasonably complex or logically twisted, consider another approach. Well-written (and well-documented!) SQL code can replace a series of DATA and PROC steps.
- **Search other programs for similar problems.** If development and debugging programmers comment their code adequately, other programs become a valuable resource for problem resolution. If, for example, an algorithm fails in the program creating data set CM, the programmer can refer to the validated program that created data set VS, whose specifications required use of the same algorithm.
- **Look outside the program for sources of error.** The program may stop working due to undocumented or non-communicated changes in one of a host of external objects: format library, autocall path, CONFIG and other initialization files, and the like.

Additionally, here are some points specific to macro debugging:

- **Distinguish compile, execution time activities.** The macro code that passes syntax checks (%DOS and %ENDS match, %IF condition is followed by %THEN, etc.) may generate errors when the generated statements execute. Remember, too, that Base SAS statements that interact with the macro language (CALL SYMPUT, SQL SELECT INTO, and others) create macro variables that are not available until program execution is beyond the program step boundary. Writing a macro variable with CALL SYMPUT and attempting to access it in the same DATA step is a classic example of this kind of timing issue.
- **Consider scoping, open code.** Macro variables should be carefully scoped as global or local. While a global macro variable has the advantage of being *read* throughout the program, it also has the liability of being *altered* throughout the program. A %DO loop index variable I that is inadvertently GLOBALed is certain to come to harm when the macro uses another macro using I as a loop index – loop control of the calling macro is lost, and execution becomes unpredictable. Likewise, you must consider the distinction between “open code” (basically, code not contained within a macro) from macro code. *Some* macro functionality is available in open code: %GLOBAL; %LET (including references to functions %SYSFUNC and %QSYSFUNC); and others. There are no subtleties involved when you violate macro usage in open code, since messages during the program's syntax scan highlight the error.
- **Remember that macro syntax is somewhat different.** The underlying intent of macro language statements is similar to their Base SAS counterparts (iteration using DO, DO UNTIL, DO WHILE, IF-THEN-ELSE constructs, use of functions, etc.) There are, however, differences in syntax – some obvious, some subtle. The use of % and & clearly identifies macro code and usually generates errors if omitted (consider SAS's reaction to %DO IDX = 1 TO &NTOKEN;). Other, mostly quoting-related, syntax quirks can lead to lack of variable resolution or failed comparisons. Here are a few examples:

```
if '&month' = 'FEB' then return;
%if %sysfunc(indexW("YES NO", &dbg.)) = 0 %then %do;
%let x = &counter. + 1;
```

These statements should be rewritten as:

```
if "&month" = 'FEB' then return;
%if %sysfunc(indexW(YES NO, &dbg.)) = 0 %then %do;
%let x = %eval(&counter. + 1);
```

## SAS Tools

Programmers have *many* statements, functions, and options in Base SAS that facilitate macro debugging. This section identifies some of these, and presents scenarios showing their use. The descriptions are *not* exhaustive descriptions of syntax and usage. This is best left to the SAS online help (SAS 2003-2).



## Options

Some of these options should be set and unchanged throughout the life of the program (e.g., items related to autocall path and macro compilation). Others, however, can be turned on and off during the program, essentially shining a light on a problematic portion of code. Programmatic methods for controlling debugging output are discussed in “Technique 1: Conditional Execution of Debugging Output” and “Technique 2: Controlling the Amount of Debugging Output,” below.

Option	Comments
MERROR	Controls display of “unresolved macro reference” messages in the SAS Log. Turn off by setting to NOMERROR. It is always best to leave this option turned on.
SERROR	Controls display of “unresolved macro variable reference” messages in the SAS Log. Turn off by setting to NOSEERROR. It is always best to leave this option turned on.
MPRINT	Controls whether SAS code generated by the macro is displayed in the SAS Log. Turn off by setting to NOMPRINT. Can be turned on and off as needed within the macro. Problematic locations can be bracketed as follows: <pre>options mprint; problematic portion of the macro options NOMprint;</pre>
MPRINTNEST	Provides the same functionality as MPRINT, above, adding clearer identification of the macro calling hierarchy, or “nesting.” Turn off by setting to NOMPRINTNEST. Note that if MPRINTNEST is turned on, MPRINT must also be turned on.
MLOGIC	Displays information about the macro’s execution: start/end of macro execution; values of macro variables used in control/evaluation statements, true/false status of %IF statement conditions. Turn off by setting to NOMLOGIC. Output can become voluminous, to the point of being counterproductive, so turning the value on and off (see MPRINT example, above) is a good strategy.
MLOGICNEST	Provides the same functionality as MLOGIC, above, adding clearer identification of the macro calling hierarchy, or “nesting.” Turn off by setting to NOMLOGICNEST. Note that if MLOGICNEST is turned on, MLOGIC must also be turned on.
SYMBOLGEN	Displays the name and resolved value of macro variables. Turn off by setting to NOSYMBOLGEN. Output can become voluminous to the point of being counterproductive, so turning the value on and off (see MPRINT example, above) is a good strategy.
MFILE	Writes the program actually executed by SAS to an external file. Setting to MFILE requires: <ul style="list-style-type: none"> <li>• MPRINT is turned on</li> <li>• FILENAME MPRINT is defined.</li> </ul> Turn off by setting to NOMFILE. MFILE is particularly useful when the text generated by the macro in question – that is, the text passed to SAS for execution – needs to be captured. This enables you to run the program independently and macro-less, which makes SAS Log messages about line numbers and other locations easier to identify.
MAUTOLOCDISPLAY	Displays the source location of an autocall macro. Turn off by setting to NOMAUTOLOCDISPLAY. Useful for identifying where the macro came from. If a configuration or setup program changed the macro autocall path from “dir1”, “dir2”

Option	Comments
	to "dir2", "dir1" and DSETN.SAS was present in both directories, a program that expected to use the version in "dir1" would actually be using the macro stored in "dir2". Examination of where, exactly, a macro came from is important, especially when a project-specific autocall directory may be used with other, enterprise-wide directories.
SASAUTOS MAUTOSOURCE MRECALL MSTORED SASMSTORE	These options control the definition of the autocall path, including compiled macros. Examine the values (see PROC OPTIONS, below) to ensure that modified external macros are reloaded and that the autocall sequence is as expected.

## Statements

Of the numerous flow of control, macro definition, and other macro language statements, three stand out as vital to the debugging process.

Statement	Comments
%PUT	Displays text in the SAS Log. Its flexibility makes it ideal for diagnostic messages with a variety of content: <ul style="list-style-type: none"> <li>• Checkpoints in the program:  <pre>%put &amp;SYSmacroname.-&gt; Starting report-writing;</pre></li> <li>• Macro variable values. Always take the time to make the messages informative. Don't simply print values, but give some idea of the values' context.  <pre>%put &amp;SYSmacroname.-&gt; Print dataset &amp;i. of &amp;n.: &amp;tempDsetname.;</pre></li> <li>• Sets of variables:  <pre>%put _all_;  %put _global_;  %put _automatic_;</pre> Displays current values of all, global, and automatic macro variables, respectively.</li> </ul>
%GLOBAL	Defines one or more macro variables whose value will be available for reading and updating in open code or in any macro.
%LOCAL	Defines one or more macro variables that will be available only in the current macro. Values of like-named %LOCAL variables in multiple macros are maintained independently.

## Functions

The functions described in this section enable you to perform several important programming tasks. Some of the %SYMxxx functions help identify the presence of a macro variable. Rather than letting a macro fail because an expected variable was, in fact, not created, you can use, say, %SYMGLOBL to test for its presence and write a message if it was not found. %SYMDEL serves a useful, pre-termination clean-up role.

The quoting functions protect macro variable values from being evaluated as macro calls, arithmetic operations, and the like. See SAS 2003-3 for an in-depth discussion of this arcane but vital topic.

Function	Comments
%SYMEXIST	Returns a 1 or 0 if the specified macro variable is in either the local or global symbol tables. DATA step equivalent is the SYMEXIST function. This is a more coding-intensive but robust alternative to %put variable: <pre>%if %symExist(nobs) %then %put Found &amp;nobs. After filtering.;  %else %put NOBS was not defined.;</pre>
%SYMGLOBL	Returns a 1 or 0 if the specified macro variable is global in scope. DATA step equivalent is the SYMGLOBL function.
%SYMLOCAL	Returns a 1 or 0 if the specified macro variable is local in scope. DATA step equivalent is the SYMLOCAL function.
%SYMDEL	Deletes one or more macro variables. Useful for clean-up of unwanted global macro variables during macro termination.
%STR	Recall the earlier discussion of the macro processor's role of evaluating logical conditions and

Function	Comments
%NSTR %QUOTE %BQUOTE %NRQUOTE %NRBQUOTE %SUPERQ	<p>passing text to SAS for execution. There are cases where special characters (% - , = and others) in macro variable values need to be interpreted as characters rather than part of expressions. These functions, which would warrant a separate paper, control the interpretation of the macro variable values.</p> <p>The distinctions, while arcane, are often critical to debugging. Consider macro %DIRINFO, which accepts a directory name for parameter DIR:</p> <pre>%dirInfo(dir=c:\temp)</pre> <p>executes correctly, but</p> <pre>%dirInfo(dir=c:\temp\feb-09)</pre> <p>generates the error message</p> <pre>ERROR: A character operand was found in the %EVAL function or %IF condition where a numeric operand is required.</pre> <p>c:\temp\feb-09 is interpreted as an arithmetic expression due to the presence and misinterpretation of the intent of the minus sign. Using a quoting function prevents this error:</p> <pre>%if %quote(&amp;dir.) = %then %do;</pre>

## Procedures

While many procedures can be used to debug macro *results* (FREQ, CONTENTS on data sets created by the macro), OPTIONS has a special role in assisting the programmer who is trying to determine what the macro's execution environment looks like.

PROC	Comments
OPTIONS	<p>Displays current settings of SAS options in the SAS Log.</p> <p>Useful for getting a snapshot of an option value:</p> <pre>proc options option=symbolgen; R\run;</pre> <p>or for a group of related options (here, all those that are macro-oriented):</p> <pre>proc options group=macro; run;</pre>

## Building on the SAS Toolset

The options, functions, and other tools outlined in the previous section are powerful aids to both development and debugging programmers. Their output can, however, be verbose (e.g., MLOGIC), obscure (%PUT with no explanatory text), or just plain baffling (%PUT \_GLOBAL\_ seems to employ "maximum chaos" display rules).

This section describes some simple coding techniques that enhance and control debugging output. Although the code can be added as part of the debugging process, it is preferable to build it into the original program. This strategy expedites problem-solving, making debugging largely a matter of assigning parameter values or, at most, writing a small amount of additional code. Given the last-minute, borderline-panic atmosphere that often accompanies debugging, the less original coding that needs to be done, the better.

### Technique 1: Conditional Execution of Debugging Output

Suppose (and this is not hard to imagine) that some parts of a complex macro were problematic during development and occasionally produce odd results even after release into production due to unexpected data values, unanticipated parameter combinations or similar conditions. Using options and statements to untangle the program is straightforward:

```
options mlogic mlogicNest;
problematic code ...
data temp2;
  set master.ae;
  if _n_ <= 100 then put usubjid= visit= aesev=;
%put Dataset counter [&dsetN.];
options NOmlogic NOmlogicNest;
```

Once the problem is resolved, the question becomes what to do with the debugging statements. We don't want to clutter the Log with the now-unnecessary output from the diagnostic tools. Do we delete the OPTIONS, PUT and %PUT statements? Do we comment them out? The former is heavy-handed, the latter is tedious.

A third – and better – approach is adding a parameter to the macro and letting it control execution of the debugging statements. Code that was added to the previous example is boldfaced:

```

%macro createDomain(data=, debug=no);
%let debug = %upcase(&debug.); {1}
%local star opts; {2}
%if &debug. = NO %then %let star = *;
%else %do; {3}
    %let opts = %sysfunc(getoption(mlogic))
                %sysfunc(getoption(mlogicnest));
    options mlogic mlogicNest;
%end;
problematic code ...
data temp2;
    set master.ae;
    &star.if _n_ <= 100 then put usubjid= visit= aesev=;
%&star.put Dataset counter [&dsetN.];
%if &debug. ^= NO %then %do;
    options &opts.; {4}
%end;

```

Using the `DEBUG` parameter allows the user to turn statements on and off without having to manually modify the macro. Note these features referred to in the program:

- {1}** We immediately upper-case the parameter's value. This makes comparisons in `%IF` statements easier and relieves the user of the (admittedly minor) burden of having to remember whether the parameter is case-sensitive.
- {2}** We take control of the scope of variables related to debugging.
- {3}** Before turning on `MLOGIC` and `MLOGICNEST`, we save their original values in variable `OPTS`.
- {4}** Once we are past the problematic code that required `MLOGIC` and `MLOGICNEST` to be turned on, we return the options to their original setting.

Macro variable `STAR` equals `*` if `DEBUG` is `NO`, otherwise it is null. If the macro is invoked with `DEBUG=NO`, the following code is passed to SAS:

```

problematic code ...
data temp2;
    set master.ae;
    *if _n_ <= 100 then put usubjid= visit= aesev=;
    %*put Dataset counter [&dsetN.];

```

If `DEBUG` is not equal to `NO`, the macro passes these statements for execution:

```

options mlogic mlogicNest;
problematic code ...
data temp2;
    set master.ae;
    if _n_ <= 100 then put usubjid= visit= aesev=;
    %put Dataset counter [&dsetN.];
options original values of mlogic, mlogicNest ;

```

## Technique 2: Controlling the Amount of Debugging Output

A simple extension to the technique shown above is to use a debug parameter's value to control the amount of debugging output that will be produced. A bare-bones example follows:

```

%macro createDomain(data=, debugLevel=0); {1}
%put createDomain-> Begin. DATA [&data.] DEBUGLEVEL [&debugLevel.];

%local bad star;
%if &debugLevel. = %then %do; {2}
    %let bad = t; %put createDomain-> DEBUGLEVEL was null;
%end;
%else %if %sysfunc(anyDigit(&debugLevel.)) ^= 0 %then %do;
    %put createDomain-> DEBUGLEVEL [&debugLevel.] must be numeric 1->3;
    %put createDomain-> Execution terminating.;
%end;
%else %if %sysfunc(indexW(1 2 3, &debugLevel.)) = 0 %then %do;
    %let bad = t;
    %put createDomain-> DEBUGLEVEL [&debugLEVEL.] must be 1->3;
%end;

```

```

%if &bad. = t %then %do;
  %put createDomain-> Execution terminating.;
  %goto bottom;
%end;

%if &debugLevel. = 0 %then %let star = *; {3}
%if &debugLevel. >= 1 %then %do; {3}
  %let opts = %sysfunc(getoption(mlogic)) %sysfunc(getoption(mlogicnest));
  options mlogic mlogicNest;
%end;

data temp2;
  set master.ae;
  &star.if _n_ <= 100 then put usubjid= visit= aesev=;
%&star.put Dataset counter [&dsetN.];

%if &debugLevel = 3 %then %do; {3}
  proc freq data=temp2;
    tables trtGrp * aesev;
    title "TEMP2 distribution of treatment X severity";
  run;
%end;

%if &debugLevel. >= 1 %then %do; {3}
  options &opts.;
%end;

%bottom: %put createDomain-> End;
%mend createDomain;

```

Using the `DEBUGLEVEL` parameter requires more coding effort by the developing and/or debugging programmer, but has the benefit of providing greater control output. The choice of what warrants inclusion in levels 1, 2, and 3 is, of course, arbitrary. The point here is simply that this technique *allows* the decision to be made. Note these features referred to in the program:

- {1} We use the parameter name `DEBUGLEVEL` rather than `DEBUG`, used in the previous example. If this coding technique is used throughout a system of macros, it is easier for the macro user to remember that `DEBUG` takes a YES/NO value, while `DEBUGLEVEL` is numeric. This is a minor usability feature, but as experience bears out, "many minors = near major" in the long term.
- {2} `DEBUGLEVEL` is put to several tests: null, non-numeric value, value out of range. If any test fails, we write a message to the SAS Log and terminate. This testing requires extra coding, but is worth the effort, since invalid values are caught and explained. Without such checks in place, an incorrect value would not be noticed and could produce spurious results.
- {3} Throughout the program, we evaluate `DEBUGLEVEL` and produce output based on its value.

### Technique 3: Return Codes

A well-written, "non-simple" macro should make liberal use of specialized, single-purpose utility macros. The availability of a library of tools to count observations, create lists of data set variables, and the like allows the programmer to build applications more rapidly and more reliably. A hallmark of a good utility's design is its ability to communicate its completion status to its calling program. One source of macro design error is the failure to use return codes. This omission can cause the program to "die noisy" rather than fail gracefully. Consider the following macro:

```

%macro dsetN(data=, count=nObs, rc=RC_dsetN)
  %global &rc. &count.;
  if dataset not found or cannot be opened, assign
  RC variable = 1. Otherwise, RC variable = 0
%mend dsetN;

```

Here is a fragment of a calling program that does *not* effectively use the macro:

```

%do idx = 1 %to &dsCount.;
  %let dset = %scan(&dsList., &idx.);
  %dsetN(data=&dset.)
  proc print data=&dset.(obs=10);
    title "First 10 obs (of &nObs.) from &dset.";
  run;

```

```
%end;
```

The program neglects to use the return code created by %DSETN. If the data set in the list cannot be found or opened, PRINT will issue an error message. A few simple modifications to the program use the return code to better control program flow. The result is output that is error-free and has more professional look and feel.

```
%do idx = 1 %to &dsCount.;
  %let dset = %scan(&dsList., &idx.);
  %dsetN(data=&dset.)
  %if &RC_dsetN. = 0 %then %do;
    proc print data=&dset.(obs=10);
      title "First 10 obs (of &nObs.) from &dset.";
    run;
  %end;
  %else %put WARNING: Could not locate/read data set &dset.;
%end;
```

#### Technique 4: Revision Codes

Debugging relies in varying proportions on the programmer's knowledge of the language and his/her intuition. Once the problem is resolved, both aspects of the solution should be documented. The program header should be updated and comments should be inserted in the locations where changes were made. Most importantly, the modification should be uniquely numbered and descriptive. Program excerpts follow:

```
/*
  Revision History
  FCD 09-08-22 [U01] Correct LOCF algorithm when no qualifying visits
*/

data vs;
  merge dm(in=_dm)  raw.vs(in=_vs);
  by usbjID;
  retain _orres 0; /* [U01] Explicitly set to 0 */
run;

data sdtm.VS;
  set vs2;
  other statements related to [U01]
run;
```

The debugging programmer made changes in several locations. The header comment summarized the nature of the change (correct a flaw in the LOCF algorithm) and assigned an identifier ([U01]). Whenever a change was made to the program, the programmer inserted a comment containing, at minimum, the change code.

This seems like a lot of work. We are, after all, adding only comments, not code, to the program. The payoff comes later. If the programmer has a similar problem with the LOCF algorithm in another program, he/she can refer to this program, search for [U01] and quickly get an idea of how to proceed. Using comments with revision codes, in effect, adds to the enterprise-level knowledge of how a particular type of problem was resolved. And if that fails to convince you to take the time to use this commenting technique, consider the all-too-common alternative:

```
/*
  Revision History
  FCD 09-08-22 Corrected LOCF
*/
```

#### Home-Grown Tools

The previous sections of this paper described some of the built-in SAS tools and simple coding techniques that the programmer can use for debugging. As broad-based and powerful as these features are, they are sometimes outstripped by the demands and complexities of the applications using them. This gives rise to the need for "home-grown" tools that are not simply implementations of the native SAS tools, but complete, standalone utilities that assist macro developer and debugger alike.

This section describes several such tools. The examples are of varying complexity in order to show that even simple macros with few "moving parts" can assist in error diagnosis and prevention. For each, we present a short discussion of why the tool was needed, a generalized description of how it works, and an example of its use. Some features of these and similarly effective utilities are that they are:

- **Well-documented.** The purpose, usage, and output of the macro must be clearly understood. A well-documented program header is essential. If necessary, external documentation (web page, PDF, etc.) should supplement documentation contained in the header.
- **Available.** The macro should be accessible without extra programming (%INCLUDE, option setting, etc.) by the user. Placement in an autocall library serves this need.
- **Configurable.** Not all of the macro's diagnostic capabilities are needed every time it is invoked. The macro should have parameters that control the amount and appearance of output. This would, for example, give the user the ability to request display of a complete set of variable attributes – STYLE=FULL – or a more limited selection – STYLE=COMPACT.
- **Without Impact.** The macro should call to mind the Hippocratic Oath: "Above all, do no harm." A diagnostic tool that overwrites WORK directory data sets or leaves behind unwanted global macro variables creates more problems than it purports to solve. The macro should promise a set of outputs and, upon completion, leave *nothing behind in the SAS session but those outputs.*

*Note: The size of these macros makes their display in this paper impractical. Unless otherwise noted, the macros and programs demonstrating their use can be found in the conference Proceedings CD and the CodeCrafters, Inc. web site. (See the "Contact" section, below, for details.)*

### Tool 1 – %listMacvars: Macro Variable Printer

**Background.** There are many times in the life cycle of a macro when the programmer says what amounts to: "There are macro variables that should or should *not* be here. I've lost track of what I'm doing." The quick-and-dirty way to display variables is a %PUT statement followed by a keyword specifying the variables to list. The problem with this is that the display is ragged and the variables are not listed alphabetically. This is not too much of an issue when displaying a small number of variables, but when the number is large (or your tolerance for ragged displays is low), the default %PUT output verges on *disinformation*.

What's needed is a macro that will display macro variables neatly arranged in alphabetical order.

**Solution.** The macro's design is simple:

- Perform housekeeping chores: upper-case parameter SCOPE (controls which variables are displayed); store line size in local macro variable LS.
- In PROC SQL, read metadata table DICTIONARY.MACROS (Dilorio 2005-2 discusses the use of SAS metadata, aka "Dictionary Tables and Views"), creating data set \_MACVARS\_. Filter by SCOPE parameter and OFFSET=0 and sort by macro variable name.
- Store the maximum length of the variable name in local macro variable MAXLEN.
- In a \_NULL\_ DATA step, read \_MACVARS\_. Display name in columns 1 through &MAXLEN and value in columns &MAXLEN + 2 through &LS. This opens up the possibility of truncation of long values. This limitation should be documented in the macro header.
- Delete data set \_MACVARS\_

**Example.** Define nine global macro variables. %put \_global\_; output is shown below. The display is difficult to read even with a single-digit number of variables:

```
GLOBAL MARGINS 1 1 1 1
GLOBAL DSETLIST ae cm dm vs suppa relrec
GLOBAL DSETLISTN 6
GLOBAL PT 10
GLOBAL IGVER 312
GLOBAL ORIENT port
GLOBAL FONT Times Roman
GLOBAL CHECKTYPES d m
GLOBAL DATETIME Friday, 10JUL2009 07:20
```

To fix the problem, use %listMacvars:

```
-----
Variable      Scope [_GLOBAL_]  First 73 Characters
-----
CHECKTYPES    d m
DATETIME      Friday, July 10, 2009 2:09 PM
DATETIMEN     1562854148.516
DSETLIST      ae cm dm vs suppa relrec
DSETLISTN     6
FONT          Times Roman
```

```
IGVER          312
ORIENT        port
PT            10
# of variables = 9
```

---

## Tool 2 – %whatChanged: Identify Changes Made by the Macro

**Background.** Recall the “Above all, do no harm” admonishment at the beginning of this section. *Any* macro, particularly one which is supposed to help diagnose problematic code, should produce *only* the macro variables, data sets, etc. listed in the macro’s header documentation. Leaving “stray” data sets in the macro’s wake can alter the calling program’s data set counts and, in turn, cause it to produce incorrect results.

The question then becomes how, exactly, can you know what the macro changed? For some artifacts, such as data sets, we can compare PROC DATASETS output before and after the macro’s execution. Likewise, option settings can be compared by examining PROC OPTIONS output or, taking a slightly more sophisticated approach, comparing PROC OPTSAVE output data sets. Regardless of approach, these methods are tedious, error-prone, and not a good use of a programmer’s time.

What’s needed is a macro that compares option settings, data set names, and other aspects of the SAS environment before and after execution of the macro. The utility’s list of changed or created artifacts can then be reconciled with what was expected (i.e., those listed in the program header).

**Solution.** The macro is called twice: before and after the macro being examined. While the program flow described here is specific to global macro variables, it can readily be extended to other items such as data sets, system options, and the like.

Call 1, before the macro:

- Perform housekeeping chores: upper-case parameters ACTION (START or END) and COMPARE (any or all of DATA MACVAR OPT, separated by one or more blanks)
- Evaluate parameter values. Write message and terminate if invalid or inconsistent (e.g., END specified in a call before START)
- In PROC SQL, read DICTIONARY.MACROS where SCOPE=GLOBAL. Rename variable VALUE to VALUE\_START. Save as data set \_CHG\_MACVARS\_START.

Call 2, after the macro:

- Perform housekeeping chores: upper-case parameter ACTION (START or END)
- Evaluate parameter values. Write message and terminate if invalid or inconsistent (e.g., END specified in a call before START)
- Test for presence of data set \_CHG\_MACVARS\_START (created by call to %whatChanged where ACTION=START). Write message and terminate if it is not found.
- In PROC SQL, read metadata tables DICTIONARY.MACROS where SCOPE=GLOBAL. Save as data set \_CHG\_MACVARS\_END.
- In a \_NULL\_ DATA step, merge \_CHG\_MACVARS\_START and \_CHG\_MACVARS\_END by macro variable name. Write to the SAS Log, listing observations where VALUE does not equal VALUE\_START or where the variable name is not in both data sets, indicating creation or deletion. Also report the *absence* of discrepancies if all values matched.
- Delete data sets \_CHG\_MACVARS\_START and \_CHG\_MACVARS\_END.

**Example.** Macro %dsetList contains this text in its header documentation:

```
Output:  Macro variables
         DSET   Blank-delimited, upper-cased list of
           qualifying data sets
         DSETN  Number of items in variable DSET
```

Test %dsetList to verify only these macro variables were created.

```
statements allocating and creating data sets
%whatChanged(action=start, compare=macvar)
%dsetList(data=_all_)
%whatChanged(action=end)
```

Output from %whatChanged reveals the presence of an unexpected variable (I). %dsetList should be modified and the test program rerun until only the expected global macro variables DSETLIST and DSETLISTN appear in the variable list.

Global macro variables

Action	Variable
Added	DSETLIST
Added	DSETLISTN
Added	I

### Tool 3 – %multUpper: Upper-Case a List of Macro Variables

**Background.** The case, so to speak, has already been made for standardizing parameter values. Failure to do so gives rise to scenarios where the macro appears to do nothing because the %IF-%THEN statements controlling flow evaluated upper-cased text and the user entered a legitimate, but lower-cased, value. A series of %LET statements that use upper-casing function can avoid these problems. If there are many parameters, however, the repetition is, well, repetitive, and clutters the program.

What is needed is a macro that upper-cases a list of macro variables.

**Solution.** The macro’s design is simple:

- In a %DO %UNTIL loop, scan parameter VARLIST. If the token returned is missing, terminate.
- Evaluate the token with the %SYMGLOBL function. If the token is recognized as a global macro variable, upper-case it. Otherwise, write a message to the Log and continue execution.

**Example.**

```
%macro rpt(data=, orient=, nobs=, rpttype=compact, debug=no);
  %multUpper(varlist=data orient nobs rpttype debug)
```

### Tool 4 – %macXref: Macro Cross-Reference

**Background.** Sometimes macro “X” becomes problematic not due to errors in the macro or changes to the data it handles. Instead, a change in macro “Y”, which is called by “X” can introduce unforeseen problems. Integration testing before release to production thwarts most, but not all, of these errors. The programmer charged with maintaining macro “Y” should test the revision’s impact on all macros – “X” and others – that use it.

What is needed is a macro that identifies macro usage and dependencies: “Macro X Uses Which Macros” and “Which Macros Use Macro Y?”

**Solution.** The macro’s design is non-trivial (*Note that the source for this macro is not available on the Proceedings CD or on the CodeCrafters, Inc. web site*):

- Key assumption is that macros are stored in individual .SAS files. Compiled macros are *not* processed.
- Evaluate parameters
  - SEARCH: directories to use in the cross-reference (default is non-SAS directories in autocall path)
  - REF: list of directories containing macros that will be referenced by directories identified in the SEARCH parameter, above. REF has same default value as SEARCH.
- Use native OS commands (e.g., DOS DIR) to identify .SAS files in the directories identified by the REF and SEARCH parameters. Store path, file name, and other attributes in data sets REF and SEARCH.
- Read each file found in REF and SEARCH
  - Identify context (whether the macro is being referenced or defined)
  - Search for referenced macros. Exclude references to SAS macro functions (%LET, %SYSFUNC, etc.) and SASAUTOS macros (%CMPRES, %DATATYP, etc.)
  - Append to data set ALLMACS (one observation per macro reference)
- Sort ALLMACS by file name and macro name
- Create report “Macro X Uses Which Macros”
- Sort ALLMACS by macro name, file name, context (referenced/defined)
- Create report “Which Macros Use Macro Y”

Correct extraction of the macro name from the input SAS programs is critical, as is identifying the referenced/defined context. Once this is correct, the reporting becomes comparatively easy (%macXref produces both hyperlinked HTML and PDF reports).

An experimental extension to %macXref adds reporting of the macro calling hierarchy. The report will answer a third, even more important, question "What Is the Complete Calling Sequence of Macro X?" It will show macro X calling macro Y, then examine macro Y's dependencies, presenting output in a format such as:

```
X    →    Y    →    Z
    →    A
    →    D    →    E
```

**Example.** Partial output of a call to %macXref is shown in **Figure 1**, below.

**Figure 1: %macXref Output**

File Name	Path	Macros Defined	Macros Used
ALLMISSING	macro\util\allMissing.sas	ALLMISSING	CLEARWORKDATASETS DELMACVARS DSETATTRIBS MEMLIST NLEVELS PUTSKIP QUIET VENN
ALLOCLIB	macro\util\allocLIB.sas	ALLOCLIB	DELMACVARS DISTINCT MULTUPCASE PUTSKIP QUIET QUOTELIST TIMER VENN
BUILDCONFIG	macro\config\buildConfig.sas	BUILDCONFIG	CHARLOC CHECKVAR LISTCONFIG PUTSKIP QUIET
BUILD_SPECDATA	macro\config\build_SpecData.sas	BUILD_SPECDATA	COMPAREDATES DSETN EMAILNOTIFICATION EMPH PREPEND TIMER VARLIST
CHARLOC	macro\util\charLoc.sas	CHARLOC	
CHECKANLMETADATA	macro\config\checkANLmetadata.sas	CHECKANLMETADATA	CHECKVAR CURRENT DELETEDATA DELMACVARS DISTINCT DSETN GETDATE MACROLOC METADIFFS MULTUPCASE PDFSETUP PUTSKIP QUIET QUOTELIST REPORTSTRING STYLE
CHECKISO8601	macro\util\checkISO8601.sas	CHECKISO8601	
CHECKMETA	macro\config\checkMeta.sas	CHECKMETA	EMPH
CHECKVAR	macro\util\checkVar.sas	CHECKVAR	INITGLOBAL PREFVAL
CLEARWORKDATASETS	macro\util\clearWorkDatasets.sas	CLEARWORKDATASETS	DELMACVARS EMPH MEMLIST MULTUPCASE SETOBS
COMPAREDATES	macro\util\compareDates.sas	COMPAREDATES	GETDATE
COMPAREVARS	macro\util\compareVars.sas	COMPAREVARS	CONT CURROIR DELETEDATA DELMACVARS DISTINCT DSETN INITGLOBAL LIBINFO MACROLOC MULTUPCASE PDFSETUP PRINTMACVAR PUTSKIP QUIET QUOTELIST SETOBS
CONT	macro\util\cont.sas	CONT CONTTEMP	CONTTEMP DELMACVARS MULTUPCASE QUIET SETOBS SPLITNAME
CONVERT	macro\config\convert.sas	CONVERT REDDEN	CHECKVAR COMPAREDATES CONT CURRFILE DSETN EMPH IDENTIFYCHANGES MULTUPCASE PREPEND READMDS TIMER
CONVERTREF	macro\config\convertREF.sas	CONVERTREF	DELETEDATA DSETN QUIET
COPYDATA	macro\util\copyData.sas	COPYDATA	DELMACVARS MEMLIST QUIET
CREATEMDB	macro\util\createMDB.sas	CREATEMDB	QUIET

## Conclusion

This paper has presented a collection of strategies, caveats, SAS and home grown tools for debugging programs using macros. While the selection may be idiosyncratic, a few points rise above the ideological fray:

- The speed and success of the debugging effort is determined in varying measures by your intuition and technical expertise. As program complexity increases, so does the reliance on intuition. The inherent complexity of macro debugging thus places a greater emphasis on intuition and the breadth and depth of your problem-solving experience.
- There are *many* tools built into the macro language that assist debugging. You should be aware of their use and be able to build simple debugging tools into your macros.
- There is often a need for tools and utilities that are either not part of the SAS macro language or easily constructed. Taking the time to create tools that make the initial macro more robust or assist debugging has long-term benefits.

- Always keep in mind what distinguishes the macro language from the rest of the SAS language (issues of timing, scope, and so on). These differences give the language its power and are also usually the root cause of program failure.
- Finally, remember that programmers are people who enjoy unraveling mysteries and discovering patterns of program behavior. This activity may not be regarded as entertainment if you're fighting a deadline at 2 a.m., but on balance the macro language provides abundant opportunity for both developing defensive coding techniques and program sleuthing. It should always be a learning experience, and it should *almost* always be fun.

## Contact

Your comments and questions are welcomed and encouraged. Contact the author at [frank@CodeCraftersInc.com](mailto:frank@CodeCraftersInc.com).

You are also encouraged to visit the CodeCrafters, Inc. web site: <http://www.CodeCraftersInc.com>. The site has other papers by the author, code samples, and many links to other professional resources. Note, too, that this paper is often presented at SAS user group meetings. As such, it is a "living document," and undergoes frequent revisions. The web site will always have the "latest and greatest" version of the paper.

## Acknowledgements

Thanks to Jeff Abolafia and Robert Schechter for reviewing this paper for content and format. Thanks, as always, to my wife, April Sansom, for her proofreading and editing assistance.

## 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.

## References

Note: Items in braces refer to the conference at which the paper was presented. See the web site for the user group (SESUG, SAS/SUGI, etc.) or link via <http://www.LexJansen.com>. Another good source of macro wisdom and advice is the SAS discussion group – SAS-L – found at <http://www.listserv.uga.edu/archives/sas-l.html>, <http://groups.google.com/group/comp.soft-sys.sas/topics?hl=en>, among other locations. Also see articles and code at [http://www.sascommunity.org/wiki/Main\\_Page](http://www.sascommunity.org/wiki/Main_Page).

Adams, John, and Gratt, Jeremy. 2005. "Large Scale Standard Macros: A Methodical Approach to Development and Implementation." {NESUG}

Carpenter, Arthur L. 2002. "Building and Using Macro Libraries." {SUGI}

Carpenter, Art. 2004. *Carpenter's Complete Guide to the SAS Macro Language (Second Edition)*. Cary, NC: SAS Institute Inc.

Carpenter, Arthur L. 2005-1. "Make 'em %LOCAL: Avoiding Macro Variable Collisions." {WUSS}

Carpenter, Arthur L. 2005-2. "Job Security: Using the SAS Macro Language to Full Advantage." {PharmaSUG}

Cooper, Anthony. 2007. "Debugging Standard Macros." {PhUSE}

Delaney, Kevin P., and Carpenter, Arthur L. 2004. "SAS Macro: Symbols of Frustration? %Let us help! {PharmaSUG}

Dilorio, Frank. 2008. "Building the Better Macro: Best Practices for the Design of Reliable, Effective Tools." {NESUG}

Dilorio, Frank. 2006. "Controlling Macro Output or, 'What Happens in the Macro, Stays in the Macro'." {SESUG}

Dilorio, Frank. 2005-1. "Rules for Tools - The SAS Utility Primer." {PharmaSUG}

Dilorio, Frank and Abolafia, Jeff. 2005-2. "Dictionary Tables and Views: Essential Tools for Serious Applications." {SESUG}

Dilorio, Frank 2004. "The SAS Debugging Primer." {SUGI}

diTommaso, Dante. 2005. "Macro Design: Development by Example." {PharmaSUG}

Dunn, Toby. 2008. "Macro Quoting." {SESUG}

Dunn, Toby, and Whitlock, Ian. 2007. "Beginning Macro Design Issues." {SCSUG}

First, Steven. 2001. "Advanced Macro Topics." {SUGI}

Heaton-Wright, Lawrence. 2007. "Anticipating User Issues with Macros." {PhUSE}

Ivis, Frank. 2004. "Guidelines on Writing SAS Macros for Public Use" {SUGI}

Jackson, Clarence. 2006. "SAS Macro Language Quick Study" {SCSUG}

Lafler, Kirk Paul. 2008. "SAS Macro Programming Tips and Techniques." {SESUG}

Ratcliffe, Andrew. 2001. "Debugging Made Easy." {SESUG}

Redner, Ginger, and Zhang, Liping, and Herremans, Carl. 2008. "Techniques for Developing Quality SAS Macros" {SESUG}

SAS Institute. "General Macro Debugging Information." SAS 9.1 Macro Language: Reference. 2003-1. <<http://support.sas.com/onlinedoc/913/getDoc/en/mcrolref.hlp/a001302411.htm>> July 10, 2009

SAS Institute. "Macro Language Dictionary." SAS 9.1 Macro Language: Reference. 2003-2. <<http://support.sas.com/onlinedoc/913/getDoc/en/mcrolref.hlp/a001302411.htm>> July 10, 2009

SAS Institute. "Macro Quoting." SAS(R) 9.1 Macro Language: Reference. 2003-3. <<http://support.sas.com/onlinedoc/913/getDoc/en/mcrolref.hlp/a002047063.htm>> July 10, 2009

Scocca, David A. 2003. "The Science of Application Debugging." {SGF}

Sherman, Paul D. 2007. "SAS/MACRO NOTES: Lines and Columns in the Log." {SESUG}

Stroupe, Jane. 2003. "Nine Steps to Get Started using SAS Macros." {SUGI}

Tabladillo, Mark. 2005. "Macro Architecture in Pictures." {SUGI}

Tabladillo, Mark. 2005. "SAS Macro Design Patterns." {SESUG}

Truong, Sy. 2004. "How to Develop User Friendly Macros." {PharmaSUG}

Whitlock, Ian. 2008. "The Art of Debugging." {SGF}

Whitlock, Ian. 2005. "Macro Bugs - How to Control Them." {NESUG}

Whitlock, Ian. 2004-1. "A Second Look at SAS Macro Design Issues." {SUGI}

Whitlock, Ian, and McMullen, Quentin. 2004-2. "Macro Power." {NESUG}

Whitlock, Ian. 2002. "SAS Macro Design Issues." {NESUG}

Whitlock, Ian. 1999. "Getting Started with Macro." {SESUG}

Winn, Thomas J. 2001. "Introduction to the SAS Macro Language." {SESUG}