

A Debugging Facility for SAS® Macro Systems

David S. Frankel, Exxon Company, U.S.A.
Mark. A. Kochanski, Halliburton Energy Services

Abstract

This debugging facility makes it easy to trace the execution of one or more SAS macros within a SAS job. You can control printing start- and end-banners that identify macros, including revision date and support group; checking the values of macro parameters; printing notes following PROC and DATA steps; printing a simple or detailed source-code trace; and, deletion of temporary SAS data sets. Macros not to be traced execute "as is" with settings determined by their invocations and definitions.

The facility consists of two SAS macros -- %DEBUG and %_DEBUG. An end-user or developer calls %DEBUG from a SAS job to set the debugging options to be used for subsequent macro calls. Each SAS macro calls %_DEBUG to print banners, set status variables, delete temporary SAS data sets, and replace statements like OPTIONS NOTES. %_DEBUG determines the environment for each macro by arbitrating between "as is" settings and %DEBUG user requests.

Both end-users and developers have benefited from this facility--especially for highly nested systems. End-users control the SAS log file by printing only what they need to easily find where errors are occurring. Developers get a detailed trace of specific macros to pinpoint bugs without editing macro definitions, as well as simplified and standard debugging statements for developing macros.

Background

Developers who write macros have always had conflicting goals as to what the macros should write on the log file. The user typically is not concerned with the internal workings of the macro, and wants to see a minimum number of lines. However, if an execution problem is encountered, the developer needs to see the statements generated by macro execution.

The most primitive approach to this issue is coding an OPTIONS statement as the first and last executable statement in each macro definition. The production-level macro definition sets NOSOURCE NONOTES at entry and SOURCE NOTES at exit to avoid cluttering the log file. If an execution problem is encountered, the developer first edits the macro definition to set SOURCE NOTES at entry, then codes OPTIONS MPRINT ahead of the macro invocation.

A slightly more sophisticated approach is defining a macro parameter whose value controls the entry and exit OPTIONS statement. This lets the macro invocation control the "trace" environment for that macro and avoids the need to edit that macro definition.

Both approaches are adequate when a SAS job invokes a single macro. But when a system of nested macros is executing, these simple approaches are inadequate.

Consider a simple two-macro system comprising an outer macro that invokes an inner macro. Tracing the execution of the entire system forces you to edit both macro definitions. Even if your trace control is a macro parameter, you still must edit the outer macro definition to get at the trace control for the inner macro. Also the OPTIONS statement issued at the exit of the inner macro will control the trace environment of the outer macro. To re-assert control of its trace environment, the outer macro needs

to issue an OPTIONS statement immediately after invoking the inner macro.

The situation grows more complex with deeper nesting. But nesting is the rule rather than the exception.

Our experience shows that macro systems provide an efficient way to deliver standardized, flexible, optimized, reliable, and user-friendly SAS applications to a large number of users. Within the system, each macro performs a single task. This may be data retrieval, validation, shaping, analysis, display, or reporting. When a user invokes a macro, he is normally executing a macro system that is often nested three levels deep and may be nested as much as six or seven levels deep.

The simple, intuitive approaches to managing the debug environment are grossly inadequate for nested macro systems. We developed a facility that manages the debug environment for each executing macro. You can control these individual environments without having to edit either the macro definitions or the macro invocations.

Debug-Facility Concepts

The facility consists of two macros stored in an autocall source library: %DEBUG and %_DEBUG. End-users only need to know about %DEBUG; developers need to know about both.

The end-user invokes the %DEBUG macro to specify any combination of five debug actions to apply to targeted macros. These actions are: writing entry/exit banners; verifying data; writing notes for DATA and PROC steps; writing a source-code trace; and, saving or deleting work data sets. If %DEBUG is not invoked or if a macro is not targeted by %DEBUG, the debug environment is said to be "as-is."

The developer invokes the %_DEBUG macro as the first and last executable statement in each macro definition. This delegates control of that macro's debug environment to the debug facility. In return the debug facility makes information available to that macro during execution.

Banners

A banner may be written to the log file when a macro starts executing (the entry banner) and again when it stops executing (the exit banner). The entry banner has two lines. The first gives the macro's name and version, which normally includes the date of last modification. The second line identifies the support staff. The exit banner is a single line that names the macro that is ending execution. All banner lines begin with 'NOTE:', which makes it possible to suppress them with a NONOTES system option.

You can locate an error within the macro system simply by turning on all banners so as to bracket the SAS error messages. The entry banners are essential to verify that you are executing the proper version of the macro. Quite often you get the wrong version if the SASAUTOS file reference is associated with a concatenation of several macro source libraries.

According to our macro-writing guidelines, every macro definition has a keyword parameter that lets you toggle the banners on or off. The normal

default is on for "user-level" macros that are commonly invoked directly by end-users, and off for "utility" macros that are most often invoked from within other macros.

Verifying Data

Ironclad applications need to verify all input data. For macros this means checking the value of every macro parameter at execution time. Each of our macros contains a conditional block of code that invokes utility macros to verify:

- syntax (SAS names, host names, integers, date literals)
- existence (SAS data sets and variables, DB2 tables and columns)
- type (character or numeric)
- set functions (list, reserved, intersection, no intersection)
- range (floating point)

When a value fails a test, the verification macro writes an explanation on the log file, and the host macro sets a return code and exits without attempting execution.

Verification does trap most input-data errors and indicates how to correct them. However, it consumes time. The ability to toggle verification is helpful to speed execution when you are sure the input data are good or when you are invoking a macro within a loop.

Just as for banners, every macro definition has a keyword parameter that lets you toggle verification on or off. The normal default is on for user-level macros and off for utility macros.

The debug facility mediates between the user request (if any) and the as-is state to determine whether or not each macro should execute its internal verification code. The result of this determination is preserved as part of each macro's personal debug environment and is available to the executing macro as the value of a global macro variable.

Notes

The NOTES system option controls whether or not notes from DATA or PROC steps are written on the log file. The end-user only wants to see notes from the ultimate DATA or PROC step and from resource-intensive intermediate steps. The developer identifies these steps when coding the macro definition. However, the debug facility lets you suppress or activate all notes in targeted macros.

Source-Code Trace

The debug facility lets you select a simple or a detailed trace for targeted macros. A simple trace writes the text generated by macro execution to the log file. A detailed trace writes the macro code as it resolves during execution.

The default is no trace. The end-user does not want to see any code produced by the macro system. He is only interested in knowing that he got the correct version of the macro and that it produced the desired results. If the desired results are not produced, the developer will localize the problem and request a trace to see if the code is at fault. If so, the macro definition must be edited to correct the error.

Deleting Data Sets

Many macros create intermediate SAS data sets before completing their tasks. Some developers put these in the library associated with the WORK libref, using member names that begin with the underscore character to avoid interfering with the user's data. Other developers prefer to define a separate, temporary SAS library to ensure adequate space and to preclude interfering with the user. In both cases the work data sets should be cleared when the macro ends.

However, sometimes the developer needs to examine these work data sets. For example a macro may produce questionable results even though no syntax or execution errors are logged. You should check the intermediate data sets for unanticipated features such as duplicate observations. This check may reveal logic errors in the code.

In each macro definition the developer identifies the list of data sets to be deleted and libraries and files to be cleared when the macro finishes executing. The %_DEBUG macro does the actual deleting and clearing unless the user has requested otherwise with the %DEBUG macro.

The End-User's Perspective

The end-user invokes %DEBUG to change debug environments without editing macro definitions. Each invocation targets one or more macros for action, and specifies the settings for any combination of the five debug actions. Three of the five actions have on, off, and as-is modes: banner; verification; and notes. The trace action has off, simple, and detailed modes. The data-set option has delete and save modes.

You can invoke %DEBUG more than once in a job. These compound requests are stacked, with the most recent request taking precedence over earlier requests when the same macro is targeted more than once. Invoking %DEBUG with no parameter values clears all previous requests.

You are not required to invoke %DEBUG in each job. If you do not invoke %DEBUG, all macros execute with as-is debug environments. Likewise, when you do invoke %DEBUG, those macros not targeted execute as-is.

Targeting Macros

Targeting means identifying to the debug facility the group of macros whose debug environments you want to change. You can either list the macro names, or you select one of these categories: all; outer (one level removed from the base environment); inner (more than one level deep in the nest); user; or, utility.

Each macro identifies itself to the debug facility as a user, utility, or function macro. A user macro is commonly invoked directly by an end-user, and a utility macro is commonly invoked by a developer from within a higher-level macro. A function macro is a special kind of utility macro that generates text that is less than a complete statement. Usually it produces a single word that is interpreted as a value within an expression where it is invoked.

The As-Is Environment

As-is always means that work data sets are deleted when a macro ends, and that there is no source-code trace.

For the NOTES action, as-is reflects the way the developer coded the macro definition. If the developer did nothing, as-is means NONOTES. However, the developer normally will tell the debug facility to write notes for those DATA or PROC steps that the user may be concerned about. In such cases, as-is means NOTES for only those steps explicitly requested by the developer.

Each macro definition has keyword parameters to toggle banner and verification. The macro definition declares default settings, but these defaults may be overridden by the invoking statement. Consequently, as-is for these two actions reflects the settings declared in the invocation, superimposed on the default settings in the macro definition.

The Developer's Perspective

If you code macro definitions, you must invoke the %_DEBUG macro as the first and last executable statement within each macro definition. This allows the debug facility to manage each macro's debug environment.

The %_DEBUG invocation at the start of each macro specifies to the debug facility the macro name, category (user, utility, or function), version, and support string. The as-is values of banner and verification requests are also passed from the macro to the facility.

The %_DEBUG invocation at the end of each macro names the WORK data members to be deleted, and/or librefs and filerefs to be cleared.

You also request notes for DATA or PROC steps of interest to the user by invoking %_DEBUG before and after these steps to toggle notes on and off, respectively.

Feedback from the Debug Facility

%_DEBUG maintains global variables that can be used by the invoking macro.

The _CHECK_ variable indicates whether or not verification of parameter values should be done. If verification is on, the invoking macro (not %_DEBUG) must call the utility macros needed to check its input.

The _TRACE_ variable indicates whether the source-code trace is off, simple, or detailed. You may want to code customized debug actions in your macro that execute when a simple or detailed trace has been requested. These actions may be %PUT or PUT statements or descriptive PROC's like MEANS, FREQ, or UNIVARIATE.

The _NAMSTAK variable lists the names of macros in the invoking chain. The name of the immediate parent is separately assigned as the value of _MACNAM_.

Knowing the invoking chain is especially useful for the utility macros that handle verification functions. When these macros detect a fault, they must be able to write notes to the log file identifying the macro associated with the faulty data. Often the macro with a problem parameter is two or even three levels above the macro that detects the problem.

How the Facility Works

The debug facility has four tasks:

- communicating user requests into the executing environment, and imposing these requests on the as-is conditions;
- preserving each executing macro's debug environment;
- executing debug actions: %PUT banners; issue OPTIONS statements for notes and trace features; delete work members and clear work libraries;
- informing the executing macro about its debug environment to facilitate conditional debug actions handled by that macro, e.g. verification and custom trace actions.

These tasks are accomplished using 12, reserved, global, macro-variables that can be classified into three groups:

- seven variables used as stacks to communicate compound user-requests to the facility;
- two variables used as stacks for individual debug environments;
- three variables used to inform the executing macro about its debug environment.

Stack operations are emulated with macro character functions. New values are pushed onto the stacks by concatenation, with the new string on the left and the old string on the right. Old values are popped off the stack with the %SUBSTR function or the %SCAN function.

The debug environment for each macro is defined by single-character values for each of five macro variables corresponding to the five debug actions: banner; verify; notes; trace; and save/delete work data sets. Of these five macro variables, three are local to the %_DEBUG macro, and two are global.

The debug environment also has a macro variable whose value lists the chain of names for open (executing) macros.

Avoiding Interference with the Facility

Both the end-user and the developer can easily avoid interfering with the debug facility.

Do not set system options that are reserved for use by the debug facility. These are: SOURCE; SOURCE2; NOTES; MPRINT; MACROGEN; SYMBOLGEN; and MLOGIC. All toggling of these options is done by the %_DEBUG macro.

Do not assign values to any of these 12 reserved global macro variables: _BANNER; _CHECK; _CHECK_; _LEVEL; _MACNAM_; _MACROS; _NAMSTAK; _NOTES; _OPTSTAK; _SAVE; _TRACE; and _TRACE_. These variables provide communication among the user (%_DEBUG), the debug facility (%_DEBUG), and the executing macro.

Version 5 Considerations

Version 5.18 trace options do not toggle on and off as they should. The debug facility emulates the desired toggling with the SOURCE option. The results are adequate but not as clean as in Version 6.

This flaw precludes compound user (%_DEBUG) requests for a simple trace for one target group of macros and a detailed trace for a second target group.

Another problem stems from SAS procedures that write notes to the log file without beginning each line with 'NOTE:'. Version 5.18 DATASETS and GREPLAY do this when you ask to work with nonexistent members in the no-full-screen (NOFS) mode. If notes are turned off, the resulting log file contains orphan lines that bother end-users.

Function-Macro Considerations

Function macros respond differently than other macros. For function macros only the verification action works in the normal manner.

%_DEBUG cannot issue an OPTIONS statement while a function macro is executing. That means the banner and trace debug actions for function macros are controlled by the debug environment of the parent macro that invokes the function macro.

Since all banners are treated as notes (each line begins with NOTE:), notes must be turned on in the parent macro if you want function-macro banners to print on the log file.

The trace level for a function macro is the same as the trace level for the parent macro. This means a detailed trace for the parent macro automatically extends to any function macros invoked by the parent.

Fortunately, execution errors rarely occur in function macros.

Implementation Issues

Even if they don't know how to use %DEBUG, end-users benefit from the clean log file whenever they use the as-is environment of a macro system that has been written to work with the debug facility. Plus, it is easy to train users how to invoke %DEBUG to control the banner, verify, and trace actions so that they can find simple problems. Providing written documentation for the %DEBUG macro may be sufficient.

Developers need to retro-fit existing macros and code all new macros to use the debug facility. We are addressing this issue by providing a few model data sets that demonstrate how to call the %_DEBUG macro and how to include a conditional block of verification code in macro definitions. We also need to train these developers how to use %DEBUG to quickly find and correct more complex bugs. %DEBUG has proven invaluable to developers in cleaning up their own new macros.

Training developers how to use the library of verification utility macros is more difficult than teaching them to use the debug facility.

How To Find Bugs

The first step in dealing with execution problems in a macro system is locating the macro in which the problem occurs. You do this by calling %DEBUG to turn on all banners:

```
%DEBUG(level=all,banner=banner)
```

Any problem that previously was recorded on the log file will now be bracketed by banners, so you can pin down the problem macro.

Users often make syntax mistakes when they try to invoke a macro. For example, they misspell keywords, fail to separate keyword=value groups with a comma, or fail to properly quote values that contain special characters. Such mistakes produce "Error 1452: parameter not defined with macro" message. Since that macro never executes, the debug facility cannot help you with this problem.

After finding the problem macro (say its name is "problem"), you should verify its input data:

```
%DEBUG(macros=problem,banner=banner,check=
check)
```

This should tell you if there are problems with that macro's input. If there are no input problems, you look for errors in the base SAS code produced by the macro by requesting a simple trace:

```
%DEBUG(macros=problem,banner=banner,trace=
trace)
```

If this reveals no problems, look for errors in the macro code by requesting a detailed trace:

```
%DEBUG(macros=problem,banner=banner,trace=
trace,detail=detail)
```

If your output looks wrong but no errors are logged, save the work data sets:

```
%DEBUG(macros=problem,datasets=save)
```

You would then need to examine the saved work data sets, which is most readily done in Display Manager.

Example Macro Definition

The following model for a macro definition shows how a developer might use the %_DEBUG macro.

An entry call to %_DEBUG lets the %SORTER macro "check in" to the debug facility. The macro parameters BANNER and CHECK convey the as-is settings to the facility.

The global macro variable _CHECK_ has a value of 1 or 0 assigned by %_DEBUG to indicate whether verification is on or off. %SORTER has conditional code to actually check the values of its parameters. This checking is represented here by pseudo code to show that function macros are invoked to verify data. We have a complete library of these verification macros.

The body of the macro normally consists of a sequence of DATA and PROC steps. The names of intermediate work DATA members are collected in a list by appending to the local macro variable named DSLIST. DSLIST is referenced in the exit call to %_DEBUG so that the facility knows which members to delete.

The ultimate step in this model macro is PROC SORT. This step is bracketed by calls to %_DEBUG with ACTION=NOTES and NONOTES so the log file will normally capture the notes generated by PROC SORT.

The last executable statement is an exit call to %_DEBUG to close its debug environment and to restore the parent environment, which may be another executing macro or open code.

```
%macro sorter(data=,by=,out=,banner=nobanner,
              check=check);

    /* header info */

    %_debug(action=start,name=sorter,type=utility,
            support=%str(Western Production
                          Division,EUSA),
            version=%str(1.00.00(1/10/91)),
            banner=%banner,check=%check)

    %local datslist;
    %if %_check_ = 1 %then
    %do; /* validate data */
        %if %length(%data)=0 %then
            %let data = _last_;
        %else %if %validation-function-macro = 0 %then
            %goto exit;

        ... etc., for all parameters

    %end;
    %else %do; /* assign defaults when data are not
    validated */
        %if %length(%data) = 0 %then %let data =
        _last_;
        %if %length(%out) = 0 %then %let out =
        _data_;
    %end;

    data _a_;
    set %data;
    run;
    %let datslist = %datslist _a_;
```

```

%_debug(action=notes)
proc sort data=_a_ out=&out;
  by &by;
run;
%_debug(action=nonotes)

%exit:
%_debug(action=stop,banner=&banner,data=&datalist)
%mend sorter;

```

The following shows the log file after submitting:

```

%DEBUG(macros=sorter,banner=banner)
%SORTER(data=a,by=x,out=b)

```

```

NOTE: Starting execution of DEBUG macro, version
3.02.00 (1/11/91)
NOTE: supported by Western Production Division, EUSA.
NOTE: Ending execution of DEBUG macro
42 %DEBUG(MACROS=sorter,BANNER=Banner)
NOTE: Starting execution of SORTER macro, version
1.00.00 (1/10/91)
NOTE: supported by Western Production Division, EUSA.

NOTE: 4 CYLINDERS DYNAMICALLY ALLOCATED ON SYSDA FOR
EACH OF 3 SORT
WORK DATA SETS.
NOTE: DATA SET WORK.B HAS 100 OBSERVATIONS AND 1
VARIABLES.
3912 OBS/TRK.
NOTE: THE PROCEDURE SORT USED 0.38 SECONDS AND 1800K.

NOTE: Ending execution of SORTER macro
423 %SORTER(DATA=A,BY=X,OUT=B)

```

Conclusions

We successfully implemented a debug facility for SAS macro systems that yields benefits to both end-users and developers.

End-users get a concise log file that contains only what they need to know about the execution of the macro system. This is normally entry and exit banners from the master macro and notes for the ultimate PROC or DATA step.

If execution problems occur, an ambitious end-user can locate the error on his own. This saves time for the support staff.

Developers get the ability to carry out targeted debugging without editing macro definitions. Targeting minimizes the size of the log file. The ability to debug without editing lets you debug directly on the production macros and saves time.

David S. Frankel*
Exxon Company, U.S.A.
P.O. Box 5025
Thousand Oaks, California 91359-5025
(805) 494-2300

Mark A. Kochanski
Halliburton Energy Services
11255 Kirkland Way
Kirkland, Washington 98033
(206) 822-5200

® SAS is the registered trademark of SAS Institute Inc., Cary, NC, USA.

* Please address requests for user documentation and source code to Mr. Frankel.