Paper 108-26

Intelligent SAS® Log Manager

Paul D Sherman, IBM Corporation, San Jose, CA

ABSTRACT

The SAS log generated during program execution is one of the richest sources of application health, especially when dealing with macros which build and invoke code dynamically. Explicitly sending messages to the Log Manager at critical steps throughout the application one may disable all low-level SAS feedback and present only relevant, highly specialized information to the SAS log. One is led immediately to the source of both compile-time and run-time errors. Resulting is a clean, well-structured and specific trace of application flow throughout the entire job submitted.

INTRODUCTION

The Log Manager package is a lightweight addition to any SAS program eliminating the need for macro error and symbol resolution trace with no loss of SAS log content. Further, the SAS log is annotated in hierarchical C-like fashion familiar to programmers of block-structured languages, indented for each level of macro logic. General status messaging and critical exceptions are handled separately for easy identification by electronic text search utilities. All this functionality and debugging ease for the price of two simple macro calls inside each program macro.

If one follows the simple practice of sending a message to the Log Manager at the beginning and end of each macro, all responsibility of program flow trace can be encapsulated within the Log Manager itself. It is the aim of this package to totally eliminate any need for compilation status tools such as symbol or macro resolution logic, notes or errors. By providing the Log Manager with the name of the macro, its argument list and its argument's contents one knows exactly what content is being passed to which macro object at any time.

The Log Manager is composed of four basic methods (macros) being `%bloknote()`, `%setvar()`, `%putvar()` and `%throw()`. Bloknote encapsulates all internal details of message handling and SAS log interaction, thus is the only one which needs to be invoked from within external code. Two types of events lend themselves to Log Manager communication outside the usual bloknote invocation, however. These two events are healthy status and unexpected situations often called notifications and exceptions, respectively.

NOTIFICATIONS

Especially between pieces of code which take a long time to execute or those which interact with the local file system it is helpful to send a message of status to the SAS log indicating time or date, file name created or deleted, and so on. Such may be done using `%putlog()`. Since it is a member of the Log Manager, putlog knows at what indentation (code) level to report its messages, even during macro calls nested many levels deep.

```
...
%let t0 = %today();
%proc sort data=&indat.;
   by key;
   run;
%let t1 = %today();
%putlog(sorted &indat. in &t1.-&t0. sec);
...
```

EXCEPTIONS

It is considered poor programming practice not to exhaustively check all possible variations of any variable in a macro argument list. Simply put, the unconditional else in a logic branch serves such purpose, as shown in the code fragment below.

```
...
%if &var. = THIS %then %do;
   %* do this *;

%end; %else %if &var. = THAT %then %do;
   %* do that *;

%end; %else %if &var. = SOMETHING %then %do;
   %* do something else *;

%end; %else %do;
   %* dont know so just ignore input *;
   %throw(oops - unexpected input &var.);
%end;
...
```

Quite often it is not known how to handle input of wrong or undefined format or type. The `%throw()` method of Log Manager, similar to `%putlog()`, sends the desired message to the SAS log wrapped up together with the standard Base SAS `Error:` keyword. Therefore, log messages thrown during program execution encountering unknown conditions may easily be located electronically together with severe and unavoidable low-level SAS internal messages.

HOW IT WORKS

The Log Manager is a simple state machine sequenced through four states by a single general purpose method. The state variable is a numeric position counter indicating the level of code nesting at any given time during program execution. This state variable, or log level, must be defined on the global heap so that it is available in-scope throughout the entire program. Internal macro `%setvar()` manages all details of initializing and updating this state variable with the desired algebraic operation and numeric expression.

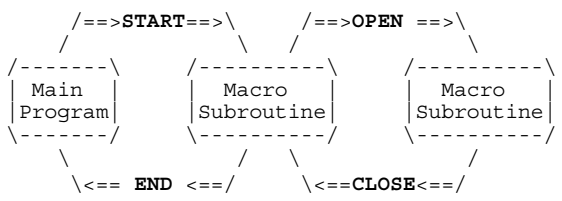

Figure 1 : The four states of `%bloknote()`

STATES

START Initializes the Log Manager, sets up its global state variable, and disables all native warnings, notes and errors stateements. Use of %putlog() or %throw() prior to entering the START state will cause many run-time errors.

OPEN Stamps the log with macro name and its argument list and values.

CLOSE Informs the log that the macro execution has finished.

END Notifies the log that all macros have finished, bringing program flow back to open-code level.

The Log Manager includes overall error checking at the two obvious state extremes: Closure when there is no open macro, and program exit when not all macros have been closed. Usually both situations are due simply to omitted `%bloknote()` calls, though any other real code runtime error may also give the same messages.

One sends a message to the LogManager by invoking bloknote indicating to which of the four code execution will proceed:

```
%bloknote(state, macroName(&arg1., ...));
```

Internal macro `%putlog()` encapsulates the Base SAS PUT function using the at (@) position option with log level state variable. This is the basic principle how log messages are indented.

EXTRACTING ARG-LIST CONTENT THROUGH VARIABLE DEREFERENCE

The chief advantage of SAS log management is not only to show which code level is executing when, but what objects and values are being exchanged. It is the latter which provides the most helpful clues during program debug without symbolgen. Simply dereference each variable in a macro's argument list when passing them to the Log Manager. In the case of keyword parameters it is also helpful to include the parameter name and equality sign (=) as well. An example follows:

```
%macro myMacro(p1, p2, kArg);
   %bloknote(OPEN, myMacro(&p1., &p2.,
                           kArg=&kArg.)
   );
    ...
   %bloknote(CLOSE);
%mend;
```

As the only exception to this rule, users of SAS software version 6.12 or earlier may need to pass only the variable name rather than its reference if such is an extremely long string, due to a 200 character variable length limitation of the macro argument list. Though loosing some generality and trace information, it is not believed to hinder greatly the program debug process.

COMPLEX EXAMPLE: ITERATOR DATA
(aka Run-time Dynamic Do-Loop)

Most interesting is the Log Manager output during repeated macro invocation with differing argument list content. This is a typical situation when one defines a data set consisting of unique keys derived from some master data set. The control or iterator data set then has one row per macro instance and is used as input to a data/_null_ step which iterates through list repeatedly invoking the desired macros or functions, often via call execute.

Suppose that there is a data set called theData which has character and numeric columns defined as key and value, respectively. Further, assume that key takes on the letters of the alphabet A through Z. Hence our sample data set theData has twenty-six possible observations, or rows. Using an sql group-by syntax, or the equivalent proc sort nodupkey and data step version, we can derive the control or iterator table which is unique on a set of columns (or keys) which we wish to pass to our action macro such as `%myMacro()` described above.

In the call execute step, any key variable derived from the iterator data set is brought out of quotes, while any un-iterated constant such as the master data set content is left quoted for dereference at a lower level of macro resolution.

```
%macro iterate(indat);
   %bloknote(OPEN, iterate(&indat.));

   proc sql noprint;
      create table iterator as
         select key,
                avg(value) as avgval
         from &indat.
         group by key
      ;
   quit;

   data _null_;
      set iterator;
      call execute(
         '%myMacro(&indat.' ||
                   ',' || key ||
                   ', kArg=' || avgval ||
         ')'
      );
      run;

   %bloknote(CLOSE);
%mend;
```

The resulting SAS log display will be highly detailed and very informative:

```
...
iterate(theData) {
   myMacro(theData, A, kArg=1419) {
   }
   myMacro(theData, B, kArg=1920) {
   }
   myMacro(theData, C, kArg=124) {
ERROR: ODBC Connection Failed.
   }
   ...
   myMacro(theData, Z, kArg=193) {
   }
   myMacro(theData, AA, kArg=193) {
      Exception ERROR: key AA not in alphabet
   }
}
...
```

Notice that something went wrong during the third pass through the iterator loop, for which SAS threw a critical error. Two things are immediately obvious: First that the action macro myMacro is generally functional and sound to which not all input conditions cause warnings, and secondly that the specific invocation sequence and values of arguments are indicated unambiguously in the SAS log. Unfortunately, critical low-level SAS messages are not catchable within runtime code and therefore cannot be indented and structured by the Log Manager. With close attention to argument value checking and exception handling, however, it is possible to keep these native SAS messages to a very minimum.

A non-SAS exception was also thrown during the last iteration of the loop, indicating the result of bad input. One immediately proceeds to examine whatever code has created data set theData, and why an observation with key value 'AA' was created. Furthermore, one knows not to expect a result, such as a graph, file or email from the last iteration, since an exception was thrown and all subsequent code in `myMacro` was bypassed.

CONCLUSION

By including only two simple macro calls, one at the beginning and end of each macro, the resulting SAS log output of any program becomes automatically annotated in a block structured C-like fashion with extremely useful information of program status and flow. The SAS internal symbolgen, merror and mlogic utilities originate from the low-level macro resolution process and are therefore transparent to code nesting level and program flow. Their messages tend to be lengthy and greatly clutter the SAS log, often requiring as much effort to decipher the resolution trace as studying specifics of the application code. Instead all macro argument list content in addition to macro invocation itself is sent to the log through use of the Log Manager, providing highly specific display of program flow and macro variable content in a clean and easy to read SAS log.

REFERENCES

Carpenter, Art, *Carpenter's Complete Guide to the SAS® Macro Language*, Cary, NC: SAS Institute Inc., 1998. 244pp.

Delwiche, Lora D. and Slaughter, Susan J., *The Little SAS® Book: A Primer*, Cary, NC: SAS Institute Inc., 1995. 228pp.

ACKNOWLEDGMENTS

I owe a special debt of gratitude to Michael Metts for his pioneering contributions and personal encouragement toward data analysis and test process engineering. I thank Donald Clark as a colleague with whose efforts, talent and experience helped introduce *The SAS System* into the wafer test process. I thank Adrienne Dong and Garth Helf for their kind and generous help, input and feedback during many helpful discussions and suggestions which have greatly improved my understanding of SAS. Finally, I am grateful to IBM Corporation for providing the tools and opportunity which made this work possible.

TRADEMARK CITATION

CONTACT INFORMATION

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

Paul D Sherman
IBM Corporation
5600 Cottle Road
San Jose CA 95193
Phone: 408-256-6091
Fax: 408-256-2700
Email: shermanp@us.ibm.com

SOURCE CODE

```
options nosource;
/* ===================================== */
/* LOGMGR - Intelligent SAS Log Manager  */
/*                                       */
/*   %setvar(name, expr, value)          */
/*      manage instance of global variable */
/*                                       */
/*   %bloknote(type, arg)                */
/*      bracket a block of code in the log */
/*                                       */
/*   %putlog(msg) - stamps log entry at  */
/*      current code level               */
/*                                       */
/*   %throw(e) - post exception to the log */
/*                                       */
/* 1.0  06/20/2000 pds   -Initial cut    */
/* 1.1  01/10/2001 pds   -mod bloknote()  */
/*                         swap annotate -  */
/*                         indent sequence  */
/* ===================================== */

/* ===================================== */
/* setvar - create/update instance of    */
/*          (global) variable            */
/*                                       */
/*          expr must either a valid     */
/*          operator + - * / or not exist  */
/*          at all (ie be blank space)   */
/* ===================================== */
%macro setvar(name, expr, value);
%* cant update macro var thru call execute();
%*   data _null_;
%*      call symput("&name.", &expr.);
%*      run;

   %if %length(&expr.) = 0 %then %do;
      %* replace contents;
      %let &name. = &value.;

   %end; %else %do;
      %* update contents using expr operator;
      %let &name. = %eval(&&&name.
                          &expr.
                          &value.);
   %end;
%mend;
```

```
/* ===================================== */
/* bloknote - bracket a block of code    */
/*            in the log                 */
/*    type may be one of four states     */
/*       START, OPEN, CLOSE or END       */
/*    requires use of global (ie static) */
/*    variable 'loglvl'                  */
/* ===================================== */
%macro bloknote(type, arg);
   %global loglvl;
   %local OPNSYM CLSDSYM MINLVL INCLVL;
   %let OPNSYM = {;   %* also <blockquote>;
   %let CLSDSYM = };  %* also </blockquote>;
   %let MINLVL = 1;   %* root code level;
   %let INCLVL = 2;   %* char steps/code lvl;

   %if &type. = START %then %do;
      data _null_;
         call symput('loglvl', &MINLVL.);
         run;
      %setvar(loglvl, , &MINLVL.);
      %putlog(&arg. &OPNSYM.);
      %setvar(loglvl, +, &INCLVL.);

   %end; %else %if &type. = OPEN %then %do;
      %* annotate ... then indent;
      %putlog(&arg. &OPNSYM.);
      %setvar(loglvl, +, &INCLVL.);

   %end; %else %if &type. = CLOSE %then %do;
      %if &loglvl = &MINLVL. %then %do;
         %throw(unopen macro - cant &type.);
      %end; %else %do;
         %* outdent ... then annotate;
         %setvar(loglvl, -, &INCLVL.);
         %putlog(&CLSDSYM.);
         %putlog();
      %end;

   %end; %else %if &type. = END %then %do;
      %if &loglvl. > &MINLVL. %then %do;
          %throw(unclosed macro-cant &type.);
      %end; %else %do;
         %putlog(&CLSDSYM.);
      %end;

   %end; %else %do;
      %* should abort sas at this point... *;
      %throw(wrong state - bloknote(&type.));
   %end;
%mend;
```

```
/* ===================================== */
/* putlog - stamps log entry at current  */
/*          code level                   */
/*    must be initialized by             */
/*    %bloknote(START)                   */
/* ===================================== */
%macro putlog(arg);
   data _null_;
      lvl = &loglvl.;
      text = "&arg.";
      put @lvl text;
      run;
%mend;

/* ===================================== */
/* throw - post exception to the log     */
/*   use also the six chars ERROR: so    */
/*   real sas errors roll up together    */
/*   must be initialized by              */
/*   %bloknote(START)                    */
/* ===================================== */
%macro throw(e);
   %putlog(Exception ERROR: &e.);
%mend;

/* ===================================== */
/* aMacro - typical macro template       */
/* ===================================== */
%macro aMacro(args);
   %* set your macro name here;
   %bloknote(OPEN, aMacro(&args.));

   %* place your macro code here *;

   %bloknote(CLOSE);
%mend;

/* ===================================== */
/* main - typical program entry point    */
/* ===================================== */
%macro main(args);
   dm 'log' clear;
   dm 'program editor' clear;

   %bloknote(START, main(&args.));

   %* place your main code here *;

   %bloknote(END);
%mend;

/* ----- program execution starts here ----*/
options nonotes noovp nomerror nomprint
        nomlogic nosymbolgen errors=max
        msymtab=MAX mvarsize=MAX
        mautosource mrecall
        sasautos=(sasautos)
        linesize=256 pagesize=1000
        yearcutoff=1920 cleanup dsnferr
        nocenter nodate
        serror nostimer
;
main();
```