Paper 077-2018

Stop the Madness!

Detecting Runtime Errors and Exiting from Nested Macros Gracefully

Ted D. Williams, PharmD, BCPS, Magellan Method

ABSTRACT

Few things are more frustrating that kicking off a long running SAS program, only to return hours later and discover the program encountered an error early in processing. One of those even-more-frustrating things is digging through the logs to determine the state of the system upon failure. Which of the many of calls to %VeryHelpfulMacro() failed? What were the parameters and state of temporary tables at that time? This paper describes one approach to monitoring execution of macros, detecting errors, and creating a call stack to determine where and when runtime errors occurred. Adopting these simple techniques can reduce debugging time, promote reuse, and reduce project develop time.

INTRODUCTION

SAS has the distinctive features of allowing programs to continue executing regardless of runtime errors. Although there are advantages to this flexibility, it can cause and sometimes obfuscate unexpected results. This is particularly true when a macro is called multiple times and creates global variables or working datasets. With small programs, it is simple enough to check the error log, correct the issue, and continue. For programs with thousands or tens of thousands of lines of code, all reusing the same core macros, debugging by log review can be a tedious and daunting task. SAS has a variety of error codes and messages with various strengths and limitations.[1,2] Monitoring the automatic macro variable SYSCC provides simple and consistent mechanism for detecting runtime errors. Once detected, the system can be halted and the error reported. Halting execution in this manner preserves the system state at the time of the error greatly simplifying debugging. This paper will demonstrate how a handful of lines of code and a few simple macros can save developers hours of waiting and developing time. These techniques are straightforward enough for the novice SAS programmer with a basic understanding of macros and powerful enough to interest even the most experienced SAS developers.

DAMN THE ERRORS, FULL SPEED AHEAD

It does not take long to discover that SAS executes statements regardless of previous errors. A simple illustration appears in the following code and results:

```
%Macro BrokenMacroNoErrorTrap();
    %put Things seem fine at the start ;
    PROC I_Made_A_Typo;
    RUN;
    %put Things seem fine at the end ;
%mend;

%Macro HelpfulMacroWithoutErrorTrap();
    %put Things seem fine at the start ;
    *Do some stuff here dependent upon the broken Macro;
    %put Things seem fine at the end ;
%mend;

%Macro CallingMacroNoErrorTrap();
    %put ****************************************;
    %put ****Ex. 1 no error trap****************;
    %put ****************************************;
    %put Things seem fine at the start ;
    *Call a macro which causes an error;
    %BrokenMacroNoErrorTrap();
    *Will This macro execute?;
    %HelpfulMacroWithoutErrorTrap();
    *What does the error status look like?;
    %put Things seem fine at the end;
%mend;

%CallingMacroNoErrorTrap();
```

In the unfortunate example above, neither CallingMacroNoErrorTrap() nor HelpfulMacroWithoutErrorTrap() know that a runtime error occurred. Depending upon the function and results of the system, the program may produce results which appear correct, but could be erroneous or incomplete.

```
***************************************
****Ex. 1 no error trap***************
***************************************
Things seem fine at the start
Things seem fine at the start
ERROR: Procedure I_MADE_A_TYPO not found.
Things seem fine at the end
Things seem fine at the start
Things seem fine at the end
Things seem fine at the end
```

The goal for every programmer is robust, easily debugged code. But how much time will it take to handle errors? Surprisingly little effort and code are required.

YES DEAR, I AM LISTENING – DETECTING ERRORS

SAS has several built in macro variable which make detecting and managing error handling surprisingly simple. Possibly the most helpful is SYSCC.[3] SYSCC is both a readable and writable variable. That is, when SAS encounters an error, SAS will set the value for the programmer to read the value. Programmers can also set and reset the value. If there are no errors, SYSCC=0. Values 1-4 are warnings and values above 4 are runtime errors. Detecting errors can be as simple as adding a %IF ...%THEN … %GOTO statement:

```
%Macro CallingMacroWithErrorTrap();
    %put ***************************************;
    %put ****Ex 2. Simple Reactive Error Trap***;
    %put ***************************************;
    %put Things seem fine at the start while SYSCC=*&SYSCC*;
    *Call a macro which causes an error;
    %BrokenMacroNoErrorTrap();
    *After each step, Check for an error;
    %If &SysCC>4 %Then %GoTo ErrorTrap;
    *Will This macro execute?;
    %HelpfulMacroWithoutErrorTrap();
    *After each step, Check for an error;
    %If &SysCC>4 %Then %GoTo ErrorTrap;
    *What does the error status look like after executing macro?;
    %put Things seem fine at the end  while SYSCC=*&SYSCC*;
%Return;
%ErrorTrap:
%put Things went wrong SYSCC=*&SYSCC*;
%put and I found it and stopped execution;
%put by adding a single line of code after each macro call;
%mend;

%CallingMacroWithErrorTrap();
```

In the example above, after each macro is called, the calling macro checks the state of SYSCC. If there is an error message (SYSCC>4), the program flow is redirected by the %GOTO statement to the label %ERRORTRAP. The custom error message is then printed to the log with %PUT statements and execution stops.

```
***************************************
****Ex 2. Simple Reactive Error Trap***
***************************************
Things seem fine at the start while SYSCC=*0*
Things seem fine at the start
ERROR: Procedure I_MADE_A_TYPO not found.
Things seem fine at the end
Things seem fine at the start
Things seem fine at the end
Things went wrong  SYSCC=*3000*
and I found it and stopped execution
by adding a single line of code after each macro call
```

There are advantages and disadvantages of this strategy. Most obviously, when execution stops at the first error, there is no longer a need to scroll through the log checking for errors. The error is always at the end of the log. This approach also requires very little of the program to be changed to detect errors, stop execution, and report the findings. Simply cut and paste the %IF…%THEN…%GOTO… statement after each macro call. The error handler is also very simple and easy to code. The limitations are that there is no reporting of where in the program flow the error occurred. In this simple example it is fairly obvious. But with larger programs and nested macro calls, this strategy leaves much to be desired. Fortunately, taking error handling to the next level is surprisingly easy.

RECYCLE AND REUSE – CREATING AN ERROR HANDLING MACRO

Wrapping error handling into a dedicated macro make consistent error handling a snap. A reusable macro turns error handling into 3 lines of code at the beginning and end of the macro. What could be easier?

There are two parameters that the error handler will need: where the error occur and what the error was. Where the error occurred can be answered using the SYSMACRONAME automatic macro variable.[4] This is a fantastic time saver, but it does have a catch. If the automatic variable is pass directly to another macro, the called (not the calling) macro name is used. So the contents of the SYSMACRONAME variable must be saved to a local macro variable. In the example below, the CalledBy local macro variable is used. The error message is also passed to the error handler. Although there are several automatic macro variables the handler could use, these all have limitations.[2] An error message parameters provides greater flexibility, which will be demonstrated a bit later. For now, the following code show how 3 lines of code at the beginning and 3 lines of code at the end of the macro create the framework for error trapping system.

```
%Macro BetterErrorHandling();
   %put **************************************;
   %put ****Example # 3 Better error handler***;
   %put **************************************;
   ***************************
   Standard Error Handling Definitions
   ***************************;
   %local Message;
   %Local CalledBy;
   %let CalledBy=&SYSMacroName;

   *before doing anything, check for any errors;
   %If &SysCC>=&GlobalErrorDetectionLevel  %Then %goto ErrorTrap;
   ***************************
   Local Macro Definitions
   ***************************;
   %put Things seem fine at the start of &CalledBy while SYSCC=*&SYSCC*;
   %BrokenMacro();
   %If &SysCC>=&GlobalErrorDetectionLevel %Then %GoTo ErrorTrap;
   %put Things seem fine at the end  of &CalledBy  while SYSCC=*&SYSCC*;

%Return;
%ErrorTrap:
%SimpleErrorHandler(CallingMacroName=&CalledBy,ErrorMessage=&Message);

%mend;

%Macro BrokenMacro();
   ***************************
   Standard Error Handling Definitions
   ***************************;
   %local Message;
   %Local CalledBy;
   %let CalledBy=&SYSMacroName;
   %If &SysCC>=&GlobalErrorDetectionLevel  %Then %goto ErrorTrap;
   ***************************
   Local Macro Definitions
   ***************************;
   %put Things seem fine at the start of &CalledBy while SYSCC=*&SYSCC*;
   PROC SQL NOPRINT;
         SELECT * FROM MyDataSetIsMissing;
   QUIT;
   %If &SysCC>=&GlobalErrorDetectionLevel %Then %GoTo ErrorTrap;
   %put Things seem fine at the end of &CalledBy while SYSCC=*&SYSCC*;
%Return;
%ErrorTrap:
%SimpleErrorHandler(CallingMacroName=&CalledBy,ErrorMessage=&Message);
%mend;

%Macro SimpleErrorHandler(CallingMacroName=,ErrorMessage=);
   %Put &CallingMacroName encountered an error &ErrorMessage;
%mend;

%Let SYSCC = 0;
%global GlobalErrorDetectionLevel;
```

```
  %Let GlobalErrorDetectionLevel=5;
  %BetterErrorHandling();


  ;
```

```
*************************************
****Example # 3 Better error handler***
*************************************
Things seem fine at the start of BETTERERRORHANDLING while SYSCC=*0*
Things seem fine at the start of BROKENMACRO while SYSCC=*0*
ERROR: File WORK.MYDATASETISMISSING.DATA does not exist.
BROKENMACRO encountered an error
BETTERERRORHANDLING encountered an error
7488
```

It is important to check for errors after each PROC. Although this is a little cumbersome, the call is a 1 line cut-and-paste (see bolded text above).

The hard coded check of SYSCC>4 has been replaced by the global macro variable GlobalErrorDetectionLevel. Although global variable can cause problems, this is really a global constant. Although it is a bit longer, it is also easier to read and if there are ever changes to the SYSCC variable (or sensitivity to errors), it will be much easier to update than searching for the number 4 across all programs and macro libraries.

Note how the log contains messages from both BrokenMacro and BetterErrorHandling macros. This looks a lot like a call stack.[5] A call stack is indispensable for debugging highly recursive programs, i.e. programs with macros calling macros calling macros. SAS does not have a built in call stack, but one can easily be built into the error handler, as is demonstrated in the next example.

Prior to executing any macros, the SYSCC variable must be set to zero. SYSCC is not reset when a program is run multiple time, so it must be manually reset prior to executing the code. Otherwise, the program will detect residual errors, falsely reporting them as failed execution.

THIS ONE GOES TO 11 – GETTING FANCY WITH ERROR LOGS

Now it is time to put all of these pieces together into a usable and robust log and call stack. The log will consist of three variable, CalledBy, Message, and WasError. The first two are familiar from the previous example. Adding the WasError flag will allow the log to perform double duty. It will track both progress and errors, so the entire program flow will be tracked, further facilitating debugging. The error handler will also check if there are pre-existing errors in the log. If there are, the error message will be replaced with the “CALL STACK” message.

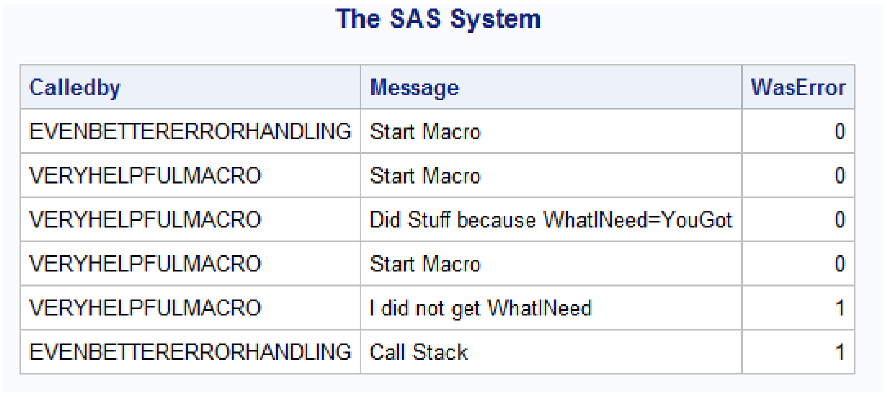
These modifications to the error handler provide a number a new opportunities. With the addition of the call stack feature, adding an error check at the beginning of each macro prevents them from continuing when there has already be a fault in the system. The new macro AddLog can be used to track the starting and/or ending of a macro, as well as report on the results of the macro such as number of observations found. It is now possible for macros to proactively check the state of parameters and exit gracefully when the system is in an unexpected state, such as:

```
      %If %Length(&WhatINeed)=0 %Then
```

The code segment below puts all of these pieces together, and displays the contents of the log.

```
%macro ResetLog();
    proc sql;
          create Table ExecutionLog
          (
                Calledby CHAR(100),
                Message CHAR(1000),
                WasError numeric
          )
          ;
    quit;
%mend;
%macro AddLog(CallingMacroName=,Message=,WasError=0);
    %If %SYSFUNC(EXIST(ExecutionLog))=0 %Then
          %ResetLog();
    proc sql;
          insert into Executionlog
          VALUES ("&CallingMacroName","&Message",&WasError)
          ;
    quit;
%mend;



%Macro FancyErrorHandler(CallingMacroName=,ErrorMessage=);
    %Put &CallingMacroName encountered an error &ErrorMessage;
    *First, make sure a log dataset exists;
    %If %SYSFUNC(EXIST(ExecutionLog))=0 %Then %ResetLog();
    *Check for prior errors, If found, no need to repeat the message text ;
    Proc sql NOPRINT OUTOBS = 1;
          SELECT WasError FROM ExecutionLog WHERE WasError=1;
    quit;
    %IF &SqlObs ne 0 %then
    %do;
          %let ErrorMessage = Call Stack;
    %end;
    *If the ErrorMessage is blank, use the last system error message;
    %IF %Length(&ErrorMessage)=0 %then
    %do;
          *Use superq to handle control characters in system error message;
          %let ErrorMessage = *%superq(syserrortext)*;
    %end;
    *Log the error;
    %AddLog(CallingMacroName=&CallingMacroName,Message=&ErrorMessage,WasErro
r=1);
%mend;

%Macro EvenBetterErrorHandling();
    %put ****************************************;
    %put ****Example # 4 Even Better error handler;
    %put ****************************************;
    ****************************
    Standard Error Handling Definitions
    ****************************;
    %local Message;
    %Local CalledBy;
    %let CalledBy=&SYSMacroName;
    *before doing anything, check for any errors;
    %If &SysCC>=&GlobalErrorDetectionLevel %Then %goto ErrorTrap;
    ****************************
    Local Macro Definitions
    ****************************;
    %AddLog(CallingMacroName=&CalledBy, Message=Start Macro);

    %VeryHelpfulMacro(WhatINeed=YouGot);
    %If &SysCC>=&GlobalErrorDetectionLevel %Then %GoTo ErrorTrap;

    %VeryHelpfulMacro(WhatINeed=);
    %If &SysCC>=&GlobalErrorDetectionLevel %Then %GoTo ErrorTrap;

    %AddLog(CallingMacroName=&CalledBy, Message=End Macro);



%Return;
%ErrorTrap:
%FancyErrorHandler(CallingMacroName=&CalledBy,ErrorMessage=&Message);

%mend;

%Macro VeryHelpfulMacro(WhatINeed=);
    ****************************
    Standard Error Handling Definitions
    ****************************;
    %local Message;
    %Local CalledBy;
    %let CalledBy=&SYSMacroName;
    %If &SysCC>=&GlobalErrorDetectionLevel  %Then %goto ErrorTrap;
    ****************************
    Local Macro Definitions
    ****************************;
    %AddLog(CallingMacroName=&CalledBy, Message=Start Macro);

    %If %Length(&WhatINeed)=0 %Then
    %do;
          *Tell SAS there was an error;
          %Let SYSCC = &GlobalErrorDetectionLevel;
          %let Message = I did not get WhatINeed;
          %GoTo ErrorTrap;
    %end;

    *DO STUFF;

    %If &SysCC>=&GlobalErrorDetectionLevel %Then %GoTo ErrorTrap;
    %AddLog(CallingMacroName=&CalledBy, Message=Did Stuff because
WhatINeed=YouGot);

%Return;
%ErrorTrap:
%FancyErrorHandler(CallingMacroName=&CalledBy,ErrorMessage=&Message);
%mend;
%Let SYSCC = 0;
```

```
%global GlobalErrorDetectionLevel;
%Let GlobalErrorDetectionLevel=5;
%ResetLog();
%EvenBetterErrorHandling();
proc sql outobs = 100;
    select * from ExecutionLog;
quit;
```



The SAS System

Calledby	Message	WasError
EVENBETTERERRORHANDLING	Start Macro	0
VERYHELPFULMACRO	Start Macro	0
VERYHELPFULMACRO	Did Stuff because WhatINeed=YouGot	0
VERYHELPFULMACRO	Start Macro	0
VERYHELPFULMACRO	I did not get WhatINeed	1
EVENBETTERERRORHANDLING	Call Stack	1

Note that the error handler macro interface has not changed. This demonstrates how easily the error handler can be modified and enhanced without requiring changes to the calling macros. Any changes to the error handler are encapsulated within the macro.

The macros ResetLog, AddLog, and FancyErrorHandler and the global variable GlobalErrorDetectionLevel can now be saved into a separate program and referenced using the %INCLUDE statement for reuse in every program.[6] Now 3 lines of code can be integrated into any existing macro (%INCLUDE, %LET SYSCC=0, %ResetLog()) to enable robust error handing.

CONCLUSION

Integrating a robust and low-overhead error handling system into new and existing SAS programs is simple and requires minimal recoding. Just 3 lines of code at the beginning of each macro, 3 lines of code at the end of the macro, and 3 lines of code at the beginning of each program is all it takes. The built-in macro variable SYSCC can be used to check for errors after each proc. When errors are detected, macros can call a single error hander which encapsulates logging of program flow, errors, and gracefully halts program execution, even across multiple macros.

REFERENCES

1. Billings, Thomas E. *Strategies for Error Handling and Program Control: Concepts*.; 2015. https://support.sas.com/resources/papers/proceedings15/1565-2015.pdf. Accessed May 1, 2018.

2. Hughes, Troy Martin. *Why Aren't Exception Handling Routines Routine? Toward Reliably Robust Code through Increased Quality Standards in Base SAS®*.; 2015. https://support.sas.com/resources/papers/proceedings15/3387-2015.pdf. Accessed May 1, 2018.

3. SYSCC Automatic Macro Variable :: SAS(R) 9.3 Macro Language: Reference. http://support.sas.com/documentation/cdl/en/mcrolref/62978/HTML/default/viewer.htm#p11nt7mv7k9hl4n1x9zwkralgq1b.htm. Accessed May 1, 2018.

4. Automatic Macro Variables: SYSMACRONAME Automatic Macro Variable. http://support.sas.com/documentation/cdl/en/mcrolref/61885/HTML/default/viewer.htm#a001659612.

htm. Accessed July 10, 2018.

5. SAS® Help Center: Construct a Stack. http://documentation.sas.com/?docsetId=imlug&docsetTarget=imlug_lists_examples02.htm&docsetVersion=14.2&locale=en. Accessed July 10, 2018.

6. Statements: %INCLUDE Statement - 9.2. http://support.sas.com/documentation/cdl/en/lrdict/64316/HTML/default/viewer.htm#a000214504.htm. Accessed July 10, 2018.

ACKNOWLEDGMENTS

RECOMMENDED READING

CONTACT INFORMATION

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

Ted D. Williams
Magellan Method
88 Silva Lane, Tech 4, Middletown, RI 02842
314-387-4305
tdwilliams1@magellanhealth.com