Paper 1565-2015

Strategies for Error Handling and Program Control: Concepts

Thomas E. Billings, MUFG Union Bank, N.A., San Francisco, California

ABSTRACT

SAS® provides a complex ecosystem with multiple tools and products that run in a variety of environments and modes. The SAS system provides numerous error-handling and program control statements, options, and features. These features can function differently according to the product/run environment, and many of them have constraints and limitations. Here we review a number of potential error handling and program control strategies that can be employed, along with some of the inherent limitations of each. The bottom line is that there is no single strategy that will work in all environments, all products, and all run modes. Instead, programmers need to consider the underlying program requirements and choose the optimal strategy for their situation.

BACKGROUND: THE COMPLEX SAS ECOSYSTEM

SAS® program code can be user-written, system-generated (e.g., the code generated by SAS® Enterprise Guide® Tasks) or a combination of both types. SAS programs can be run in any of a number of different modes and environments, including:

- Interactively in a windowing system like SAS for Windows (aka "PC SAS") and/or using X-Windows or similar interface on other systems,
- In *interactive line mode*, a dated mode originally designed to work on and subject to the constraints of dumb terminals and/or (old-fashioned) terminals that printed everything on paper (note: this mode is still potentially useful for short, simple jobs).
- Interactively using SAS Studio (which runs in a client-server environment), SAS Enterprise Guide and possibly other products
- User-written code and/or system-generated code (code *exported* from SAS Enterprise Guide or *deployed* from a metadata-based SAS product like, e.g., SAS® Data Integration Studio®) that is run in batch, i.e., via the "sas" command on your operating system
- Program code can be run selectively and manually in SAS Enterprise Guide and metadata-based SAS products (e.g., SAS Data Integration Studio Transformations and Jobs)
- Program code can be executed via a stored process; these can be scheduled.

Many sites provide parallel or nearly parallel versions of certain environments, the familiar **dev** (development), **test**, **prod** (production), and **DR** (disaster recovery) environments. These are usually provided for batch processing but may also be supported for some of the metadata-based SAS products.

Programs are frequently developed outside the batch environment, say using one of: SAS Enterprise Guide, SAS for Windows, or SAS Studio, and then ported to batch for production. Initially the code would

be run in dev, then after sufficient testing, promoted through test and into production. If an error or problem arises in a batch production system, the code may be debugged and fixed in dev or test, using the same (non-batch) tools used for program development.

The relevant point here is that, over their lifecycle, SAS programs may be developed, tested, and run in multiple, different environments and using different products. It so happens that many of the SAS statements, options, and features commonly used for error handling and program control have different functionality based on the product/environment they are running in. Some of these features also have constraints that limit their use or reduce their effectiveness. Needless to say, this complicates program control and error handling for programs that need to run safely in multiple environments.

This paper presents an overview of select error handling and program control features, and describes some strategies to consider when developing programs and systems. There is no single best strategy that works in every SAS tool and environment. Accordingly, the intent of this paper is to encourage discussion and research on the topics of program control and error handling in SAS programs and systems. We begin with a (very) brief overview of program control features in the next section.

PROGRAM CONTROL

Our primary interest in program control here is vis-à-vis error handling. In that context, there are a few common strategies for program control:

1. Terminate immediately on error
2. Terminate gracefully on error
3. Detect errors and use macros to control what runs after the error
4. Rely on OPTIONS SYNTAXCHECK or DMSSYNCHK.

There is also the “run everything regardless of errors” approach which can be a high risk situation; unfortunately, that is how SAS Enterprise Guide works. Our discussion of program control is integrated with the review of error handling options, in the following section.

ERROR HANDLING

Error handling in large SAS-based systems can be very complex, as it reflects the interaction between multiple, different factors. The Base SAS system provides a number of native features relevant to error handling:

- Numerous system OPTIONS for error handling; some of these are specialized, others more general.
- Syntax check mode
- Support for checkpoints
- Error codes and messages are available in macro variables and via functions.
- DATA step: ABORT, ERROR statements
- Macro statements: %ABORT
- PROC OPTIONS to capture/restore option settings (also function GETOPTION)
- SAS 9.4+: PROC PRESENV to preserve global statements and environment variables for later restart.

Some of the features listed above have different functionality depending on the product/run environment; see Billings (2014) for examples and discussion.

If you are using a metadata-based SAS product, it may have features for/related to error handling:

- SAS Enterprise Guide – basically no error handling,
- SAS Data Integration Studio – this product provides a significant number of error handling options, including termination-on-error and checkpoints

The architecture of the program/system is also relevant:

- Is it designed to support reruns of a standard production run, i.e., you can run the program multiple times with no duplication or corruption of data?
- Can it be run safely in all environments you might use, or are additional/special precautions needed when running in certain environments?
- Are critical elements of the system (e.g., permanent/history files) afforded a very high degree of protection?

Finally, error handling does not exist in a vacuum: how important the system is to the enterprise, is very relevant. Systems that are critical to the enterprise should have substantial error handling; a low priority/limited use system can be run with less protection.

Challenges:

There are many options and methods to choose from when deciding how to implement error handling in a large SAS-based system, and many challenges to overcome. Some of the challenges here are:

- Most SAS error handling options/features have inherent limitations or constraints. For example, macro variables that contain the error condition codes get reset: &SYSERR is reset at every step boundary which is inconvenient; &SQLRC gets reset at the start of every invocation of PROC SQL.
- SAS error-handling options can function differently depending on the run environment. OPTIONS ERRORABEND may be a good choice for batch, but if an error occurs when running a job manually in a metadata-based SAS product, you may be kicked out of the system with no error messages and no log. (In SAS Enterprise Guide, the server may disconnect, then reconnect as a new process and run downstream tasks in normal mode; this is high-risk and can potentially cause loss of data.)
- Default values of error options may vary by environment. Syntax check mode is turned off in SAS Enterprise Guide before each task as it is inconvenient for development work (note that you can specify the option in Program Tasks and tasks that let you insert code). It is a good idea to explicitly set the options needed, to be sure of coverage.
- Inserting user-written error checking code into the code generated by some metadata-based SAS products can be cumbersome and complex.

POSSIBLE STRATEGIES

Multiple strategies are feasible, and the developer needs to choose the methods that are optimal for the target system, relevant environment(s), products, and overall system requirements, including operational requirements. A number of strategies are described below; note that hybrid approaches – combinations of multiple strategies below – are possible.

#1: Rely primarily on syntax check mode: OPTIONS SYNTAXCHECK DMSSYNCHK;

When the syntax check mode options are selected and errors occur in a DATA or PROC step that creates a data set, the system enters a special scan mode where it creates only header records (OPTIONS

OBS=0) and does not replace permanent data files (OPTIONS NOREPLACE). The special scan mode is low risk but not zero risk in batch or windowing environments.

However, in SAS Enterprise Guide the behavior of this feature is very different from its behavior in batch. In SAS Enterprise Guide, the syntax check option and special scan mode are turned **off** at the beginning of **every** task, and downstream tasks will execute normally after errors. (The option can be specified in user-written code in Program Tasks and via insertion of pre-code.) Syntax check is best suited to development work; in general, you should be reluctant to rely solely on syntax check in production -- see Billings (2013, 2014) for more details.

#2: Fail ASAP (as-soon-as-possible), method 1:
Rely primarily on OPTIONS ERRORABEND;

OPTIONS ERRORABEND – when used in conjunction with a number of other SAS error options - will terminate your batch program **immediately** when most errors occur. This is not a graceful shutdown, i.e., you cannot send an error email from the program (after the error occurs) and instead will have to do that in a separate, follow-up program or in the script that runs the job. This option can be very useful in batch production programs that implement checkpoints or the functional equivalent thereof.

OPTIONS ERRORABEND may terminate some SAS metadata-based products that are running programs in manual mode. Your program ends, the tool closes, and you might not get an error message or a log. Needless to say, it is difficult to debug with no or an incomplete log. If the option is specified in SAS Enterprise Guide and an error occurs in a task, that task will be terminated, the server may disconnect and reconnect, after which downstream tasks execute in a different session (lacking the macro variables of the original session), in normal mode. Clearly, ERRORABEND is best suited for runs in batch mode.

#2: Fail ASAP, method 2:
Use the terminate-on-error features of your SAS metadata-based product

Your SAS metadata-based product may have terminate-on-error options. The SAS Data Integration Studio has this feature, along with related options such as send email on error. Termination of jobs in manual runs can have the effect described above, so this strategy is more relevant to running deployed source code in batch.

#3: Architecture: programs that can be easily rerun after errors

Some programs can be written so they can be rerun multiple times (for the same target run), as-is or with only minor changes without corrupting permanent data sets or files – and/or that fix the files when a successful run is achieved. When errors do occur, the program would be fixed and/or relevant files would be restored from backup and the entire program rerun for the target date or data range. Some examples: report programs that simply overwrite the previous report versions – if it exists, or e.g., a program that adds 1 month of data to a file can first remove the target month's data – if any – already present in the file (such data may have been added in previous runs, and the current run is actually a rerun to fix a previous error). This method is most relevant for small to medium size programs.

This strategy should not be used for huge monolithic programs that take a long time to run and/or consume large amounts of system resources, as they are expensive to rerun in their entirety. Note that large monolithic programs can often be divided into smaller individual segments that run as separate programs, and some (or even all) of those segments may be rerunable.

This strategy is feasible if it is acceptable for your production files or reports to possibly contain bad data for the latest run period, for a presumably short period until the program can be fixed and rerun. This situation is more common than may be recognized, as it can occur in normal operations as the result of errors driven by new data values or other new conditions in the input sources.

#4: *Fail-safe architecture:* capture error codes, terminate gracefully, and protect permanent data sets

Consider a large program with extensive ETL: extract-transform-load processing. Divide the program into smaller segments, each with the sequential logical structure:

1. Extract raw or intermediate data into work files
2. Transform the data and hold the updated data in work files
3. Insert or load the updated data into permanent data sets.

Now if an error occurs in steps 1, 2 above it is not a problem, so long as corrupt data are not loaded into the permanent data sets. In this approach, a macro is invoked after each DATA or PROC step to test the value of system error codes, via the macro variables &SYSERR and &SQLRC. If an error condition exists, an error flag macro variable is set.

The values of &SYSERR and/or &SQLRC are step-specific and are reset when the next step boundary (or PROC SQL invocation) is encountered, so it is necessary to run a macro after each step to capture and save the error condition codes. The SAS Data Integration Studio has a similar process where it captures the value of &SYSERR and/or &SQLRC via the **%rcset** macro (see link in references for more information).

Besides testing/capturing the values of &SYSERR and &SQLRC, a macro invoked after each DATA or PROC step can also take other actions when errors occur – e.g., update an error data base, send email, and/or gracefully terminate a program. That approach is described in a NESUG paper, Gaudana & Sezek (2005).

Returning to steps 1-3 above, if the code is organized so that the processing for steps 1 & 2 precedes the step 3 code, then one can test the (cumulative) macro error flags set when the step 1, 2 code was run, to determine if an error has occurred during processing. If indeed an error has occurred in steps 1, 2 then the update code of step 3 is **not** executed. More precisely, the code is under macro control and is not even scanned/compiled as might occur in, say, syntax check mode. This prevents updates from occurring when an error condition exists, providing a degree of protection to permanent data sets.

Properly implemented, this strategy will work in batch, in single-session manual runs in some SAS metadata-based products, and in SAS Enterprise Guide so long as the program does not use OPTIONS ERRORABEND, ENDSAS, ABORT/%ABORT ABEND/RETURN as these can cause server disconnect & reconnect, with loss of global macro variables; see Billings (2014) for more information.

Syntax check mode should be specified in this approach for extra protection, though it can be disabled during development work. (There are constraints on syntax check mode: if you enter the special scan mode it can be turned off, but you can't re-specify the option as if you do, the system will immediately reenter the special scan mode; see Billings (2013) for details on this point.)

SAS Data Integration Studio. This approach can be modified to use source code generated by some SAS metadata-based products, for example SAS Data Integration Studio. There you have to write wrapper code and test the value of &job_rc, which is the maximum (over a job or most of the steps in a job) value of &SYSERR and other macro error variables tested in the generated code. The test on &job_rc should be done right after the end of a job, so you need to test after each job when working with a large set of SAS Data Integration Studio jobs.

SAS Enterprise Guide. In SAS Enterprise Guide this approach can work well with user-written code in Program Task windows, where the error-check macro invocations can be included as the code is written. (As mentioned above, for this approach to be effective, the programs should not use any of the features that can cause server disconnect/reconnect in SAS Enterprise Guide.) For code generated by Tasks, the strategy is less optimal. SAS Enterprise Guide can insert user-written pre- and post-code for all tasks, but that code is fixed and not necessarily tailored to the task.

The SAS Enterprise Guide Task that may be the most frequently used – the Query Builder task - won't let you insert a (customized for the specific task) macro invocation after the system-generated code. Instead, you have to put it in a Program Task after the Query Builder (or, if possible, insert before PROC/DATA step invocation in next task) and the end result may be a cluttered/scattered project that is more complex than desirable and difficult to maintain.

Macro to capture error codes. A sample macro that implements this approach is provided in Appendix 1. The macro does not check &SYSCC (which can be set by users) or &SYSLIBRC, but it could be easily modified to do so. (If there is an error in a LIBNAME statement, it usually causes errors downstream when the libref is used.)

The last part of this strategy is implemented when you use/test the macro error flags created by the macro that captured the values of &SYSERR and &SQLRC. The source code to write or update the permanent data sets (or create final reports) is encapsulated in a macro; the derived data are **not** written out/updated IF an error has occurred upstream.

ADDITIONAL CONSIDERATIONS

Avoid large monolithic jobs. These are usually slow to run, and that can increase the time required for debug. Instead, if/when possible split the job into multiple smaller programs, providing suitable logical layering. Encapsulating jobs in Program Tasks in SAS Enterprise Guide can be advantageous for development as it surfaces the intermediate files on a project page, where SAS Enterprise Guide Tasks can be used for rapid querying and analysis when errors occur. However, programmers working in SAS Enterprise Guide should be aware of the limitations on error handling features that are inherent to the platform; see Billings (2014) for more information.

The programs should not run at all if the required input files don't exist or the files exist but have no (or insufficient) data for the target analysis. Code can be written to check that the target input files exist and that the expected data are actually present. (Depending on the application, additional checks on the input data quality/quantity may be appropriate before the program runs.)

Avoid invalid dates/date ranges. If the programs are operating for a given date or date range, it is imperative that the dates used for a run must be legal, i.e., within valid ranges. If illegal dates are specified, or no dates specified at all, the options are to terminate with an error (i.e., don't run anything) or default to some standard processing, e.g., run the production for the most recent complete calendar month (or week). Running with an invalid date range in a program that is designed to process a timespan is high risk in that it may cause loss of data in some circumstances.

Backup should be "intelligent" to the extent possible. This means you backup once for a scheduled run and don't repeat the backup if you have to rerun. In particular, you don't want to run a redundant backup if that might overwrite a good backup with damaged or incomplete files from a run that experienced errors.

Have at least some quality assurance (QA) checks. Ideally, on successful completion of a run, a QA report is generated on the new data/files and emailed to relevant staff or saved in a known location. The content will vary based on the target system, but might contain row counts for important tables and PROC FREQ runs for a few select, important variables.

EPILOGUE

Strategy #4 above covers the widest range of environments and run modes, but it does not work everywhere. There is no single error handling/program control strategy that will work in all tools, products, environments, and run modes. Instead, choose a strategy based on the program requirements and the environments, tools, and run modes relevant to your situation.

APPENDIX 1:
SAMPLE MACRO TO CAPTURE &SYSERR AND &SQLRC

```
* This program code is released under a Berkeley Systems Distribution
  BSD-2-Clause license, an open-source license which permits free
  reuse and republication under conditions;

/*
Copyright (c) 2014, Thomas E. Billings
All rights reserved.

Redistribution and use in source and binary forms, with or without
modification, are permitted provided that the following conditions are met:

1. Redistributions of source code must retain the above copyright notice,
this list of conditions and the following disclaimer.

2. Redistributions in binary form must reproduce the above copyright notice,
this list of conditions and the following disclaimer in the documentation
and/or other materials provided with the distribution.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS"
AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE
ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT HOLDER OR CONTRIBUTORS BE
LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR
CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF
SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS
INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN
CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE)
ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE
POSSIBILITY OF SUCH DAMAGE.
*/

%global sqlrc save_syntax save_syserr save_sqlrc;

%* macro variables to flag/retain error indicators;
%let sqlrc=0;
%let save_syntax=0;
%let save_syserr=0;
%let save_sqlrc=0;

*************************************************************************;
* macro to capture error conditions from system-set macro variables.    ;
* also check if in special syntax check scan mode                       ;
*************************************************************************;
```

```
%macro save_err_flags;
      %if ((%nrstr(OBS=0) = %nrquote(%sysfunc(getoption(obs,keyword)))) AND
            (NOREPLACE = %sysfunc(getoption(replace,keyword)))) %then %do;
            %let save_syntax=1;
          %put save_err_flags: save_syntax=&save_syntax;
            %end;

      %if ((&SYSERR ne 0) and (&SYSERR ne 4)) %then %do;
            %let save_syserr=&SYSERR;
          %put save_err_flags: save_syserr=&save_syserr;
        %end;

      %if ((&SQLRC ne 0) and (&SQLRC ne 4)) %then %do;
            %let save_sqlrc=&SQLRC;
          %put save_err_flags: save_sqlrc=&save_sqlrc;
            %end;

      %* more elegant macro IN operator is available but works only if
        system option selected.  The above code will always work;
%mend;
```

REFERENCES

Note: all URLs quoted or cited herein were accessed in April 2014.

Billings T. An Overview of Syntax Check Mode and Why it is Important . *WUSS Conference Proceedings,* 2013. URL: http://wuss.org/Proceedings13/12_Paper.pdf

Billings, T. Differences in Functionality of Error Handling Features, SAS® Enterprise Guide vs. Batch. *WUSS Conference Proceedings,* 2014. URL: http://www.wuss.org/proceedings14/13_Final_Paper_PDF.pdf

Gaudana S, Sezek B. Error Handling: An Approach for a Robust Production Environment. *NESUG Conference Proceedings,* 2005. URL: http://www.nesug.org/proceedings/nesug05/ap/ap9.pdf

SAS Institute, Inc. *SAS(R) 9.3 Language Reference: Concepts, Second Edition.* Online documentation:

- Error Processing in SAS https://support.sas.com/documentation/cdl/en/lrcon/65287/HTML/default/viewer.htm#n1nzmsupywf45qn1m6j1eczg1cb4.htm

SAS Institute, Inc. *SAS(R) Data Integration Studio 4.2: User's Guide.* Online documentation:

- Macro Variables for Status Handling http://support.sas.com/documentation/cdl/en/etlug/60948/HTML/default/viewer.htm#p0p2c5hzlgy6acn18ql0fpetn885.htm

CONTACT INFORMATION:

Thomas E. Billings
MUFG Union Bank, N.A.
Basel II - Retail Credit BTMU
350 California St.; 9th floor
MC H-925
San Francisco, CA 94104

Phone: 415-273-2522
Email: tebillings@yahoo.com