Paper 28-26

Error Trapping Techniques for SCL

Andrew Rosenbaum, Trilogy Consulting, Kalamazoo, MI

ABSTRACT

An often-overlooked aspect of applications programming is error trapping. There are always unexpected situations and a good program should catch as many errors as possible. There are two new classes in Version 8 of the SAS® System to assist the applications developer in error trapping. These are the SCLException class and the ProgramHalt class. This paper will introduce these classes to the SCL programmer. Also discussed will be a simple technique for detecting errors within SUBMIT blocks. Finally, mechanisms for communicating error messages to the user will be discussed.

INTRODUCTION

The goal of this paper is to introduce the SCL programmer of any level to the SAS System's new error trapping classes and to discuss error trapping in general. There is also discussion of detecting errors in SUBMIT blocks. SAS/AF® is required to create an SCL program but not to run it. The examples were created using the SAS system version 8.1 using Windows 98.

NEW CLASSES:

PROGRAMHALT

Using the ProgramHalt class in an SCL application is a very simple way trap general errors. This class is easy to set up and to use and can be set up to either halt a program or to let it continue. Once instantiated, the ProgramHalt class can detect several specific run-time errors as well as detecting errors that do not fit into a specific category. The specific errors are overflow, underflow, zero divide, generic math, and generic errors. The programmer subclasses the ProgramHalt class and decides what to do when an error occurs. A separate section can be written for each of the specified error conditions. The new subclass is instantiated and once that is done any time that an error occurs in the application, the ProgramHalt subclass will automatically trap the error.

Example 1:

```
/* Create class Halt from the ProgramHalt class
*/
class sasuser.classes.Halt.class extends
             sashelp.classes.ProgramHalt.class;
  _onZeroDivide: method / (STATE='O');
    put 'A DIVIDE BY ZERO ERROR HAS OCCURRED';
    /* other code may be placed here */
  endmethod;
endclass;
```

Now, whenever a divide by zero event occurs, the _onZeroDivide method will be called automatically by SAS:

```
/* Program that contains division by zero */
import sasuser.classes.halt.class;

INIT:
   dcl num1 num2;
   dcl halt errorobj=_new_ halt();
return;

MAIN:
   num1=0;
   num2=6/num1;
   put 'After the error';
return;

TERM:
   errorobj._term();
return;
```

Upon execution, the following will be displayed to the log:

```
A DIVIDE BY ZERO ERROR HAS OCCURRED
```

In this case, the program will stop executing when it encounters the error. The PUT statement in the MAIN section is not run.

There is an attribute on the ProgramHalt class that dictates whether the application stops when an error is encountered or if it is allowed to continue. The attribute, called stopExecution, defaults to "Yes". This means that if any of the described errors occurs the application will halt. This can also be set to "No" so that the application may continue after an error occurs.

Here is the same HALT class with a method added to it to handle generic errors:

Example 2:

```
class sasuser.classes.Halt.class extends
            sashelp.classes.ProgramHalt.class;

  _onZeroDivide: method / (STATE='O');
    put 'A DIVIDE BY ZERO ERROR HAS OCCURRED';
  endmethod;

  _onGeneric:method / (STATE='O');
    put 'A GENERIC ERROR HAS OCCURRED';
  endmethod;
endclass;
```

This example contains a generic error and sets the stopExecution attribute to 'No'.

```
/* Program that contains a generic error */
import sasuser.classes.Halt.class;
INIT:
   dcl halt errorobj=_new_ halt();
   errorobj.stopExecution='No';
return;

MAIN:
   /* the list FAKELIST does not exist */
   /* this will cause a generic error */
```

```
   rc=insertc(fakelist,'fred',-1);
   put 'After the error';
return;

TERM:
   errorobj._term();
return;
```

The log shows:

```
A GENERIC ERROR HAS OCCURRED
After the error
```

Even though an error has occurred notice that the PUT statement has been executed. Since the StopExecution attribute was set to 'No', the program continued to run after the error. The StopExecution attribute may be turned on or off as needed.

The HaltProgram class is a simple way to detect errors and to notify the user that something unwanted has occurred. It also offers the programmer the option of halting the program. However, it is limited in its functionality and may not be suited to all error-trapping needs. If that is the case, then the SCLException class may be the right tool for the job.

SCLEXCEPTION:

The next new class to explore is the SCLException class. This class is more powerful and, of course, more complex. It is used when a specific error condition might occur and the programmer wants complete control of the situation. It also offers the chance to recover from an error, if possible. Courses of action can be written into the program or the program can ask the user to take appropriate action.

The SCLException class uses the concept of catch-throw exception handling. When an error occurs in a called program, the called program throws an exception. The calling program "catches" it in a block of code called a catch block. If there is a stack of calling programs and if the calling program does not contain a catch block, the thrown exception continues to be passed up the chain of calling programs until it finds a catch block. If none are found, then the exception is treated the same as a program halt.

Example 3:

(This example is from the SAS ONLINE documentation.)

```
/* Create a method that intentionally throws an
exception */
import sashelp.classes;
class y;
   testMeth: method;
     throw _new_ SCLException('Exception in
                         method testMeth');
   endmethod;
endclass;

/* Create a class that has a method that calls
the previous method */
import sashelp.classes;
class x;
 m: method;
 /* catch block must be within a DO statement */
 do;
   /* SCLE will hold the information from the
                            thrown exception */
   dcl SCLException scle;
   dcl y y=_new_ y();

   y.testMeth();
   catch scle;
     put scle.getMessage();
   endcatch;
  end;
 endmethod;
endclass;
```

The log shows:

```
Exception in method testMeth
```

When the method TESTMETH is called, an exception is thrown. Back in the calling program (method m) the catch block catches the exception and the appropriate message is displayed. The program does not automatically halt. If the programmer wants the program to halt, it must be explicitly programmed to do so.

Example 4:

```
import sashelp.classes;
class x;
  m: method;
  do;
   dcl SCLException scle;
   dcl y y=_new_ y();

   y.testMeth();

   catch scle;
     put scle.getMessage();
     /* send message to user */
     /* end application */
   endcatch;
   put 'This will never run';
  end;
```

The log shows:

```
Exception in method testMeth
```

If an exception is thrown, the catch block is executed. After that control passes to the next statement after the do loop. If an exception is not thrown, control passes sequentially as it normally does until it reaches the CATCH block. At this point control passes out of the DO loop. As a result, the PUT statement 'This will never run' will never be executed.

The SCLException class offers several advantages to the applications developer. First, it allows the programmer to trap errors where, if possible, a recovery can be made. Second, it allows all of the error handling code for a section to be grouped in one place. For example, several method calls are made in the main part of a program. One CATCH block can be used to handle an error from any of the method calls. (See example 5) This is much easier than coding a separate error trap for each method call. Last, it results in an easier to read program because the error trapping code is within catch blocks. It will be fundamentally the same from application to application and from programmer to programmer instead of each one having its own method for handling errors.

Example 5:

```
MAIN:
  do;
    dcl SCLException sclerr;
```

```
    /* call to method 1 */
    obj.meth1();

    /* call to method 2 */
    obj.meth2();

    /* call to method 3 */
    obj.meth3();

    catch sclerr;
      put sclerr.getMessage();
    endcatch;
  end;
return;
```

TRAPPING ERRORS WITHIN SUBMIT BLOCKS:

When an error occurs during execution of a submit block, there is no return code to communicate to the SCL code that an error has occurred. This can cause trouble if something has gone wrong within the SUBMIT block and the application (and the user) is unaware of the problem

For example, a submit block is used to update a data set with some new information but during the update something goes wrong. The SCL code is unaware of the error and continues the application. The user assumes that since there is no error message that everything went well but in reality, the update never took place.

If it were possible to detect the error within the SCL code then the user could be issued a message that something has gone wrong and the program could end gracefully and the user could get the problem fixed.

Using the system macro variable SYSERR, most errors and warnings that are issued from Base SAS can be detected. The SYSERR variable equals 4 when a warning has occurred, 0 when no error or warning occurred and some other positive number when an error has occurred.

WARNING! Not all error conditions set the SYSERR variable! Some Proc's and some data step situations can still result in SYSERR being set to zero. Here is a partial list of procedures that do not set SYSERR when an error and/or warning occurs:

Proc Format
Proc Datasets
Proc Printto
Proc SQL
Proc Mixed
Data step with input statement. (Check the automatic variable _error_ instead.)

If this error trapping technique is to be implemented, run a test on the data step or procedure to see if SYSERR is set during an error or warning condition.

Example 6:

```
INIT:
   submit continue:
     data step;
       set work.fred;
      run;
    endsubmit;
    rc=symgetn('syserr');
    if rc=4 then put 'A Warning has occurred
                      within the submit block';
    else if rc ^=0 the put 'An error has occurred
                      within the submit block';
    else put 'Everything is hunky-dory!';
return;
```

Since data set WORK.FRED does not exist, SYSERR is set to a positive number other than 4 (1012 in this case). Reading this variable in the SCL code will tell us if an error has occurred.

Remember to always use SYMGETN to retrieve SYSERR. This is done for two reasons. First, if &syserr is used, then macro resolution will be done at compile time. This will not be very helpful when the application is actually run since SYSERR is equal to missing at compile time. Second, using SYMGETN instead of SYMGET will convert the character macro value into a numeric SCL variable.

Another thing to remember is that SYSERR is reset after each data step or procedure. Therefore, additional procedures must be in place if there are multiple data steps and/or procedures within a submit block. Each data step or procedure could be separated out into its own submit block and then SYSERR could be checked after each submit block. This is certainly possible but definitely impractical if there are more than a few data steps or procedures.

The practical solution is to enclose the base SAS code within a macro. Then check SYSERR after each step and set a macro variable to flag if an error occurs.

Example 7:

```
INIT:
  submit continue;
    %global error_rc;
    %global warn_rc;

    %macro PROCESS;
      data eric;
        set kenny;
        killed='YES';
      run;
      %if &syserr=4 then %let warn_rc=YES;
      %else %if &syserr ^= 0 %then error_rc=YES;

      proc sort data = eric;
        by date age;
      run;
      %if &syserr=4 then %let warn_rc=YES;
      %else %if &syserr ^= 0 %then error_rc=YES;
    %mend PROCESS;
    %PROCESS
  endsubmit;
  if symget('warn_rc')='YES' then put 'A warning
          has occurred within the submit block';
  else if symget('error_rc') ='YES' then put 'An
    error has occurred within the submit block';
return;
```

This can be taken one step further by creating a macro for the %if-%then-%else statement. The macro TESTIT can be placed at the beginning of the application and will be available anywhere during the execution of the program.

```
INIT:
  submit continue;
    %global error_rc;
```

```
    %global warn_rc;

    %macro TESTIT;
      %if &syserr=4 then %let warn_rc=YES;
      %else %if &syserr ^= 0 %then error_rc=YES;
    %mend;

    %macro PROCESS;
       data stan;
          set kyle;
          killed='NO';
       run;
       %TESTIT

       proc sort data = stan;
          by date age;
       run;
       %TESTIT
    %mend PROCESS;
    %PROCESS
  endsubmit;
  if symget('warn_rc')='YES' then put 'A warning
          has occurred within the submit block';
  else if symget('error_rc') ='YES' then put 'An
    error has occurred within the submit block';
return;
```

MESSAGES:

Once an error has occurred there are two tasks that must be accomplished. First the user must be alerted to the error and second, a record of the error should be made so that a programmer can have as much information as possible to use to correct the error.

There are several methods for communicating messages to the user of an application. Pop-up messages are very popular and simple to program in a Windows environment. A programmer can create a custom, generic message box using a SAS Frame. This frame can be used by any application to communicate a message to the user.

There is a new function called MessageBox in Version 8 of the SAS system. This calls a Windows DLL and pops up a Windows message box. There are several parameters so that the programmer can customize the message box for the specific situation. There is a choice of icons that can be used to signal an error, a warning, or just a friendly reminder. The title can be set as well as the contents of the message. There is also a choice of buttons that can be displayed. For example, an OK button, or a choice of YES and NO buttons. (see MESSAGEBOX in the ONLINE documentation for full documentation.)

Example 8:

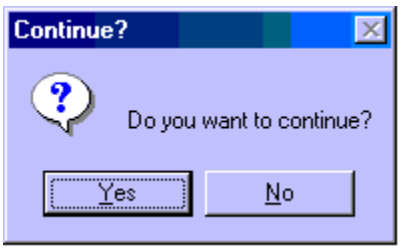
```
/* This example displays a message */
/* The letter 's'  displays an error icon  */
/* 'o' places an OK button in the message box
*/
INIT:
  id=open('work.fred');
  if id = 0 then do;
    message_list=makelist();
    rc=insertc(message_list,'Data set OPEN
failed',-1);

  response=MESSAGEBOX(message_list,'s','o',
                                    'SAS Error');
  end;
return;
```

Example 9:

```
/* This example asks the user for a response */
/* The question mark displays a query icon   */
/* 'YN' places two buttons in the message box -
                                    YES and NO */
length response $3;
INIT:
  message_list=makelist();
  rc=insertc(message_list,'Do you want to
continue?',-1);

  response=MESSAGEBOX(message_list,'?','yn',
                                   'Continue?');
  if response='Yes' then do;
    /* continue program */
  end;
  else do;
    /* discontinue program */
  end;
return;
```



In version 6.12, the same Windows message box can be displayed using the SAS function MODULEN. This is a little more complicated but once it has been set up, is simple to use.

Messages to a log are also important. Using PUT statements, Call Putlist, and the SYSMSG() function are useful tools for determining why an error has occurred. Using the ALTLOG option in CONFIG.SAS to create a copy of the log to a file is a great help in debugging applications that have terminated prematurely.

CONCLUSION

There is nothing more frustrating to the user of an application than to have it terminate abnormally. The user cannot complete his task and doesn't know why. (Besides, there is already a very large software company that has exclusive rights to leaving users in the dark with useless error messages.) Several error trapping techniques and tools have been discussed. When used in conjunction with carefully designed and well thought-out coding should make for a robust application. There is no need for a user to be left frustrated by an application. As programmers it is our responsibility to find out if an error has occurred, fix it, if possible, and alert the user that something has happened and what he might be able to do to fix it.

ACKNOWLEDGMENTS

The author would like to thank Dean Clous and Ken Baker for their assistance in preparing this paper.

CONTACT INFORMATION

Contact the author at:

Andrew Rosenbaum
Trilogy Consulting
5278 Lovers Lane
Kalamazoo, MI 49002
Phone: 616.344.4100
Email: andrew_rosenbaum@trilogyusa.com
Web: www.trilogyusa.com