



ON TARGET
software
www.targetsoft.com

Error Simulator
Programmer's Guide
Revision 1.0

Copyright 1997 On Target Software All Rights Reserved



Table of Contents

PLEASE READ FIRST	4
ERROR SIMULATION OVERVIEW	5
WHAT IS AN ERROR HANDLER?	5
WHY TEST ERROR HANDLERS AT ALL?	5
WHAT IS ERROR SIMULATION?	5
WHY TEST WITH ERROR SIMULATION?	5
GETTING STARTED	6
BUILD ENVIRONMENT	6
DOWNLOADING AND INSTALLING	6
BUILDING THE SAMPLE	7
INTEGRATION STEPS	7
CALLING SIMULATION FUNCTIONS	8
NOW FOR THE REAL WORK	8
SIMULATING ERRORS AND EXCEPTIONS	9
INSERTING CALLS TO ERROR SIMULATORS	9
SIMULATING SYSTEM RESOURCE ALLOCATIONS	9
SYSTEMATIC VS. RANDOM TESTING	9
HOW MANY ERRORS WILL I FIND?	10
BUT IT CLUTTERS THE CODE	10
AND IT'S REALLY BORING	10
ADDING AN EXCEPTION TYPE	11
DEFINE A NEW ERROR TYPE	11
MODIFY THE ERROR INFORMATION TABLE	11
THROW THE EXCEPTION FROM THE CALLBACK ROUTINE	12
DEFINING ADDITIONAL SIMERR FUNCTIONS	13
INTERPRETING SIMULATOR OUTPUT	14
OUTPUT AT STARTUP	14
OUTPUT BEFORE SIMULATING AN ERROR	14
DETAILED REPORT, ERRORS SIMULATED	14
DETAILED REPORT, ERRORS NOT SIMULATED	15
SUMMARY REPORT	15
FUNCTION AND DATA STRUCTURE REFERENCE	16
HOW THE SIMULATOR FUNCTIONS WORK	16
SIMERRINIT	17
SIMERREXIT	18
SIMERRENABLE	19
SIMERRRESULT	20
SIMERRZERO	21
SIMERRNEW	22
SIMERRMALLOC	23
SIMERR EXCEPTION	24
MORE FREE SOFTWARE	25
C++ NOTIFIERS	25
DEBUG MESSAGE LOGGING	25
WHY ALL THIS FREE SOFTWARE?	25
HOW TO REACH ON TARGET SOFTWARE	26
REVISION HISTORY	26
1.0 MONDAY, NOVEMBER 17, 1997	26



Code Samples

Code Sample 1. The <i>PrintSimulatorMessage</i> callback routine.	7
Code Sample 2. Inserting calls to error simulation functions.	9
Code Sample 3. The error type definitions.	11
Code Sample 4. The error information table.	11
Code Sample 5. The <i>ThrowSimulatorException</i> callback routine.	12
Code Sample 6. Defining additional simulation functions.	13
Code Sample 7. Detailed report of errors simulated.	14
Code Sample 8. Detailed report of errors not simulated.	15
Code Sample 9. Summary report.	15
Code Sample 10. Calling an error simulation function	16
Code Sample 11. The <i>SimErrNew</i> template definition	16
Code Sample 12. <i>simerrInit_t</i> structure definition.	17
Code Sample 13. Example: <i>SimErrResult</i>	20
Code Sample 14. Example: <i>SimErrZero</i>	21
Code Sample 15. Example: <i>SimErrNew</i>	22
Code Sample 16. Example: <i>SimErrMalloc</i>	23
Code Sample 17. Example: <i>SimErrException</i>	24



Please Read First

The documentation and source code is free for your use in any software you develop. You may not, however, copy the source files or documentation or any significant part of them without also including the copyright notice and web address for On Target Software in the source code and the source code documentation.

If others want a copy of the source code, please direct them to my website at www.targetsoft.com.

Neither On Target Software nor Dave Pomerantz assume any responsibility or liability for errors, incorrect results, or other problems that may occur with these files or with software in which these files are embedded. Use at your own risk.

If you run into problems with the source code or if you have questions, please send e-mail through the [Contact](#) link on my website. If you want to send attachments, contact me and I'll reply. That way I can keep my email address out of the view of spam robots.

The error simulator is freely available with these simple restrictions. If you cannot abide by them, please don't use the software.

Thank you!



Error Simulation Overview

The error simulation library has three parts:

- Small inline functions and function templates called from the target program to simulate errors and throw exceptions. You can add to these.
- A small custom source file that adapts the generic error simulation to the specific needs of your program.
- An underlying library that produces reports and logs which errors have been simulated.

To test your program, you insert calls to simulation functions, then you run your program repeatedly, observing its behavior in the face of the simulated errors.

When you build for release, the conditionally compiled simulation functions do not appear in the linked program.

What is an error handler?

Anywhere a program tests the return code of a function or catches an exception, it's handling an error. In my own software, I've found that testing, propagating, and responding to errors makes up between 7% and 34% of the executable lines of code in my own programs, averaging about 20%. It's lowest in modules that have few external interfaces. It's highest in I/O modules and routines that interact heavily with the system.

Why test error handlers at all?

Error handlers don't normally execute so they get less attention than feature-driven software. Instead, they're entirely responsible for robustness: the ability of a program to continue in the face of hardware failures, system errors and resource limitations. Robustness is important to commercial applications, but it's essential to embedded and mission-critical software that cannot afford to abort or corrupt their data when handling failures.

Only the most trivial software has the luxury of aborting without cleaning up. Most programs will need to close files and databases, or release communications ports or shared memory. Without testing error handlers, you can safely assume the cleanup will fail.

What is error simulation?

Error simulation is a developer's tool for finding defects in error handlers by forcing errors at the point of detection. Immediately after calling a function that can fail, the programmer inserts a call to a function that simulates the failure, either by returning a failure code or by throwing an exception.

Why test with error simulation?

You can perform limited tests of error handlers by running on a machine with low memory or faulty hardware. This is hit-or-miss and will execute only a few error handlers. Worse, tests are nearly impossible to reproduce, making it difficult to fix problems. Regression testing is hopeless.

The error simulator exercises all error handlers in a repeatable way that can be automated with commercial test software.

For many products, testing and fixing all error handlers using error simulation is faster and cheaper than finding and fixing a single bug in a single error handler, when the only facility for reproducing the bug is an intermittent hardware failure.



Getting Started

Soon you'll be making changes to your software to integrate the error simulation. Most programmers have a rational fear of introducing foreign code to their system. To make this as painless as possible, the following steps let you dip your toe into the water. You can try out the error simulation and if it doesn't suit your needs you can easily remove the files and function calls from your system.

After you download and install the software, you'll need to:

- build and test the sample program, to make sure it all works as advertised
- copy the header files
- copy a small source file and customize it to your system requirements
- call the simulator init and exit functions from your main routine
- call a few simulator functions
- build with SIMERR #defined and link to the simulator DLL
- run your program and interpret the results

This whole process should take two or three hours, depending on problems with the build environment, makefiles, etc. When everything goes smoothly, it takes less than an hour, but nothing ever goes smoothly.

Build environment

The error simulator is built with Visual C++ 5.0. That means you need to be running on a Win32 platform and your target application must be written in C++. You can use another C++ compiler, like Borland, with some modifications to the makefile and possibly to the source.

Other than the requirements for building a Windows DLL, I tried to stick to ANSI C++. It should not be difficult to build the software for another C++ environment on a different platform.

It's also possible for you to rewrite the simulator in C or Pascal or any other language. After all, it's only about 1200 lines of code. If you do port the software, please let me know.

Downloading and installing

Extracting the files

1. Download the source files, if you haven't already. They are in a 32-bit Windows self-extracting file built with WinZip®.
2. Run the downloaded file and extract the error simulation files.

Files and folders

Assuming you extracted into C:\ONTARGET, The files are organized as follows:

C:\ONTARGET\BIN\	contains binary files, including the error simulation dll and lib files, and the release and debug versions of the sample program. The projects are setup to write .exe, .lib, and .dll files to the bin directory. To prevent name conflicts, all debug versions of DLL's are appended with the letter 'd'.
C:\ONTARGET\CUSTOM\	contains files that you copy and customize to integrate error simulation into your application
C:\ONTARGET\DOC\	contains this document
C:\ONTARGET\INCLUDE\	contains simerr header files that will be included in the sample program and in your own application
C:\ONTARGET\SAMPLE\	contains source code for the sample program
C:\ONTARGET\SIMERR\	contains source code for the error simulation library



Building the sample

To build and test without modifying the simulator, you need the Microsoft Visual C++ development environment. Lacking that, you may need to make minor changes to the source code and/or make files prior to building. The following discussion assumes you have Visual C++ 5.0 or later.

Build the error simulation DLL

You could reasonably skip this step, since the `simerr.dll` file is already present in the `bin` directory.

To build `simerr.dll`, open the `simerr.dsw` workspace in the `simerr` directory. Build the release version of the error simulator. Unless you need to debug the simulator itself, there's no need to build the debug version of the simulator.

Build the sample program

Open the `simerr_sample.dsw` workspace in the `simerr_sample` directory. Build the debug version of the `simerr_sample` program. It's a simple console application that runs from a DOS shell. You can examine the `simerr_sample` program and its output as you read the rest of the documentation..

Integration Steps

These steps integrate error simulation into your application. After you've tried out the process, you can add exception simulation for a more thorough test.

Copy the library files

- copy `simerr.dll` and `simerr.lib` to the directory you use for libraries and DLL's. You'll find these two files in the `ontarget\bin` directory.

Copy the header files

- Copy `simerr.h` and `simdef.h` from the `ontarget\include` directory
- Copy `simcust.h` file from the `ontarget\custom` directory

Put them in your project's include path.

Copy and modify the custom simulator file

- Copy `simcust.cpp` from the `ontarget\custom` directory to your project source directory
- Modify the `PrintSimulatorMessage` function to write simulator messages to your debug trace output, as described immediately below.

The `PrintSimulatorMessage` function is a callback routine that directs the output of all the simulator messages and the simulator report. Most large applications have a trace or log mechanism.¹ You should use your own debug trace mechanism, so the simulator's messages are interleaved with the normal trace output of your program.

Here's an example of the `PrintSimulatorMessage` function modified to use the Visual C++/MFC trace output:

```
static void PrintSimulatorMessage(const char *pMsg)
{
    TRACE("<SIMERR> %s\n", pMsg);
}
```

Code Sample 1. The `PrintSimulatorMessage` callback routine.

¹ If you don't have a message trace facility or you're not satisfied with the one you have, I'm planning to post a powerful debug logger at the On Target Software website. If it's not there yet, please send email. Don't let me get lazy.



Note that messages passed in from the simulator do not have a carriage-return, so *PrintSimulatorMessage* appends one.

Fatal Error C1010

If you're building an MFC application, you may get the message:

```
fatal error C1010: unexpected end of file while looking for precompiled header  
directive
```

Be sure you've included `stdafx.h` in `simcust.cpp`, if you're building with MFC. Alternatively, you can build the one file without precompiled headers.

Call Init and Exit functions

From your main program, call *SimErrCustomInit* at application startup and *SimErrExit* at application shutdown. To the extent possible, these should be the first and last functions called.

#define SIMERR and link with simerr.lib

In your compiler settings for your debug version, or in one of your main header files, define SIMERR. Make sure it's not defined for the release version.

In your linker settings, link with `simerr.lib`.

Calling simulation functions

Now that your application is ready for error simulation, you need to simulate a few errors and try it out.

Including simulation header

Every function that calls a simulation function needs to include `simcust.h`.

Simulate an error

Pick a function that returns an error. Insert a call to an error simulation function below it. There are many examples of this below, in the section on *Simulating Errors and Exceptions*.

Compile and test

Compile your application and run a test. For information on reading the SIMERR output, see the section on *Interpreting Simulator Output*.

Now for the real work

The real work involves testing and debugging that hidden twenty percent of your code: error detection, propagation, and recovery. It's not easy. It requires planning and diligent testing throughout the development cycle.

But the result can give your company a substantial competitive advantage.



Simulating Errors and Exceptions

Inserting calls to error simulators

To test a program, identify all the places where the program checks a function result for an error or where a called function can throw an exception. These are the failure points. At each failure point, insert a call to the most appropriate error simulation function, to simulate the error.

Here are some examples:

Calling any of your internal functions:

```
hr = InitSensor();  
hr = SimErrResult(SIMLINE, hr);
```

Calling a function that can throw an exception:

```
m_temperatureArray.SetSize(newSize);  
SimErrException(SIMLINE, SIMERR_MEMORY_EXCEPTION);
```

Instantiating an object:

```
m_pTemperatureThread = new CWinThread(TemperatureThreadProc, &m_hEvent);  
m_pTemperatureThread = SimErrNew(SIMLINE, m_pTemperatureThread);
```

Allocating system resources:

```
bCreated = Create("OK", BS_PUSHBUTTON | WS_CHILD | WS_VISIBLE,  
                 m_rectWindow, pParent, nID);  
bCreated = SimErrZero(SIMLINE, bCreated);
```

Code Sample 2. Inserting calls to error simulation functions.

Simulating system resource allocations

In some cases, the function whose failure is simulated has created a system resource. If that function actually failed, no resource would have been created. But since we're only simulating a failure, the resource was created and we simply substituted a failure return code. In this case the simulation is not behaving like a true failure.

This problem occurs in the fourth example, when *CWnd::Create* creates a new window but *SimErrZero* doesn't destroy the window. To solve this problem, we need to create a simulation function, *SimErrCreateWin*, to automatically destroy a window as part of the error simulation. Creating new simulation functions is simple and is described in the section *Defining Additional SimErr Functions*. For any given application and environment, you may have to create five or ten of these functions.

Systematic vs. random testing

A systematic test simulates each error exactly once, the first time the simulation function is executed.

A random test uses two parameters, present in all error simulation functions, that determine the probability the error will occur when the function is executed, and the maximum number of times it will occur. In all functions, these parameters default to a probability of 1.0 and a maximum count of 1, which is identical to a systematic test. So all functions default to a systematic test.

By specifying the probability and maximum count to be other than the default, you can create tests that are a better representation of a real-life scenario. Let's say your program controls a valve in a chemical plant. To test operation of the program with an intermittent failure, you might have code like this:

```
BOOL bOK = AdjustValve();  
bOK = SimErrZero(SIMLINE, bOK, 1000, 0.0001);
```

This will simulate a valve failure once in ten-thousand executions, up to a maximum of 1000 failures.



You may want to code a lot of your error simulations just like this, for complex stress-testing of your software. You may also want to test all your error-handlers one-by-one to make sure each one works. To allow for this, the *SimErrInit* function has a parameter that determines the type of test. If the test type is SIMERR_RANDOM, all the simulation functions are run with their parameters interpreted as you wrote them.

If, however, the test type is SIMERR_SYSTEMATIC, the count and probability parameters passed to the functions are ignored, and each function triggers an error exactly once, the first time it's executed.

This way, you can switch back and forth between systematic and random testing, without changing the parameters on all the error simulation function calls.

How many errors will I find?

I've used this technique extensively on previous projects. To test this most recent incarnation of the error simulator, I used my notifier utility and thermostat program, both of which are available on the website.

Here are some of the mistakes I made that were not apparent and would never have been discovered without the error simulation. This is where error simulation shines: it uncovers errors you cannot find any other way.

1. Found an improper deletion of an exception in a catch clause, which caused a memory protection error.
2. Found a bug in list synchronization that could have occurred during normal multi-threaded operation. This was a bug that was not in the error-handler but was forced into the open by the simulated error.
3. Found a place where a SetFont message was being sent even if the CreateWindow operation failed. This caused a crash.
4. Found that the program failed quietly when it failed during initialization, and this was disconcerting. So I inserted an error message and validated that it appeared under all initialization failures.

To test the programs, I inserted 57 calls to error simulation functions and detected four bugs, three of which caused crashes. The process took about three hours. In a few places, I had some minor restructuring of the error-checking in order to insert the error simulation statements.

What did I get out of it? I fixed four bugs I wouldn't have found any other way, with less than three hours of work, and I now know that my software is more robust than most.

But it clutters the code

Yes it does. Even if the error simulation isn't present at run-time, you still have to look at the function calls and it makes the code more complex to the eye.

That's a definite disadvantage. I don't know anyway around that, but the benefits are large enough that I'm willing to put up with it.

And it's really boring

Testing is tedious. You might spend an hour testing just to find one or two bugs. Bear in mind that QA people who test all day long are having a great day when they find one bug per hour. And they do it all day, every day.

You'll probably find more than one bug per hour the first time you use the simulator. For a programmer accustomed to fast progress developing features and fixing bugs, it may still seem like a waste of time, but it's a productive way to test and you'll be finding difficult, well-hidden bugs much faster than the QA people could. You'll also gain a lot of respect for your QA department (and you might get some respect from them, too!)



Adding an Exception Type

To simulate exceptions, the error simulator calls *ThrowSimulatorException*, a callback function in the custom simulator file.

There are three steps to modifying the custom simulator file to throw a particular exception:

- define an error type that you use to request the exception
- add the new error type to the table of error types
- throw the exception from *ThrowSimulatorException*.

Define a new error type

The `ERROR_TYPE` enum is in the custom error simulation header, `simcust.h`, that you copied during integration. It looks like this:

```
enum ERROR_TYPE
{
    // IMPORTANT: The first two entries must always be
    //             SIMERR_RESULT_ERROR and SIMERR_MEMORY_ERROR
    SIMERR_RESULT_ERROR,    // failing function returns an error code
    SIMERR_MEMORY_ERROR,    // failed allocation returns zero
    NUM_SIMERR_TYPES
};
```

Add the new type in the line immediately before `NUM_SIMERR_TYPES`. It's important that `SIMERR_RESULT_ERROR` and `SIMERR_MEMORY_ERROR` remain the first two error types. Your change might look like this:

```
enum ERROR_TYPE
{
    // IMPORTANT: The first two entries must always be
    //             SIMERR_RESULT_ERROR and SIMERR_MEMORY_ERROR
    SIMERR_RESULT_ERROR,    // failing function returns an error code
    SIMERR_MEMORY_ERROR,    // failed allocation returns zero
    SIMERR_MEMORY_EXCEPTION, // throw a CMemory exception
    NUM_SIMERR_TYPES
};
```

Code Sample 3. The error type definitions.

Modify the error information table

The error information table is in the custom error simulation source file, `simcust.cpp`, that you copied during integration. It looks like this:

```
static const errorInfo_t g_errorInfo[] =
{
    // type          name          error or exception
    { SIMERR_RESULT_ERROR, "RESULT", SIMERR_TYPE_ERROR },
    { SIMERR_MEMORY_ERROR, "MEMORY", SIMERR_TYPE_ERROR }
};
```

The table must be modified to exactly match the `ERROR_TYPE` enum. To match the change made above, you would change `g_errorInfo` to look like this:

```
static const errorInfo_t g_errorInfo[] =
{
    // type          name          error or exception
    { SIMERR_RESULT_ERROR, "RESULT", SIMERR_TYPE_ERROR },
    { SIMERR_MEMORY_ERROR, "MEMORY", SIMERR_TYPE_ERROR },
    { SIMERR_MEMORY_EXCEPTION, "MEM XCPT", SIMERR_TYPE_EXCEPTION }
};
```

Code Sample 4. The error information table.

Note that the third data value of your new entry, `SIMERR_TYPE_EXCEPTION`, tells the error simulator to call the exception callback routine



Throw the exception from the callback routine

The callback routine is called whenever an exception-type error is simulated. Modify this routine to throw the exception. Since exceptions are specific to environments and applications, the ability to throw an exception cannot be built into the general-purpose simulator library.

```
static void ThrowSimulatorException(int nErrorType,
                                   const char *pSourceFilename,
                                   int nSourceLine)
{
    switch(nErrorType)
    {
    case SIMERR_MEMORY_EXCEPTION:
        AfxThrowMemoryException();
        break;
    default:
        {
            char msg[128];
            sprintf(msg, "invalid exception type at %s(%d)",
                  pSourceFilename, nSourceLine);
            PrintSimulatorMessage(msg);
        }
    }
}
```

Code Sample 5. The *ThrowSimulatorException* callback routine.



Defining Additional SimErr Functions

An error simulation function

- decides whether to simulate an error
- deletes or destroys information associated with the error
- returns a value indicating an error

For example, the *SimErrMalloc* function frees a C allocation and returns zero. After *SimErrMalloc* executes, it's as if the memory was never allocated.

What if you create a window during the initialization of a *CWnd* object? You could use the *SimErrZero* function to simulate a failed creation, like this:

```
BOOL bCreated;  
bCreated = Create("OK", BS_PUSHBUTTON | WS_CHILD | WS_VISIBLE,  
                 m_rectWindow, pParent, nID);  
bCreated = SimErrZero(SIMLINE, bCreated, this);
```

The problem is that the window really was created and we're just pretending it wasn't. As we respond to the failed creation, the window will still be there, which is wrong. We'd like to have a function that destroys a window when it simulates a *CWnd::Create* failure.

If you look in *SIMERR.H*, you'll find a number of function templates and inline functions defined. Copy the most appropriate one and add your own error simulation function. To simulate an error during MFC window creation, I wrote this function:

```
inline BOOL SimErrCreateWin(const char *strFile, int nLine, BOOL bCreated, CWnd  
*pWnd,  
                           long nCount = 1, double nProb = 1.0)  
{  
    if ((bCreated) &&  
        SimErr(strFile, nLine, SIMERR_RESULT_ERROR, nCount, nProb))  
    {  
        pWnd->DestroyWindow();  
        bCreated = FALSE;  
    }  
    return bCreated;  
}
```

Code Sample 6. Defining additional simulation functions.

Now when you call *SimErrCreateWin*, the window will no longer exist when the error simulation function returns.

For more information on how to write simulation functions, see *How the simulator functions work* in the reference section.



Interpreting Simulator Output

Output at startup

When the simulator starts, it calls *PrintSimulatorMessage* to report that it's simulating errors. The message indicates if the simulation is systematic

```
<SIMERR> Starting error simulator with a systematic test.
```

or random. If it's a random test, the random number seed is given, in case you need to exactly replicate the session.

```
<SIMERR> Starting error simulator with random number seed 20442.
```

The simulation log file is always copied before starting, and the simulator tells you the name of the backup copy. This also helps when you need to exactly replicate a session:

```
<SIMERR> backup log file written to "bkuplog.txt"
```

Output before simulating an error

Immediately before simulating an error, *PrintSimulatorMessage* is called, with a message like this:

```
<SIMERR> at thermostat.cpp(84) *** SIMULATING MEMORY ERROR ***
```

If the error handler fails, you know where to look to find the line that triggered the error.

Detailed report, errors simulated

When *SimErrExit* is called, it generates a summary report with the total number of errors simulated, and the new errors simulated in the current session. If you request it, it will also generate a detailed report, listing each error it has simulated in this and prior sessions.

Here's a sample of a detailed report:

```
<SIMERR> *** ERRORS SIMULATED ***
<SIMERR>
<SIMERR>      File                Line Error-Type Requested Total New
<SIMERR> -----
<SIMERR> thermostat.cpp             66   MEMORY           1     1   0
<SIMERR> thermostat.cpp             72   RESULT            6     1   0
<SIMERR> NEW thermostat.cpp         112  MEMORY            5     3   1
<SIMERR> view.cpp                   168  MEMORY            1     1   0
```

Code Sample 7. Detailed report of errors simulated.

Each line of the report has this information:

file/line	source file and line number of the error simulator function
error-type	type of each error, from the simulator function
requested	number of times the error was requested (the <i>nCount</i> parameter, which defaults to 1)
total	total number of times the error has been simulated in this and previous sessions
new	number of times the error was simulated in this session alone.

If this session is the first time the error was simulated, the line has *NEW* in front.



Detailed report, errors not simulated

During random testing, simulation calls may be executed without triggering errors. If this happens, the detailed report will list errors that were not simulated along with their probability. The only way an error can be passed by is if its probability is less than one, and the simulator is running a random test.

```
<SIMERR> *** ERRORS NOT YET SIMULATED ***
<SIMERR>
<SIMERR>      File                               Line Error-Type Requested Probability
<SIMERR> -----
<SIMERR>      thermostat.cpp                       84     MEMORY           10           0.10
<SIMERR>      thermostat.cpp                       124    RESULT              4           0.50
```

Code Sample 8. Detailed report of errors not simulated.

Note that this section of the report will only appear if some of the error simulations were skipped during random testing.

Summary report

Lastly, the simulator totals all the errors from this and previous sessions:

```
<SIMERR> *** ERROR SIMULATION TOTALS ***
<SIMERR>
<SIMERR> Type                Triggered Remaining Total
<SIMERR> -----
<SIMERR> RESULT                1             1         2
<SIMERR> MEMORY                3             1         4
<SIMERR> MEM XCPT              0             0         0
<SIMERR> RES XCPT              0             0         0
<SIMERR> FIL XCPT              0             0         0
<SIMERR> -----
<SIMERR> ALL TYPES             4             2         6
```

Code Sample 9. Summary report.

This shows a breakdown, by type of error, of how many errors have been simulated and how many were passed over. In a systematic test, the number remaining will always be zero.

Note that each error simulation function is counted once in the totals, even if you specified an *nCount* greater than 1. This is simply because I found that more useful.



Function and Data Structure Reference

How the simulator functions work

In the example below, *SimErrNew* simulates a C++ allocation error immediately after a call to *new*:

```
// Instantiate object with C++ allocation
pObject = new CSampleObject;
pObject = SimErrNew(SIMLINE, pObject, -1, .4);
if (pObject==0)
{
    printf("C++ allocation error\n");
    bOK = false;
}
else
{
    delete pObject;
}
```

Code Sample 10. Calling an error simulation function

SimErrNew is a C++ template function that frees an allocation and returns NULL. The debug version uses the following template:

```
// This is a convenience macro for use in the error simulation functions.
#define SIMLINE __FILE__, __LINE__

// Simulate failed C++ allocation
template <class T>
T *SimErrNew(const char *strFile, int nLine, T *pObject,
            long nCount = 1, double nProb = 1.0)
{
    if ((pObject != NULL) &&
        SimErr(strFile, nLine, SIMERR_MEMORY_ERROR, nCount, nProb))
    {
        delete pObject;
        pObject = 0;
    }
    return pObject;
}
```

Code Sample 11. The *SimErrNew* template definition

SIMLINE is a simple macro that passes the first two parameters to *SimErrNew*, the source file and line number. These uniquely identify the location of the simulation function. The third parameter, *pObject*, points to the new object in memory.

The last two parameters, *nCount* and *nProb*, apply only to a random test. They control the number of times to simulate an error at this location and the probability of an error each time. The default is to simulate the error once (*nCount* = 1), the first time through (*nProb* = 1.0).

SimErrNew calls *SimErr* to find out if it should simulate an error. *SimErr* looks up the location, finds out how many times it has already simulated an error in that location, and returns *true* to simulate an error or *false* to continue on.

If *SimErr* returns *true*, *SimErrNew* deletes¹ the object and returns zero, simulating a failed creation.

¹ The use of a template function is critical in this case, because *delete* needs to operate on the specific type of the object in order to call its constructor. A macro could also be used. An inline function with a *void ** pointer will not work.



SimErrInit

Purpose

SimErrInit initializes the error simulator:

- allocates the simulator object
- sets the pointers to the *PrintSimulatorMessage* and *ThrowSimulatorException* callback routines
- sets the array of error types
- seeds the random number generator
- loads the simulator log file
- writes a backup copy of the log file to *bkuplog.txt*.
- enables the simulator

Definition

```
bool SimErrInit(simerrInit_t *pInit);
```

Parameters

To initialize the error simulator, pass a pointer to the following structure:

```
struct simerrInit_t
{
    int                m_nVersion;    // set this to SIMERR_VERSION
    const char *      m_pFilename;    // simulator log file
    int                m_nSeed;       // random number seed
    SIMERR_TEST_TYPE  m_testType;     // systematic or random test
    pfSimErrMsg_t     m_pfMsg;        // callback for message output
    pfThrowException_t m_pfThrow;     // callback for throwing exceptions
    const errorInfo_t * m_pErrInfo;   // info on each error type
    int                m_nErrTypes;   // number of error types
};
```

Code Sample 12. simerrInit_t structure definition.

<code>m_nVersion</code>	is the simulator version number. Always pass <code>SIMERR_VERSION</code> .
<code>m_pFilename</code>	is the name of the log file the simulator creates to track which errors were simulated.
<code>m_nSeed</code>	is used to seed the random number generator. Pass <code>SIMERR_TIME_SEED</code> to tell the simulator to generate a seed based on the time-of-day.
<code>m_testType</code>	tells the simulator whether to run a random or a systematic test. Set this to either <code>SIMERR_SYSTEMATIC</code> or <code>SIMERR_RANDOM</code> . A systematic test ignores the <i>nCount</i> and <i>nProb</i> parameters passed to the simulation functions, triggering each error exactly once, the first time a simulation function is called.
<code>m_pfMsg</code>	sets the pointer to the callback routine that outputs all error messages. See the section <i>Integration Steps</i> on page 7 for details.
<code>m_pfThrow</code>	sets the pointer to the callback routine that throws exceptions. See the section <i>Adding an Exception Type</i> on page 11 for details.
<code>m_pErrInfo</code>	sets the pointer to the error type table. This is also shown in <i>Adding an Exception Type</i> .
<code>m_nErrTypes</code>	is the number of entries in the error type table.

Return value

Returns *true* on success, *false* on failure.



SimErrExit

Purpose

SimErrExit terminates the error simulator

- writes a report on which errors have been triggered
- saves the current status of triggered errors in the log file
- deallocates the simulator object

Definition

```
void SimErrExit(bool bFull);
```

Parameters

bFull if true, write a detailed report and a summary report. If false, write a summary only.

Remarks

If the target program fails to exit normally, that is, if the error handling fails, *do not call SimErrExit*. That way, the simulation log file will not be overwritten and the same error will be triggered on the next run of the target program. This allows you to continue to trigger the same error on each new run, until you've fixed the error handler and the program exits normally. Once the program exits normally, it overwrites the error log, allowing you to continue to the next error.



SimErrEnable

Purpose

SimErrEnable turns error simulation on and off.

Definition

```
bool SimErrEnable(bool bEnable);
```

Parameters

bEnable is set to *true* to enable, *false* to disable the error simulation.

Return value

Returns *true* if error simulation was enabled, *false* if it was not.



SimErrResult

Purpose

SimErrResult simulates errors in functions that return zero on success. Use this to test system functions and your own application functions that return zero on success.¹

Definition

```
template <class T>  
T SimErrResult(const char *strFile, int nLine, T nResult, T nSimResult,  
              long nCount = 1, double nProb = 1.0)
```

Parameters

SIMLINE	is always passed for the first two parameters.
nResult	is the value returned by the function whose result is being simulated. The simulator only triggers an error if this value is zero, that is, the simulator will not mask a real error.
nSimResult	will be returned by <i>SimErrResult</i> when an error is simulated.
nCount	is the maximum number of times the error will be simulated. The simulator tracks this count across sessions. Set this to SIMERR_FOREVER to simulate errors indefinitely.
nProb	This is a value between 0.0 and 1.0 that determines the likelihood that an error will be simulated.

Return value

If *nResult* is zero and it's time to simulate an error, *SimErrResult* returns *nSimResult*.

If *nResult* is non-zero or it's not time to simulate an error, *SimErrResult* returns *nResult*.

Remarks

If the test type passed to *SimErrInit* is SIMERR_SYSTEMATIC, *nCount* and *nProb* are ignored.

Example

```
HRESULT hr = UpdateDatabase();  
hr = SimErrResult(SIMLINE, hr, E_OUTOFMEMORY, SIMERR_FOREVER, 0.01);  
hr = SimErrResult(SIMLINE, hr, E_ACCESSDENIED);  
if (hr == E_OUTOFMEMORY)  
{  
    printf("Not enough memory.\n");  
}  
else if (hr == E_ACCESSDENIED)  
{  
    printf("Not authorized to make this update, please see your administrator.\n");  
}
```

Code Sample 13. Example: *SimErrResult*

In a random test, the first call to *SimErrResult* simulates an out-of-memory error roughly once every 100 times it's called, indefinitely. In a systematic test, the first call will return E_OUTOFMEMORY once, the first time.

The second call simulates an access error once, the first time.

¹ Win32 functions that return HRESULT should have their own inline function, *SimErrHResult*, which you can create using the techniques described in the section *Defining Additional SimErr Functions*. This inline function would use the Windows macros SUCCESS and FAILURE.



SimErrZero

Purpose

SimErrZero simulates errors in functions that return zero on failure. Use this to test operating system functions and your own internal application functions that return zero on failure.

Definition

```
template <class T>  
T SimErrZero(const char *strFile, int nLine, T nResult,  
             long nCount = 1, double nProb = 1.0)
```

Parameters

SIMLINE	is always passed for the first two parameters.
nResult	is the value returned by the function whose result is being simulated. The simulator only triggers an error if this value is non-zero, that is, the simulator will not mask a real error.
nCount	is the maximum number of times the error will be simulated. The simulator tracks this count across sessions. Set this to SIMERR_FOREVER to simulate errors indefinitely.
nProb	This is a value between 0.0 and 1.0 that determines the likelihood that an error will be simulated.

Return value

If *nResult* is non-zero and it's time to simulate an error, *SimErrResult* returns zero.

If *nResult* is zero or it's not time to simulate an error, *SimErrResult* returns *nResult*.

Remarks

Certain system functions, like the Windows *SetTimer* function, return a timer index on success, zero on failure. If you use *SimErrZero* to simulate failures in these function, be aware that your program will not cleanup if the system function succeeded (which it probably did) and allocated a resource. The best solution to this problem is to define your own template functions, that free these resources when simulating errors. For examples on when and how to do this, see the section on *Defining Additional SimErr Functions*.

If the test type passed to *SimErrInit* is SIMERR_SYSTEMATIC, *nCount* and *nProb* are ignored.

Example

```
bool bIsOK = AdjustValve();  
bIsOK = SimErrZero(SIMLINE, bIsOK);  
if (!bIsOK)  
{  
    printf("Failed to adjust valve.\n");  
}
```

Code Sample 14. Example: *SimErrZero*

SimErrZero simulates a failed call to *AdjustValve* by returning *false*.



SimErrNew

Purpose

SimErrNew simulates errors in C++ allocations. Use this to test out-of-memory conditions after calling *new*.

Definition

```
template <class T>  
T SimErrNew(const char *strFile, int nLine, T *pObject,  
            long nCount = 1, double nProb = 1.0)
```

Parameters

SIMLINE	is always passed for the first two parameters.
pObject	is the pointer returned by <i>new</i> .
nCount	is the maximum number of times the error will be simulated. The simulator tracks this count across sessions. Set this to SIMERR_FOREVER to simulate errors indefinitely.
nProb	This is a value between 0.0 and 1.0 that determines the likelihood that an error will be simulated.

Return value

If *pObject* is non-zero and it's time to simulate an error, *SimErrNew* deletes *pObject* and returns zero.

If *pObject* is zero or it's not time to simulate an error, *SimErrNew* returns *pObject*.

Example

```
pObject = new CSampleObject;  
pObject = SimErrNew(SIMLINE, pObject, 20, 0.4);  
if (pObject==0)  
{  
    printf("C++ allocation error\n");  
    return E_OUTOFMEMORY;  
}
```

Code Sample 15. Example: *SimErrNew*

In a random test, forty percent of the time, up to 20 times, *SimErrNew* will delete *pObject* and return zero, to simulate a failed allocation. In a systematic test, *SimErrNew* will delete the object once, the first time.



SimErrMalloc

Purpose

SimErrMalloc simulates errors in C++ allocations. Use this to test out-of-memory conditions after calling *malloc*.

Definition

```
template <class T>  
T SimErrMalloc(const char *strFile, int nLine, T *pData,  
              long nCount = 1, double nProb = 1.0)
```

Parameters

SIMLINE	is always passed for the first two parameters.
pData	The pointer returned by <i>malloc</i> .
nCount	is the maximum number of times the error will be simulated. The simulator tracks this count across sessions. Set this to SIMERR_FOREVER to simulate errors indefinitely.
nProb	This is a value between 0.0 and 1.0 that determines the likelihood that an error will be simulated.

Return value

If *pData* is non-zero and it's time to simulate an error, *SimErrNew* frees *pData* and returns zero.

If *pData* is zero or it's not time to simulate an error, *SimErrNew* returns *pData*.

Example

```
pArray = (char *) malloc(100);  
pArray = SimErrMalloc(SIMLINE, pArray);  
if (pArray ==0)  
{  
    printf("C allocation error\n");  
    return E_OUTOFMEMORY;  
}
```

Code Sample 16. Example: *SimErrMalloc*

The first time *SimErrMalloc* is called, it will delete *pArray* and return zero, to simulate a failed allocation.



SimErrException

Purpose

SimErrException simulates errors in functions that throw an exception when they fail.

Definition

```
inline void  
SimErrException(const char *strFile, int nLine, int nType,  
                long nCount = 1, double nProb = 1.0)
```

Parameters

SIMLINE	is always passed for the first two parameters.
nType	The application defined exception type. See the section on <i>Adding an Exception Type</i> .
nCount	is the maximum number of times the error will be simulated. The simulator tracks this count across sessions. Set this to SIMERR_FOREVER to simulate errors indefinitely.
nProb	This is a value between 0.0 and 1.0 that determines the likelihood that an error will be simulated.

Example

```
try  
{  
    nResult = UpdateDatabase();  
    SimErrException(SIMLINE, SIMERR_DATABASE_EXCEPTION, SIMERR_FOREVER, .001);  
}  
catch(CDeviceException *pDeviceException)  
{  
    printf("Unrecoverable device error.\n");  
    delete pDeviceException;  
    bOK = false;  
}
```

Code Sample 17. Example: *SimErrException*

In a random test, approximately once every 1000 times, *SimErrException* will throw a database exception. In a systematic test, *SimErrException* will throw a database exception the first time it's called.



More Free Software

C++ Notifiers

C++ notifiers are a simple but powerful mechanism for sending messages between C++ objects. Inter-object messaging is a technique used extensively in Java and Smalltalk, but it's not supported by the standard C++ language or libraries.

At the On Target website, I've posted documentation, source code, and a threadsafe Win32 DLL for C++ notifiers.

If you're curious enough to find out more, check out the website. I wouldn't design a major interactive application in C++ without this tool. I've recommended it to several major clients and each time I've gotten an enthusiastic response.

Debug message logging

Coming soon to the On Target Software website – the humble *printf* with a new suit of clothes. Too many times I've found myself working on problems where a debugger is useless. Interactive software that depends on change of focus can't be fixed with a debugger. Problems in the release configuration (that aren't reproducible in the debug configuration) can't be touched with a debugger. For those times, and as a supplement when working on the real tough bugs, I need to trace program functions.

For this reason, I've developed a set of debug functions that work like *printf* but write to a circular log file. They are conditionally compiled, of course, so you can remove them from the release version. But you can also leave them in the release version – for debugging, early alpha testing, etc.

It's another must-have tool. There's no reason for everyone to reinvent it.

Why all this free software?

I'm a software contractor and my products are the prototypes, designs, and software that contribute to my clients' commercial applications. The work that would best advertise my skills is the confidential property of my clients. I wouldn't have it any other way, but that means I have no visible portfolio.

So I decided to take some of my ideas, those few that are general purpose, neither confidential nor proprietary, and deliver them as software components or utilities through my website.

I hope you find the software useful. If you think it is and you think I could make a more direct contribution to your C++/Windows project, please call me at (781) 834-6542. I work with companies throughout North America.

Thanks,
Dave




How to reach On Target Software



[dave.pomerantz](mailto:dave.pomerantz@targetsoft.com)

56 bartlett's island way
marshfield, ma 02050

781/834 6542 phone

781/834 6416 fax

<http://www.targetsoft.com>

dave@targetsoft.com

Revision History

1.0 Monday, November 17, 1997

First version of document. This is associated with version 1.0 of the software.