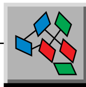


GETTING STARTED WITH VBA

Chop Down Errors with the Logger Class

Create your own generic class to log events and error conditions in all your applications.

by Chris Barlow

When you write a new application, do you try to anticipate potential error conditions and make sure that your application handles them gracefully? For example, do you write the code so that the application can find a data file that the user moved from its previous location? What does your application do if the user forgets to insert the diskette or doesn't connect to the network server? Does your application keep track of the actions taken by multiple users?

If the answers to most of these questions is "No," don't feel that you're alone. Very few programmers (yes, including me!) write code that handles error conditions properly and completely. With the time constraints most of us operate under, it is too convenient to make gross assumptions about the state of the environment and the actions of the user and just write "straight-line" code that assumes the application can get from point A to point B without any detours.

How many of you have written and tested a neat application, then given it to the user only to watch it crash and burn? As you walk back to your desk pulling out your hair, you mutter, "But no one would enter that kind of data into that field and then click on that button!"

We all know the academic solution that we learned back in computer science class. I used to tell this to my own Computer Science 101 classes: design your application assuming that anything that can go wrong *will* go wrong. As you write each line of code, imagine what branch of code might execute here, no matter how unlikely, and add more code to handle that branch. But few of us practice what we preach. The pressures to complete the application are just too great. Regardless, we need to create a robust application that will work for a wide range of users, from beginners to advanced users. What can we do?

At SunOpTech we take a middle-of-the-road approach. We know we can't afford to take the time to write code to handle every possible error. So, like a M.A.S.H. unit, we triage the potential errors. Some errors are "code red" errors—either so rare or so bad that we're not going to spend any time trying to handle them ourselves. We'll just let Windows or Visual Basic handle these errors with their own default-error processing. For example, if the user removes the diskette while the application reads from it, the operating system presents an appropriate message.

On the other hand, some errors are "take two aspirin" errors—so common and easy to handle that we just automatically write the code to deal with them. For example, if you use the Kill statement to delete a file that doesn't exist, you get an error. When you write this code, you should either use the Dir statement to check in advance whether the file really exists, or trap the potential error with an On Error statement. Even better, you can create your own Kill function that does this kind of checking, and call your own function rather than just calling the Kill statement:

```
Function KillFile(sFile$) As Boolean
If Len(Dir(sFile)) Then
   Kill sFile
   KillFile = True
End If
End Function
```

CONTINUOUS IMPROVEMENT

However, many errors do not fit easily into either the "code red" category or the "take two aspirin" category. How should you handle these errors? Sadly, most programmers simply pop up a message box saying an error occurred and then terminate the program. This just annoys and frustrates the user. The first few times it happens, the user may report the problem, although the user usually cannot remember what he or she was doing before the error occurred. After a few times, the user just clicks on the OK or Retry button and lives with the problem—and learns to hate the program and the developer!

The even more damaging result of this "solution" is that the programmer gets no feedback. Because there is no built-in feedback loop to let the programmer know about these error conditions and their causes, it is impossible to make continuous improvements to the program so that each new version handles more of these common errors. The philosophy of Continuous Improvement has

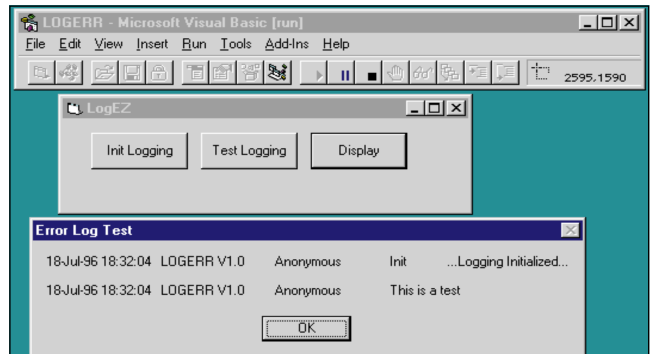
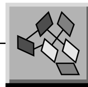


FIGURE 1 ***The Basic Test Form.*** *When you develop a class, you need to test it. I try to start with a simple form and develop it as I develop the class, to test each method.*

Chris Barlow is president and CEO of SunOpTech, a developer of manufacturing decision-support applications, including the ObjectBank and the ObjectJob Systems, where he and Ken Henderson hold a software patent related to decentralized distributed asynchronous object-oriented systems. Chris holds degrees from Harvard Business School and Dartmouth College, where he worked with Drs. Kemeny and Kurtz on the Basic language. Reach Chris on the Internet at ChrisB@SunOpTech.com or through SunOpTech's World Wide Web server at www.SunOpTech.com.



been crucial to the success of many modern enterprises, but generally ignored in software development.

The critical first step in continuous improvement is to make sure that you know where the error occurred in your application, and that you can trace how the user got to that point. This means your program must be able to log its actions and any errors that occur, in a form that is flexible and transparent to the user. Even better, if you create an adjustable level of logging based on the severity of the error, you can fine-tune the logging to your needs. That way you can have intensive logging during development and initial testing of your application, and less logging as the application matures.

A complete yet flexible logging system allows you to analyze your application in operation, identify the common errors, and develop a solution that improves your application and reduces the “hassle factor” for the user. When Stan Schultes joined SunOpTech, he developed a Debug module that SunOpTech uses in all its VB3 applications to control the error logging process. I’ll re-create this module in this month’s column.

Stan designed the module to accomplish several important goals. For example, the logging class allows you to easily create a record of your application’s actions and errors through a series of log entries. Each log entry contains standard information about the current time, program, version, and user, and it’s easy to add a variable amount of other data, depending on the event. He also designed the logging notification method to be flexible. It logs certain conditions quietly, while for others it lets you notify the users. It allows you to adjust the logging severity level based on the maturity of the application and the experience of the user. It also allows you to globally turn off certain logging methods.

DESIGNING A LOGGING CLASS

Let’s begin by defining the properties and methods for the cLogErr class. Start a new VB4 project and insert a class module. View the class module properties by pressing F4, and change the name to cLogErr.

You need to define a property for the log file name and other properties to hold the general information that you want to appear in every logging entry:

```
Public LogFile As String
'Path and name for log file
Public System As String
'System name to log
Public Program As String
'Program name to log
Public Version As String
'Program version to log
Public User As String
'User name to log
```

Then define properties to control the severity of log entries to process, whether to overwrite or append to the log file, and the global properties to control the three logging methods I’ll describe. The log file should allow you to use any combination of these methods for any individual logging event. The first method writes the log entry to an ASCII file, and the second method displays the log entry in the debug window. The second method is useful only when running in design mode. The third method pops up a message box showing the log entry. You can easily extend the class to support other methods such as e-mail or Internet notification:

```
Public Append As Boolean
' whether to erase log file
Public Severity As Integer
' only log entries < this
Public DoFile As Boolean
' show in log file
Public DoMsg As Boolean
' show in message box
Public DoDebug As Boolean
' show in 'debug window
```

You want to be able to use this class with a minimum of setup work, so put some defaults in the Class_Initialize event

Using All the Bits in an Integer

We use so many new things in Visual Basic that we tend to forget some of the old programming techniques. One of the methods I still use often is to pack a lot of information into a function’s argument by using the individual binary bits of an integer parameter. Because an integer has 16 binary bits, you can actually pass 16 true/false or yes/no values within a single integer.

For example, in the logging class you can specify which logging method you want to use for an individual log entry by calling the LogIt method with separate arguments for each logging method:

```
MyLog.LogIt DoFile:=True, _
   DoDebug:=True, DoMsg:=True, _
   "Invalid File Error"
```

But this makes it difficult to add new logging methods because to add one you need to add a new argument to the function call. A more flexible method combines the logging method into a single integer value in much the same way the Visual Basic MsgBox function operates.

If you want to display a message box with an OK button, a Cancel button, and an exclamation point, you call the Visual Basic MsgBox function with the buttons argument equal to two constants added together: vbOKCancel + vbExclamation. This actually passes a binary 11001 to the function. The function uses these bits to determine which options have been selected.

You can use this same technique to provide flexible options to your logging class. If you pass a LogMethod argument with each logging event, you can use bits 5, 6, and 7 to control logging to a file, debug window, and message box. You can use bits 1-4 to indicate the severity level of this event or error. For example, if you set up some constants to mask the appropriate bits:

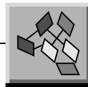
```
Const FileMask = &H10
'mask all but bit 5
Const DebugMask = &H20
'mask all but bit 6
Const MsgMask = &H40
'mask all but bit 7
Const LevelMask = &HF
'mask bits 5-8
```

you can write a function that returns a LogMethod argument by combining several arguments into a single integer:

```
Public Function
MakeLogMethod(Severity _
   As Byte, bFile As Boolean, bDebug _
   As Boolean, bMsg As Boolean) As _
   Integer
Dim iLogMethod%
iLogMethod = Severity
If bFile Then iLogMethod = _
   iLogMethod + FileMask
If bDebug Then iLogMethod = _
   iLogMethod + DebugMask
If bMsg Then iLogMethod = _
   iLogMethod + MsgMask
MakeLogMethod = iLogMethod
End Function
```

This function gives the application developer the alternative of setting the bit-oriented integer argument directly or using this method with explicit arguments.—*C.B.*

 http://www.windx.com



to set these properties to default values. You can make use of the version information and the App object to set these defaults:

```
Private Sub Class_Initialize()
Program = App.EXEName
LogFile = App.Path & "\" & Program & ".log"
System = Program
Version = App.Major & "." & App.Minor _
   & "." & App.Revision
User = "Anonymous"
Append = True
DoFile = True
DoMsg = True
DoDebug = True
Severity = 8
End Sub
```

The first method your class will support is the Init method to initialize the logging. In most cases, you call this method at the beginning of the application program to change from the default properties. For example, you can use the Init method to set the actual user name in the User property of the class. Design this method with arguments using the Optional keyword so that you need to specify only those arguments that are different from the default. Then you can use the IsMissing function to change only the passed properties:

```
Public Sub Init(Optional sProg, Optional sLogFile, _
   Optional sSYS, Optional sVer, Optional sUser, _
   Optional bAppend, Optional bFile, Optional bMsg, _
   Optional bDebug, Optional iSeverity)
'Set missing parameters to defaults
If Not IsMissing(sProg) Then Program = sProg
If Not IsMissing(sLogFile) Then LogFile = sLogFile
If Not IsMissing(sSYS) Then System = sSYS
If Not IsMissing(sVer) Then Version = sVer
If Not IsMissing(sUser) Then User = sUser
If Not IsMissing(bAppend) Then Append = bAppend
If Not IsMissing(bFile) Then DoFile = bFile
If Not IsMissing(bMsg) Then DoMsg = bMsg
If Not IsMissing(bDebug) Then DoDebug = bDebug
If Not IsMissing(iSeverity) Then Severity = iSeverity

If DoFile Then LogIt FileMask, _
   "Init", "...Logging Initialized..."
End Sub
```

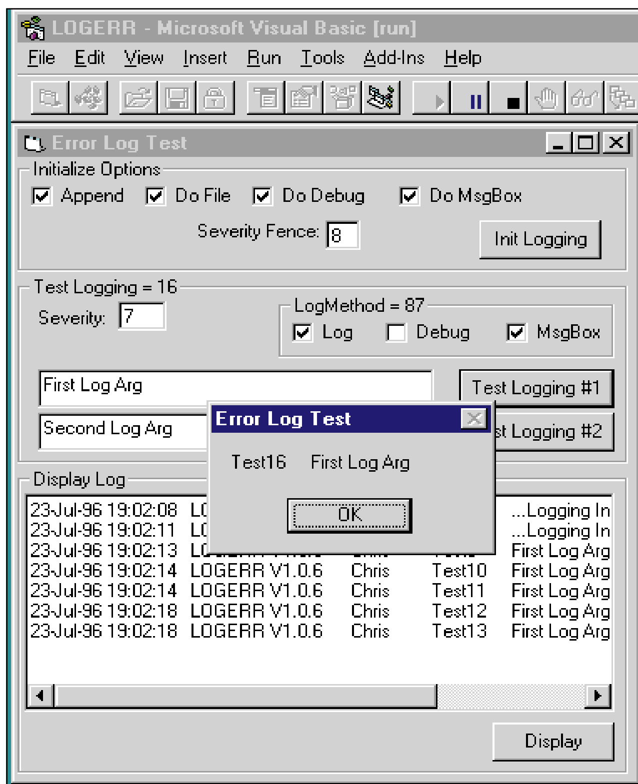


FIGURE 2 ***The Complete Test Form.*** *When the class nears completion, I create a more comprehensive test form to really put the class through its paces.*

This allows the application to call this method and specify only the necessary parameter:

```
Private Sub butInit_Click()
MyLog.Init sUser:="Chris"
End Sub
```

THE LOGIT METHOD

The second method, LogIt, does the actual logging. Many standard logging modules fail in this area because they restrict the format of the log entry to a specific number of fields. Because most log files store data in a delimited format, most standard logging modules either specify a fixed set of parameters or they require the application to preformat the log entry with delimiters.

In VB4, you can design a flexible method that allows the application to use your class with any number of arguments. When you declare a function using VB4's new ParamArray keyword with the last parameter, you can pass an arbitrary number of arguments. The function retrieves an argument as an array of variant elements:

```
Public Sub LogIt(iLogMethod%, ParamArray Args())
```

The first argument for the LogIt method is the LogMethod. This argument combines a severity level and a log method into a single integer that the function can interpret (see the accompanying sidebar, "Using All the Bits in an Integer").

The next step in the procedure is to dimension two local variables: LogLine, to contain the standard information for the log entry, and ArgLine, to tab delimit the arbitrary number of arguments. Here you can check whether this log entry exceeds the current Severity property in order to control the level of logging for this event. If the first byte of the LogMethod is greater than the Severity, you can exit the procedure:

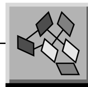
```
Dim LogLine As String
Dim ArgLine As String
Dim i As Integer
If (iLogMethod And LevelMask) > Severity Then Exit Sub
```

You create the LogLine string by concatenating the current date and time to the Program, Version, and User properties of the class:

```
LogLine = Format(Now, "dd-mmm-yy hh:mm:ss")
LogLine = LogLine & vbTab & Program
LogLine = LogLine & vbTab & "V" & Version
LogLine = LogLine & vbTab & User
```

You create the ArgLine string by stepping through each element in the ParamArray:

```
For i = LBound(Args) To UBound(Args)
```




```
   ArgLine = ArgLine & Args(i) & vbTab
Next
```

Finally, when you have set the proper bit for a logging method, call the appropriate procedure to complete the log entry:

```
If iLogMethod And FileMask Then LogToFile LogLine, ArgLine
If iLogMethod And DebugMask Then LogToDebug ArgLine
If iLogMethod And MsgMask Then LogToMsgBox ArgLine
End Sub
```

The actual logging methods are quite simple. Just check the global property to ensure that you have enabled this method, then call the standard VB function:

```
Public Sub LogToFile(LogLine$, ArgLine$)
Dim FileNum%
If DoFile Then
   If Not Append Then KillFile LogFile
   FileNum = FreeFile
   Open LogFile For Append Shared As #FileNum
   Print #FileNum, LogLine & vbTab & ArgLine
   Close #FileNum
End If
End Sub

Public Sub LogToMsgBox(sMsg$)
If DoMsg Then MsgBox sMsg
End Sub

Public Sub LogToDebug(sMsg$)
If DoDebug Then Debug.Print sMsg
End Sub
```

In the complete source code for the cLogErr class module, I added a Display method that returns the contents of the log file. You can get this code listing on the Registered Level of the Development Exchange (for details, see the Code Online box at the end of this article).

TESTING THE CLASS

When you develop a class, you need to test it. I try to start with a simple form and develop it as I develop the class, to test each method (see Figure 1). When the class nears completion, I create a more comprehensive test form to really put the class through its paces (see Figure 2). You can get the code for the more comprehensive test application from the Premier Club on The Development Exchange Web site (for more information, see the Code Online box at the end of this column). Here, I'll cover how to create the simple form.

First, add the code to the form to create an instance of the cLogErr class:

```
Option Explicit
Dim MyLog As New cLogErr
```

Then place this code in the Init button Click event to initialize the class and change the default User property:

```
Private Sub butInit_Click()
MyLog.Init sUser:="Chris"
End Sub
```

Then put this code in the Test button Click event to call the LogIt method so that the text will be written to the log file:

```
Private Sub butTest_Click()
MyLog.LogIt 16, "This is a test"
End Sub
```

Finally, code the message box statement to display the log of the Display button Click event:

```
Private Sub butDisplay_Click()
MsgBox MyLog.Display
End Sub
```

With this flexible logging class, you can write your application code to easily provide the feedback required to implement Continuous Improvement. If you belong to the Premier Level of The Development Exchange, be sure to get the comprehensive test app code. ☒

Code Online

For all the current issue's listings in one file, go to the Registered Level of The Development Exchange (http://www.windx.com), The Microsoft Network (GO WINDX), or CompuServe VBPJ *Forum's Magazine Library (GO VBPJ). Development Exchange Premier Level subscribers ($20 for six months) can get each article's listings in a separate file, as well as additional code and utilities for selected articles, plus archives of all code ever published in* VBPJ *and* Microsoft Interactive Developer *magazines.*

Chop Down Errors with the Logger Class

Locator+ Codes

Listings ZIP file (free Registered Level): VBPJ1096

Listings for this article plus both the simple and comprehensive test applications, as well as the Logger class module (subscriber Premier Level): GS1096P