Get a Handle on OLE Errors

Click & Retrieve Source CODE!

Develop applications that take advantage of the underlying error-handling mechanism in VB4.

by Keith Pleas

In this column, I'm going to cover VB error handling, OLE Automation error handling, and the new Win32 method for error handling. Fortunately for me (and you), they're all the *same thing*. Because everything builds on what's available in the operating system, I'll start there.

At one level, you can view Windows as simply a library of functions. As you know, functions always return something. If you reach back in your memory to when you were calling 16-bit Windows APIs you will recall that, in general, they returned a number that indicated the success or failure of the call. Each call had a limited number of reasons for failing. There was virtually no correspondence of error values among APIs, and you probably wrote a lot of Select statements based on the return value for each call.

With the Win32 API, Microsoft changed this dramatically. A call that failed for some reason returned zero. A complete list of errors existed for almost all Windows APIs, and each thread had an error value that could be retrieved using the GetLastError function.

In VB4, because API calls are embedded in p-code instead of natively compiled code, any number of calls might be made to Windows between one statement and the next. When you call GetLastError in your code after you call an API, you must guarantee the corresponding error will return. To get around this, Microsoft cached this value in the LastDLLError property to the Err object. It contains the error code for the last API call that the programmer explicitly made in his or her code, and *not* calls that VB itself made. Once you retrieve this value using the Err.LastDLLError property, you must look up that value in the Windows header file WINERROR.H. For example, if the error value were 1005, searching for "1005L" would yield this:

```
// MessageId: ERROR_UNRECOGNIZED_VOLUME
//
// MessageText:
//
//  The volume does not contain a
//  recognized file system.
//  Please make sure that all required
//  file system drivers are loaded and
//  that the volume is not corrupt.
//
#define ERROR_UNRECOGNIZED_VOLUME _1005L
```

Interestingly, all of these messages are stored in your Windows system files. Because it's somewhat inconvenient to open and search the header file, plus many of you may not have it, I created a little VB app that calls the FormatMessage Windows API to retrieve the string and display it (see Figure 1). I've included both the declarations and all the code for this utility (see Listing 1).

WIN32 RICH ERRORS

Although the Win32 error-handling method cleaned up the Windows API dramatically while maintaining a high degree of portability from Win16 to Win32, it didn't meet the needs of the OLE developers looking farther down the road. To address those needs, Microsoft created an enhanced error-handling system that used new error objects and error codes with multiple fields. Of course, it isn't just the low-level OLE internals that use this mechanism. Everything from the new Windows shell, the Microsoft Internet services, MAPI, RPC, and even—gasp!—VB itself are all related to OLE and use this error information system.

OLE maintains one error object per thread. An error object is composed of an exception information structure plus a pointer to the interface that raised the exception. The structure has this C prototype:

```
typedef struct tagEXCEPINFO {
   WORD wCode;
   WORD wReserved;
   BSTR bstrSource;
   BSTR bstrDescription;
   BSTR bstrHelpFile;
   DWORD dwHelpContext;
   void FAR * pvReserved;
   HRESULT (STDAPICALLTYPE FAR* pfnDeferredFillIn)
         (struct tagEXCEPINFO FAR*)
```

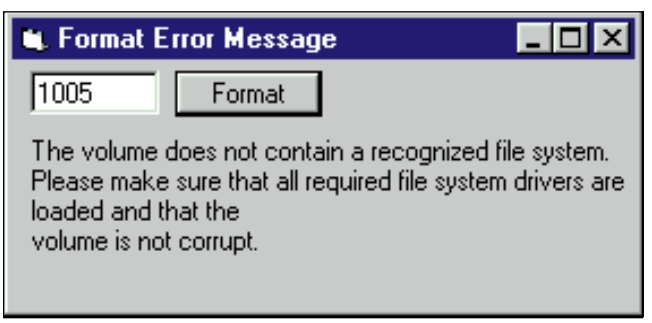


FIGURE 1 ***Accessing System Error Strings.*** *Using the Format Error Message utility described in Listing 1, you can retrieve error messages and display them in a window.*

Keith Pleas is an independent developer, author, and trainer. He is the author of the forthcoming book, Visual Basic Tips & Tricks, *from Addison-Wesley. Reach Keith on CompuServe at 71333,3014 (from the Internet: 71333.3014@compuserve.com).*

```
    SCODE scode;
}
```

Notice how close the structure's C prototype corresponds to the properties of VB4's Err object (see Table 1).

The return value of functions that use this new error-handling method is an HRESULT. Confusingly, an HRESULT isn't a handle to a result. In fact, it isn't a handle to anything. An HRESULT is a 32-bit value with several fields (see Figure 2).

Because an HRESULT is a 32-bit value, VB4 represents it using a Long. VB4 essentially dictates to you the high 16 bits. Because VB4 is built on OLE, you must use the corresponding facility code for OLE Automation and therefore all of your error codes must be greater than &H80040000. Of course, that leaves the lower 16 bits to play with—anywhere from 512 to 65535 (errors from &H0 to &H200 are reserved for use by OLE's Common Object Model, or COM).

To make it a little easier for us lazy VB developers, Microsoft added a constant, vbObjectError, to VB's type library. This constant is mentioned throughout the VB4 documentation. You must add it to internal error numbers in the less-than-64K range if they're to be passed to external applications.

The VB4 documentation recommends using error values less than 64K and adding the vbObjectError constant whenever they are to be passed externally. Frankly, I think it's a poor solution to maintain two sets of the same code, one for internal use and one for external use. Instead, I suggest you use one set of error codes that already includes the error constant.

In historical documentation, 16-bit OLE distinguishes between an SCODE, which is an error code value, and an HRESULT, which is a function and method return type. When Microsoft designed 32-bit OLE error codes, it decided to remove contextual information that would also have to be propagated through RPC. In 32-bit OLE, an SCODE and an HRESULT are the same thing and HRESULT is the preferred terminology, although the Win32 documentation is still littered with references to SCODEs.

WHAT ABOUT VB?

So, what does this have to do with VB? Lots. Every subroutine you execute in your VB4 class modules actually returns an HRESULT. What you thought your method/function returned is actually packaged in with the parameters: VB examines the HRESULTs before doing anything with them. While this is, for the most part, abstracted from VB developers, it is also the same mechanism that VB developers use for communicating error information between objects.

VB developers should already be familiar with how VB handles errors in forms and code modules. Since the first version of VB, any error in a subroutine could either be handled or passed back up the call stack. If the error bubbles up to the top of the call stack without being handled, VB displays an error dialog and terminates the app—without performing any cleanup code. Generally, this is considered suboptimal behavior. Therefore, unhandled exceptions in form and control events are considered anathema to VB developers.

The error-handling goal in VB classes is somewhat the opposite of forms and code modules. Object servers pass error information back to requesting applications by deliberately raising untrapped errors. EXEs that act as both object components and servers must use a combination of these techniques.

It's also worth pointing out that there are other possible methods for handling errors. For example, VB4's Remote Data Objects (RDO) uses an rdoErrors collection, in addition to VB trappable errors, to communicate ODBC error and information messages. Jet's Data Access Objects (DAO) uses a similar Errors collection.

VB4

```
   Alias "FormatMessageA" (ByVal dwFlags As Long, _
   lpSource As Long, ByVal dwMessageId As Long, _
   ByVal dwLanguageId As Long, ByVal lpBuffer _
   As String, ByVal nSize As Long, _
   Arguments As Long) As Long
Public Const FORMAT_MESSAGE_FROM_SYSTEM = &H1000
Public Const FORMAT_MESSAGE_IGNORE_INSERTS = &H200
Private Sub cmdFormat_Click()
  Dim sBuff As String
  Dim lMsgId As Long
  sBuff = String(256, " ")
  lMsgId = CLng(Val(txtID))
  ret& = FormatMessage(FORMAT_MESSAGE_FROM_SYSTEM Or _
     FORMAT_MESSAGE_IGNORE_INSERTS, _
     0&, lMsgId, 0&, sBuff, Len(sBuff), 0&)
  lblMessage = sBuff
End Sub
```

LISTING 1 ***Display Error Value Messages.*** *Instead of opening and searching the header file for error value messages, you can use this little VB app to call the FormatMessage Windows API to display a description of the string in question.*

Property	Value
Number	(Default property) Any valid error number.
Source	Object server name; default for VB4 is ProjectName.
Description	Description of error; default for VB4 is "OLE Automation Error."
HelpFile	Fully qualified file name of the Windows Help file.
HelpContext	Help context ID.
LastDLLError	(Win32) Error code from the last call to a DLL.

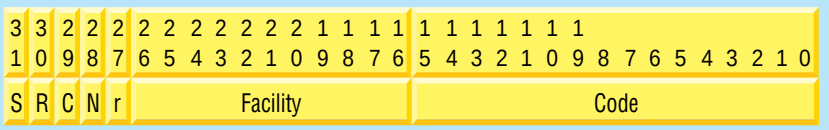
TABLE 1 ***Defining Err Object Properties.*** *When handling errors, you must determine the cause of the error, and how to deal with it. You must also raise errors back in controlling applications. Begin by setting the Err object's properties, shown in this table, and then execute a Raise method to initiate the process.*

You could also implement a system similar to that used by most of the old-style Windows APIs where the value returned by the method can be an error code. In this case, the client application must examine the return value in line. Also, if you choose this method but want to use an actual Err object, you must use a ByRef parameter for the return value, which would normally contain an actual value. In general, you'll want to raise an error in your object server, pass it back to the client, and implement an error handler in the client to trap that error.

Now that you have an understanding of the underlying error-handling mechanism in VB4 classes, I'll focus on developing server applications to take advantage of the mechanism. Assuming your program is acting as an intermediate object server in an application—meaning the program uses objects and acts as an object server to another controller—you should first consider the types of errors you might want to trap. These include errors from object servers you handle yourself, errors from object servers you modify and pass on, internal runtime errors you handle yourself, internal runtime errors you modify and pass on, internal raised errors you handle yourself, and internal raised errors you modify and pass on.

Obviously, potentially thousands of errors might be thrown. While it isn't feasible to trap each of them individually, you shouldn't let any error go untrapped. This doesn't necessarily require elaborate code. You can use a default case for all but the most common or critical errors.

CONTINUED ON PAGE 120.



S: Severity - indicates success (0) / fail (1).

R: (reserved) corresponds to NT's second severity bit.

C: (reserved) corresponds to NT's C field.

N: (reserved) indicates a mapped NT status value.

r: (reserved) for internal use.

Facility: the facility code.

Code: the facility's status code.

FIGURE 2 ***An HRESULT Breakdown.*** *The Structure of an HRESULT contains an error code, a system-defined facility code, and several reserved flags.*

VB4

```
Private Const errFetchOne = vbObjectError + 2001
Private Const errFetchTwo = vbObjectError + 2002
Private Const errFetchDefault = vbObjectError + 2000
Private Const errFetchExternal = vbObjectError + 2999

Public Sub Fetch()
  On Error GoTo ErrHandler
  '...Code goes here
  Exit Sub

ErrHandler:
  If Err.Source <> DefaultErrSource Then
     'External error
     Err.Raise Number:=errFetchExternal, Source:=DefaultErrSource _
        & "[" &Err.Source & "]"
  Else
  Select Case Err.Number
  Case errFetchOne
     Err.Raise Number:=errFetchOne, Description:=LoadResString_
        (errFetchOne)
  'Case errFetchTwo...
  Case Else
     Err.Raise Number:=errFetchDefault, _
     Description:=LoadResString(errFetchDefault)
  End Select
  End If
End Sub
```

LISTING 2 ***Generic Error Handler Template.*** *This error handler template has areas for handling both internal and external errors, and it formats the Source property when debugging.*

CONTINUED FROM PAGE 117.

One caveat you may have already heard is that error trapping has an impact on application performance and should only be used in select situations. To test this idea, I experimented with classes with On Error Resume Next, with On Error Goto Label, and with no error handling. In a fairly simple case with nested loops of subroutine calls, error trapping slowed execution only about 10 percent. Even if you know the objects you're using won't ever produce an error (yeah, right), you should implement some form of checking because any failure in the underlying OLE communications layer will also trigger an error. The VB documentation also suggests writing a last-resort error handler that shuts down your application, but that technique is inappropriate for OLE servers. Never shut down your server: that's up to the controller.

The process of handling errors is all about figuring out what caused the error, what to do about it, and, most interesting in this case, raising errors back in controlling applications. The process of raising the error itself is pretty simple. You set the properties of the Err object and then execute a Raise method. If you're not familiar with this, you might want to review the book titled, *Creating OLE Servers*, found in the *Professional Features* manual that comes with the Pro and Enterprise

Editions of VB4. See the section called "Generating and Handling Errors," on page 59 in Chapter 2. The book includes a fairly good introduction to the process, although it has a few problems.

For example, VB could have never compiled this line of code from the section titled, "Errors from Another OLE Server":

```
Case Is > (vbObjectError + 512) And _
   Is < (vbObjectError + 65536)
```

The first property that you have to deal with is the Source. Although the VB docs say that the default Source for your error will be the ProjectName combined with the Name of the class module, it's actually just the ProjectName. Also, while it's not mentioned anywhere, the default Description is "OLE Automation error." What's really ugly is that at run time, there's no way to programmatically get at the ProjectName. It is available in the design environment using VB's extensibility model and the Application object, but this would require a fair amount of additional code and is only a design-time solution.

Fortunately, this little routine, from *Object Programming with Visual Basic 4* (by Joel Dehlin and Matt Curland, available soon from Microsoft Press), illustrates how to trigger an internal error and archive off the Source property in a Static variable:

```
Public Function DefaultErrSource() _
   As String
   Static strSource As String
   If Len(strSource) Then
      DefaultErrSource = strSource
      Exit Function
   End If

   On Error GoTo ErrorTrap
   Err.Raise 0

ErrorTrap:
   strSource = Err.Source
   DefaultErrSource = strSource

End Function
```

If your object server calls this code in Sub Main, you'll end up with an easy way of telling whether or not an error was generated by your application by simply comparing DefaultErrSource to the Err.Source property. This is important because you need to encapsulate errors returned from servers you're using because whoever's using your server might have no clue what objects you're calling. Because they know only that they're talking to you, it's up to you to provide your error back to them.

Also, because you didn't generate the Source and Description properties in errors returned from servers, you have no control over their formatting, which might include changing information, such as version numbers. Rely on error numbers only, and use the text itself to display a message to an end user. Of course, your object server should never pop up a message. It's up to you to return this information back to the controller.

Although it may break encapsulation somewhat, consider appending the error message to the beginning of the Source property as you pass it along. Doing so would be appropriate mainly for debugging purposes because a multi-object situation could generate a long source string. If you do try this, wrap it with conditional compilation and use a Debug/Release flag.

While you don't have much control over the events you received from objects you're using (unless you wrote them too, of course), you do have quite a bit of freedom with your own objects. The most common advice is to be consistent. I have yet to see any coding standards for class error trapping, but I expect this is already a hot topic for developers implementing object applications. For runtime errors generated within your application, I suggest you use VB's errors and error messages where possible. The one exception to this is if you choose to implement somewhat friendlier error messages.

When debugging object servers, take advantage of the Error Trapping option on the Advanced tab of the Options dialog. The documentation is a bit cryptic. The possible settings are: Break on All Errors, including handled errors; Break in Class Module (Default), which is a Raise method or error with no error handling (it's confusing, because it might be handled further up the chain); and Break on Unhandled Errors (meaning never in the OLE server).

I'll leave you with a sample generic error-handling template (see Listing 2). This code has areas for handling both internal and external errors and implements my suggestion for formatting the Source property when debugging. Note that setting the Err.Number is required in the errFetchOne case even though it's already set to that value. Number is a required argument of the Raise method. ×

