Date: 08/08/2012
Procedure: Visual Basic - Error Handling
Source: LINK
Permalink: LINK
Created by: HeelpBook Staff
Document Version: 1.0

VISUAL BASIC – ERROR HANDLING

Despite your best efforts to cover all possible contingencies, run-time errors will occur in your applications. You can and should do all you can to prevent them, but when they happen you have to handle them.

Introduction

The various functions, statements, properties and methods available in **Visual Basic** and the components used in **Visual Basic** expect to deal with certain types of data and behavior in your applications. For example, the **CDate()** function can convert a value to a **Date** variable. The function is remarkably flexible in the type of information it can accept, but it expects to receive data that it can use to derive a date. If you provide input that it can't convert, it raises error number **13** – "**Type mismatch**" – essentially saying "*I can't handle this input data*".

In an application, this type of error may be a program logic error (you simply passed the wrong data) or it may be a data entry error on the part of the user (you asked for a date and the user typed a name). In the first case, you need to debug the program to fix the mistake. However, there is no way for you to anticipate the behavior of the end users of the application. If the user enters data you can't handle, you need to deal with the situation.
Dealing with errors at run-time is a two step process:

- Trap the Error: Before you can deal with an error, you need to know about it. You use VB's**On Error** statement to setup an error trap.
- Handle the Error: Code in your error handler may correct an error, ignore it, inform the user of the problem, or deal with it in some other way. You can examine the properties of the Err object to determine the nature of the error. Once the error has been dealt with, you use the Resume statement to return control to the regular flow of the code in the application.

In addition to dealing with run-time errors, you may at times want to generate them. This is often done in class modules built as components of **ActiveX** server **DLLs** or **EXEs**. It is considered good programming practice to separate the user interface from the program logicas much as possible, so if a server component cannot deal with an error, it should raise the error in its client application rather than simply display an error message for the user.

In VB5, there is an option that allows you to specify that an application has been designed for unattended execution (this is typically used for remote server applications). If you plan to allow the application to run unattended or on a remote computer, you can't simply display an error message because there will be nobody there to see it or dismiss the message box.

Trapping Errors at Run-Time

Before you can do anything to deal with a run-time error, you need to capture the error. You use the **On Error** statement to enable an error trap. On Error will redirect the execution in the event of a run-time error. There are several forms of the **On Error** statement:

- On Error Goto label: This form of the **On Error** statement redirects program execution to the line label specified. When you use this form of **On Error**, a block of error handling code is constructed following the label.

- On Error Resume Next: This form of the **On Error** statement tells VB to continue with the line of code following the line where the error occurred. With this type of error trap, you would normally test for an error at selected points in the program code where you anticipate that an error may occur.

- On Error Goto 0: **On Error Goto 0** disables any error handler within the current procedure. A run-time error that occurs when no error handler is enabled or after an **On Error Goto 0** is encountered will be handled using VB's default error handling logic.

It's not necessary to code an error handling routine in every procedure you write in **Visual Basic**. If an error is raised in a procedure, VB will work its way back up through the call tree looking for an error handler.

```
Public Sub SubA()

On Error Goto ProcError

' other code

MsgBox FuncA()

ProcExit:

Exit Sub

ProcError:

MsgBox Err.Description

Resume ProcExit

End Sub

Private Function FuncA() As Date

FuncA = CDate("hi there")

End Function
```

In this example, procedure **SubA** enables an error handler using the statement On Error Goto ProcError. When function FuncA is called in the MsgBox statement, the On Error Goto ProcError handler is still enabled.
The **CDate** function in **FuncA** will generate error 13 (type mismatch) because **CDate** can't make a date from the input data. VB first looks in FuncA for an error handler. None was enabled, so the error is propogated back up the call tree to **SubA**.

Since there is an error handler in **SubA**, program execution is redirected to the ProcError label in SubA. The MsgBox statement displays a description of the error and the Resume statement directs **VB** to continue execution at the **ProcExit** label.
There are some situations where **VB** cannot pass an error back up the call tree. This applies to Sub Main, most event procedures, and the **Class_Terminate** event procedure. **Sub Main** (if defined in the project property sheet) is the first code executed, so there is no procedure higher in the tree at application startup time. Most event procedures are also fired by **Visual Basic** when no other code is running so these are also at the top of the tree.

Finally, the **Class_Terminate** event of class modules cannot raise an error because this event can also occur when no other code is executing in the application.

If an error is generated in one of these types of procedures and no error handler is enabled in the procedure, VB invokes its own default error handler, displays a message, and terminates the application. Because of this behavior, it is vital that you always code an error handler in Sub Main, all event procedures, and the **Class_Terminate** event for class modules.

Unlike the Class_Terminate event, the Class_Initialize event of a class module can raise an error or allow it to go untrapped. However, it is considered good programming practice to have classes trap their own errors, deal with them if possible, and if necessary raise errors explicitly, providing a number and description defined within the class.

You can code your classes to map any error the class encounters to class-defined error numbers, but given the large number of potential errors that could occur in an application, that may not always be practical. An alternative is to have the class assign specific numbers and descriptions to errors that are specific to problems with the code or data in the class (such as a value that violates a rule for the data) but pass out standard VB errors (such as error 13 – type mismatch) as is. This allows applications using the class to explicitly handle the errors exclusive to the class with customized code, but handle standard VB errors with more generic code.

Regardless of the approach you take, you must always ensure that private data within the class is valid and that code within the class cleans up any local or module level object variables it creates. This may require you to setup an error handler that traps errors, cleans up local object variables, and then raises the same error again.

Building Error Handlers

Trapping an error using the **On Error** statement is only the first step in dealing with run-time errors in your code. You must also deal with the error in some way, even if the error handling code is as simple as ignoring the error (a perfectly valid approach in some situations) or displaying a message for the user.

The first step in handling an error is determining the nature of the error. This is accomplished by examining the properties of **Visual Basic**'s **Err** object. The **Err** object includes the following properties:

- Number: This is the error number that was raised.

- Description: This contains a descriptive message about the error. Depending on the error, the description may or may not be useful. (**Microsoft Access**, for example, has the the infamous error message "There is no message for this error.")
- Source: The Source property tells you what object generated the error.
- HelpContext: If a help file has been defined for the component that raised the error, this property will give you the help context ID. You can use this property along with the **HelpFile** property to display context sensitive help for errors in your application or as a debugging aid.
- HelpFile: This is the name of the help file that contains additional information about the error (if a help file has been provided).

It is important that you rely only on the error number to determine the nature of the error. While the **Description** and other properties may contain useful information, only the Number property is a reliable indicator of the exact error that occurred.
A common approach in coding an error handler is to build a **Select Case** block based on the **Number** property of the **Err** object:

```
Public Sub SubA()

On Error Goto ProcError

' code that raises an error

ProcExit:

Exit Sub

ProcError:

Select Case Err.Number

Case X

' handle X

Case Y

' handle Y

Case Z
```

```
' handle Z

Case Else

' default

MsgBox Err.Description

Resume ProcExit

End Select

End Sub
```

X, Y, and Z may be literal numbers (**Case 13 ' type mismatch**) or, if they are available, symbolic constants representing the numbers.

If you are building a class module that will raise class-defined errors, you should provide a public enumeration in the class that defines constants for any errors raised by the class. By providing constants, code that creates objects defined by the class can use the constants instead of the literal numbers and protect itself from changes in the actual numbers.

Once you have trapped and handled the error, you need to tell **Visual Basic** where to continue with program execution. There are several options available when an error handling block is entered using **On Error Goto** label:

- ***Resume***: The Resume statement tells VB to continue execution with the line that generated the error.

- Resume Next: Resume Next instructs Visual Basic to continue execution with the line following the line that generated the error. This allows you to skip the offending code.

- Resume label: This allows you to redirect execution to any label within the current procedure. The label may be a location that contains special code to handle the error, an exit point that performs clean up operations, or any other point you choose.

- Exit: You can use Exit Sub, Exit Function, or Exit Property to break out of the current procedure and continue execution at whatever point you were at when the procedure was called.

- End: This is not recommended, but you can use the **End** statement to immediately terminate your application. Remember that if you use End, your application is forcibly terminated. No**Unload**, **QueryUnload**, or **Terminate** event procedures will be fired. This is the coding equivalent of a gunshot to the head for your application.

In addition to these statements, you can also call the **Clear** method of the **Err** object to clear the current error. This is most often used with inline error handling, as shown below:

```
Public Sub CreateFile(sFilename As String)

On Error Resume Next

' the next line will raise an error if the file

' doesn't exist, but we don't care because the

' point is to kill it if it does anyway.

Kill sFilename

Err.Clear

' code to create a file

End Sub
```

This isn't a very robust example. There are many other things besides a file that doesn't exist that could cause the Kill statement to fail. The file may be read-only, there may be a network permissions error, or some other problem.

Handling Errors You Can't Handle

In most cases you can anticipate the most common errors and build code to deal with them. The error handling code might be as simple as a message to the user such as "This field requires a valid date." In some cases, however, you will encounter errors you didn't anticipate and you must find a way to deal with them.

There are two general approaches you can take to handling unanticipated errors:

- **Assume that the error is not fatal to the application.** Most errors will not be fatal to an application. The **error** may have been bad data provided by a user, a file that was not found, etc. Normally these kinds of errors can be corrected by the user and the application can continue. A default case in an error handler can simply display a message and exit the current procedure or continue.

- **Assume that the error is fatal and the application must be terminated.** In some cases, any error may be an application killer. This should be rare because this kind of **error** should be explicitly handled, if necessary by providing the user with the tools or information necessary to correct the situation. However, if a situation occurs where an unanticipated error is fatal, you must be sure to clean up after yourself before you shut down the application by unloading all forms and releasing any object variables you have created.

You should try to avoid the latter situation at all times. Displaying a message and shutting down or – worse yet – just pulling the application out from under the user will not be well received. If you must terminate an application due to some disastrous situation, be sure to provide as much information to the user as you can so that the situation can be resolved. An even better option is to code your error handlers to call code that corrects severe problems.

For example, if you are designing a database application and encounter a corrupted databasefile, the error handling code could give the user the option of attempting to repair the damaged file. If a file cannot be found where it should be, write code to either look for it or give the user a file open dialog box so they can tell you where it went.

Raising Your Own Errors

There may be times when you need to generate errors in your code. This happens most often in class modules, but you can raise an error anywhere in a Visual Basic application. You raise an error by calling the Raise method of the **Err** object.

Not surprisingly, the parameters of the Raise method are the same as the properties of the **Err**object: Number, Description, Source, HelpContext, and HelpFile. The values you provide for these parameters are available to error handling code that deals with the error you generate.

When you raise an error, you should make the information you provide via the **Err** object as informative as possible so that error handling code that deals with the error has the best possible opportunity to handle it.

- Number: You can raise any of the standard VB error numbers or provide your own number. If you are raising application-defined errors, you need to add the intrinsic constant vbObjectError to the number you raise so that your number does not conflict with built in error numbers.

- Description: Make the description as informative as possible. If invalid data is provided, it may be helpful to make that data part of the error message.

- Source: The Source provides the name of the object that generated the error. For example, if a Jet **Database** object raises an error, the Source property is "DAO.Database".

- HelpContext: If you provide a help file with the component or application, use the **HelpContext** parameter to provide a context ID. This can then be passed on to the MsgBox statement so that context sensitive help about the error is available.

- HelpFile: This is the name of the help file and is used in conjunction with the **HelpContext**parameter.

The following example is a hypothetical property procedure for a class module:

```
' in the declarations section

Private mDate As Date

Public Enum MyClassErrors

errInvalidDate
```

```
' other errors

End Enum

' a property procedure

Public Property Let Date(vntDate As Variant)

' a variant is used to allow the property

' procedure to attempt to convert the value

' provided to a date

If IsDate(vntDate) Then



mDate = CDate(vntDate)

Else

' invalid data

Err.Raise _

Number:=errInvalidDate, _

Description:=CStr(vntDate) & " is not a valid date.", _

Source:="Foo.MyClass"

' help context and file go here if a help file is available

End If

End Property
```

In this example, the property procedure tests for a valid date using the **IsDate** function. If the data provided is not a date, an error is raised using the constant from the error enumeration in the declarations section of the class module and a description that includes the bad data and the nature of the error.

Summary

Handling run-time errors is something all applications must do if they are to be robust and reliable.

The key points for error handling are…There are two steps to handling run-time errors:

- Trap the error by enabling an error handler using the On Error statement.
- Handle the error by examining the properties of the Err object and writing code to deal with the problem.
- Error handlers can be dedicated blocks of code enabled by using **On Error Goto label** or can be inline handlers enabled by using **On Error Resume Next.**
- You can raise your own errors by calling the **Raise** method of the **Err** object.
- Do your best to handle run-time errors rather than just inform the user of the problem, but if you can't do anything but display a message, make it as informative as possible.
- Avoid terminating the application if at all possible.

The Err object was introduced in Visual Basic 4.0. For backward compatibility, VB continues to support the Err and Error statements and functions. Any new code should be using the Err object and legacy code should be converted to use the Err object.