Visual Basic 6 Error Handling

Try as hard as you might, it's virtually impossible to make the programs you write foolproof. Sad to say, programs bomb, that is ungracefully come to a grinding halt---and this can really leave a bad impression on the people using your programs. Nothing can ruin your programming reputation faster than having a customer mention to someone else that 'he loves your program, except that it bombs once in a while'.

The purpose of Visual Basic Error Handling is two-fold. First, to try to deal with problems that may occur so that your program continues to run. And secondly, if you cannot handle the error, to handle it in such a way that the program comes to a more graceful end.

How do errors occur in your program?

Hopefully, by the time you release a program to a user or a customer, there are no syntax or logic errors in the program. Those aren't the kinds of errors I'll be talking about in this article.

The kinds of errors I'm talking about are the kind that are largely beyond your control. An example of that kind of error would be that you write a program to read records from a database table, and prior to the program running, somehow your user manages to delete the database. Your program runs, looks for the database, can't find it, and the program bombs! Here's another example. You write a program, based on a customer's request, that reads data from a disk file of their choosing. You permit the customer to specify the location of the file, and they tell you that the file is in their diskette drive--but then they forget to insert the diskette. Sad but true, this can make your program bomb also.

When I mention this last example to my students, their typical response is one of disbelief---after all, packages such as Word or Excel don't bomb when the user does something like that---instead, a message is displayed informing the user that they need to insert a diskette into the diskette drive. Of course, that's the point---the programming team who worked on Word and Excel anticipated this kind of error---and wrote code to deal with it.

Visual Basic provides something called Error Handling Procedures which permits your program to recognize that an error of this sort has occurred, and allows you to write code to deal with it.

Our program bombs

What I want to do now is to intentionally produce an error, and to show you how Error Handling can be used to graceful handle it.

Let's start a new Visual Basic Project, and place a Command Button on the form. I'll place the following code into the Click Event Procedure of the form.

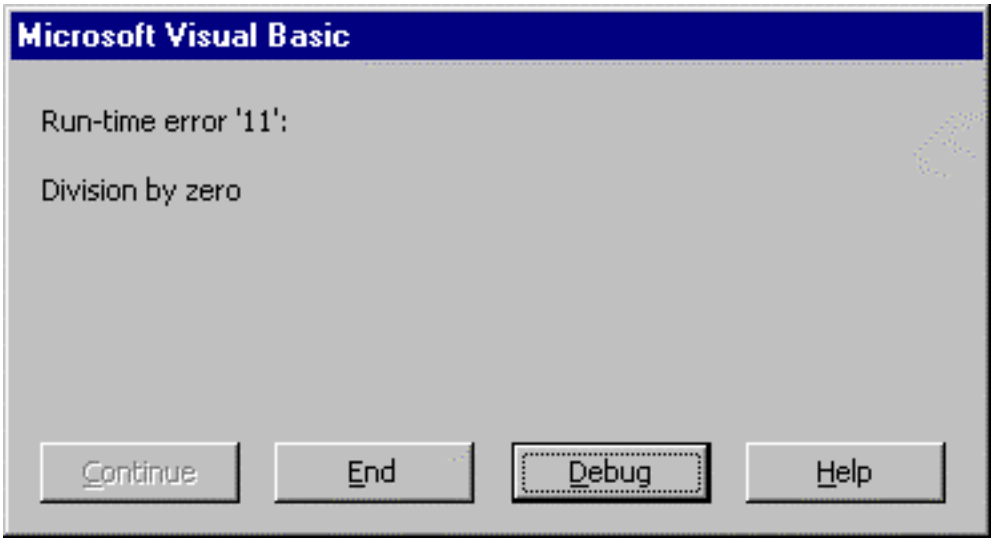
```
Private Sub Command1_Click()

Dim intValue1 As Integer
Dim intValue2 As Integer

intValue1 = 12
intValue2 = 0

Form1.Print intValue1 / intValue2

End Sub
```

Quite often, when you see a discussion of Error Handling, you'll see code similar to what you see here. This line of code

```
Form1.Print intValue1 / intValue2
```

tells Visual Basic to divide the value of intValue1--which is 12 by the value of intValue2---which is 0. In the computer world, division by zero is a 'no-no'---theoretically the result is undefined, but the bottom line is that it's illegal, and generates an error.

If we now run this program, it will bomb like this.



This is the error message you'll see in the Visual Basic IDE---if this program had been compiled into an executable, the error message would look like this:

As you can see, the only thing a user of your program can do is look at the Error Message and click on the OK button---at this point the program just terminates. The repercussions of this behavior are many---if, for example, your user had been using your program to perform processing against a file or a database, having the program end ungracefully like this could mean that they lose the work they've been doing. At best, having a program terminate like this leaves a bad impression of your program---and you---with them.

Let's implement a Visual Basic Error Handler now to more gracefully deal with this situation.

A more graceful end

Let's modify the code in the Click Event Procedure of the Command button to look like this:

```
Private Sub Command1_Click()

On Error GoTo ICanDealWithThis

Dim intValue1 As Integer
Dim intValue2 As Integer

intValue1 = 12
intValue2 = 0

Form1.Print intValue1 / intValue2

Exit Sub

ICanDealWithThis:
```

```
MsgBox "Error Number:" & Err.Number & vbCrLf & Err.
Description
Resume Next

End Sub
```

If we now run the program and click on the Command button, this time a message box of our own choosing is displayed.

At first glance, it may appear that all we've done is substitute our error message for Visual Basic's Error Message. However, you'll notice that if we (or the user) clicks on the OK button, the program keeps running---in other words, no ungraceful termination. That's what Error Handling gives us--the ability to handle an error condition gracefully. In fact, errors may occur and the user doesn't even have to know they happened.

So what's the difference?

What we've done here is to enable a Visual Basic Error Handler. A Visual Basic Error Handler must be coded in each procedure where you want the ability to 'intercept' an error the way we did here. Once you've coded it, if an error (any error at all) occurs in that procedure, Visual Basic will then skip right to code in the Error Handler, and execute whatever code it finds there. If no error occurs in the procedure, then Visual Basic will execute the code as normal.

Let's take a look at the line of code that 'enables' the Error Handler in the Click Event Procedure.

```
On Error GoTo ICanDealWithThis
```

With this line of code, we're telling Visual Basic to branch or jump to a section in the procedure called 'ICanDealWithThis'. Despite what you may have heard about the use of the 'GoTo' statement, this is the only way to implement an Error Handler in Visual Basic.

Getting back to 'ICanDealWithThis'---that's actually something called a Visual Basic Label. In old time Basic, a label was a section name in your code---almost like a bookmark in Microsoft Word. You hardly ever see it used anymore, except of course with an Error Handler. In Visual Basic, you designate a label in your code by entering the label name in your procedure, followed by a colon (:).

```
ICanDealWithThis:
```

The code that follows the label name is the actual code comprising the Error Handler---that's the code that is executed if an error occurs in the procedure.

By the way, I should mention that you can name the Error Handler label anything you want (It can't have spaces in its name). Frequently, you'll see examples where it's called ErrorHandler--but it can be called anything at all.

By convention, the Error Handler is placed at the end of the procedure, and you should also note that right before the Error Handler label name is an Exit Sub statement

```
Exit Sub

ICanDealWithThis:

MsgBox "Error Number:" & Err.Number & vbCrLf & Err.
Description
Resume Next
```

You must always code an Exit Sub statement immediately in front of the Error Handler label. Without an Exit Sub, because of the 'falling rock' behavior of Visual Basic code, Visual Basic would eventually execute the code in the Error Handler regardless of whether an error actually occurred in the procedure. Exit Sub is jus a way of skipping around the Error Handler code, and ending the procedure.

The Err Object

Let's take a look at the Msgbox statement now.

Within the MsgBox, we are displaying the Number and Description properties of something called the Visual Basic Err object. The Err object is created or instantiated whenever a Visual Basic error occurs. Because it's an Object, just like a form or a control, we have access to its properties from

within our program, and we can display its property values in a MsgBox. The Number property is used to display the Error Number---11 in this case---and the Description property is used to display the Error description.

Resume, Resume Next

You may be wondering about that Resume Nest statement.

Resume Next

Up to this point in the Error Handler, all we've done is react to the error by display a user friendly error message. At this point, if we did nothing else, the program would reach the End Sub statement in the Click Event procedure, and the procedure would end. That may be fine in some situations, but Visual Basic gives you the ability to deal more specifically with the line of code that caused the error. You can do one of three things:

1. You can end the program by coding an Exit Sub statement within the Error Handler or just let the program reach the End Sub and end that way.

2. You can instruct Visual Basic to continue running the program at the same line of code that caused the error to begin with. You do this by coding a Resume statement.

3. You can instruct Visual Basic to resume running the program from the line of code FOLLOWING the one that caused the error. You do that by coding a Resume Next statement.

Many of you might be saying that choice number 2 sounds the best--and it may be in most cases. After all, why go back to the line of code that generated the error.

That's a good point, however, it is possible that the condition that caused the error can be corrected by the user, and the line of code that generate the error condition may execute successfully the next time around. For instance, remember my earlier example of the user who tells the program that there is a file on a diskette in the diskette drive---but then forgets to insert the diskette? A user friendly message asking them to insert the floppy, followed by a Resume statement will do the trick!

The decision as to how to proceed after the code in the Error Handler has been executed is influenced by the likelihood that either the program itself or the user can somehow correct the problem that caused the error in the first

place.

In the example we've been following, there's absolutely no sense in continuing execution with the same line of code because there's no way that the expression will ever result in anything other than division by zero. Coding a **Resume** statement here would be foolish, and result in what I call an 'Endless Error'---like an Endless Loop. So be careful whenever you decide to handle an error by coding a Resume Statement.

Neither should you just automatically code a **Resume Next** statement in a vacuum. Since Resume Next essentially 'skips' the line of code that generated the error, if that line of code is crucial to the successful completion of the procedure, this can cause disastrous results as well.

The bottom line, is that you've got to carefully consider your Error Handler strategy in light of the types of conditions you might encounter. In other words, you may need to take different actions at different times. Remember, the code in the Error Handler may be triggered by different errors.

How would you know this?

You can check the Number property of the Err object, and take different corrective action based on the error that triggered the Error Handler. Let me show you.

Select Case in an Error Handler

```
Private Sub Command1_Click()

On Error GoTo ICanDealWithThis

Dim intValue1 As Integer
Dim intValue2 As Integer

intValue1 = 12
intValue2 = 0

Form1.Print intValue1 / intValue2

Exit Sub

ICanDealWithThis:
```

```
Select Case Err.Number
  Case 11
    MsgBox "Division by Zero"
  Case Else
    MsgBox "Error Number:" & Err.Number & vbCrLf & Err.
Description
End Select

Resume Next

End Sub
```

The difference between this code and the previous version? The Select Case statement in the Error Handler. Using Select Case to evaluate the Number property of the Err Object, you can react to different kinds of errors that may occur in your program.

You might be asking yourself how you can anticipate each and every error that may occur in your program? The answer is that you can't--try to anticipate as many as you can (experience is the best teacher) and write to code to react to those. And for those errors that might occur that you don't anticipate? Use the Case Else statement to handle those by displaying the Error Number and Description.

```
Case Else

MsgBox "Error Number:" & Err.Number & vbCrLf & Err.
Description
```

By doing this, you display the Error Number and the message for the user to see (you might also want to customize your message a little bit more), plus most importantly, you prevent the program from ungracefully terminating.

Do you need an Error Handler in every procedure?

The answer is yes---you need to code an Error Handler in every procedure in which you wish to catch an error. Microsoft's recommendation is to code an Error Handler in every procedure---in practice, you see both extremes--that is, no Error Handling to an Error Handler in every procedure.

For those of you concerned about this, who may be writing commercial programs, you might want to consider a product called FailSafe--which examines your code and placing Error Handling in every procedure for you. It may be well worth the price.

Inline Error Handling

There's just one more thing I want to discuss and that's something called Inline Error Handing.

With Inline Error Handling, you don't set up a formal Error Handler, but you tell Visual Basic with a single line of code how you want to handle errors. Here's our Click Event Procedure modified to use Inline Error Handling.

```
Private Sub Command1_Click()

On Error Resume Next

Dim intValue1 As Integer
Dim intValue2 As Integer

intValue1 = 12
intValue2 = 0

Form1.Print intValue1 / intValue2

End Sub
```

As you can see, the Error Handler has disappeared, and has been replaced with this single statement:

```
On Error Resume Next
```

What this is doing is telling Visual Basic to resume with the next line of code in the event of an error---no user friendly error message this time, no ability to react to different errors, but the program won't bomb either. In line Error Handling is the most bare bones type of Error Handling. Let's put it this way, it's better than nothing, but I don't use it myself. By the way, you're only choice with Inline Error Handling is to Resume Next.

Summary

The ability to detect and react to errors is vital to a sophisticated program. Hopefully, you have a good idea as to how to detect and react to errors that may occur in your own programs.