VBA Error Handling – A Complete Guide

NOVEMBER 23, 2016 (HTTPS://EXCELMACROMASTERY.COM/VBA-ERROR-HANDLING/) BY PAUL KELLY (HTTPS://EXCELMACROMASTERY.COM/AUTHOR/ADMIN/) · 25 COMMENTS (HTTPS://EXCELMACROMASTERY.COM/VBA-ERROR-HANDLING/#COMMENTS)

"Abort, Retry, Fail?" – MS-DOS error message circa 1986

This post provides a complete guide to VBA Error Handing. If you are looking for a **quick summary** then check out the quick guide table in the first section.

If you are looking for a particular topic on VBA Error Handing then check out the table of contents below(if it's not visible click on the post header).

If you are **new to VBA Error Handling**, then you can read the post from start to finish as it is laid out in logical order.

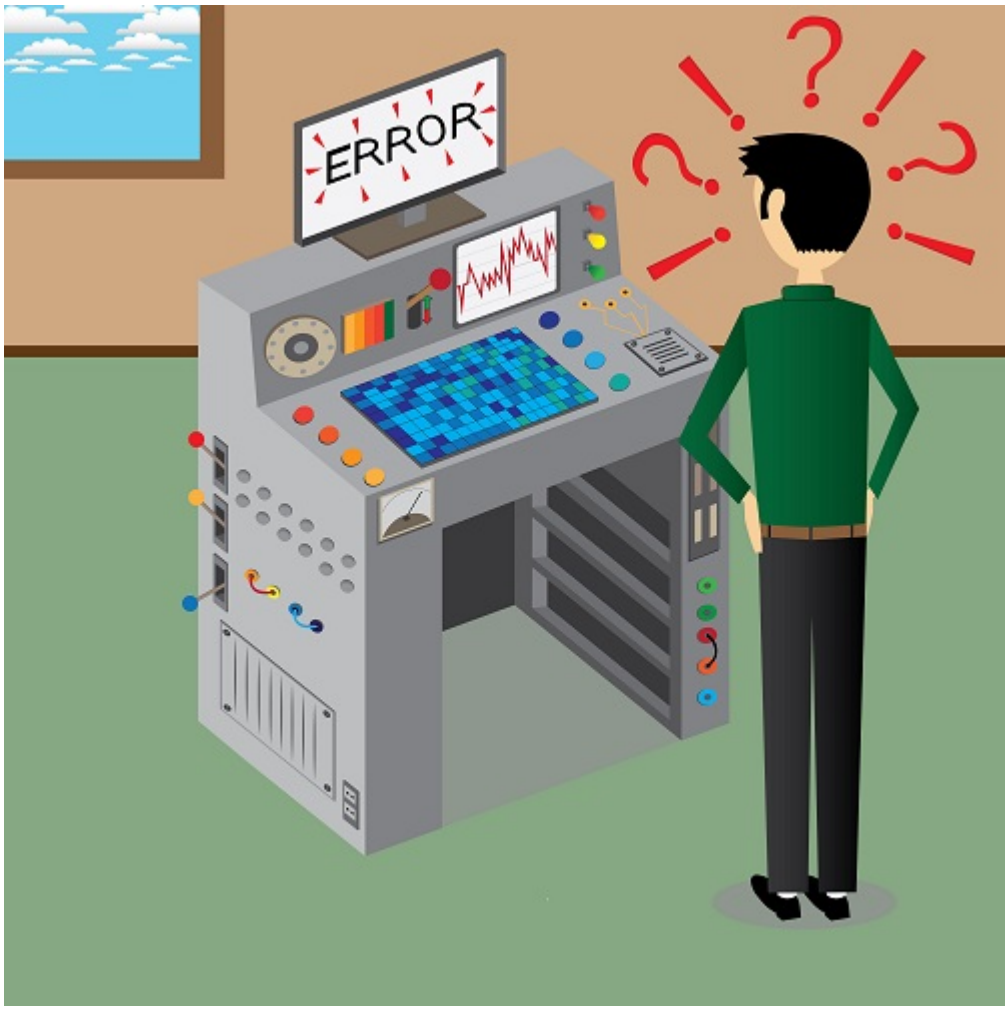
Contents [hide]

A Quick Guide to Error Handing

Item	Description
On Error Goto 0	When error occurs, the code stops and displays the error.

Item	Description
On Error Resume Next	Ignores the error and continues on.
On Error Goto [Label]	Goes to a specific label when an error occurs. This allows us to handle the error.
Err Object	When an error occurs the error information is stored here.
Err.Number	The number of the error. (Only useful if you need to check a specific error occurred.)
Err.Description	Contains the error text.
Err.Source	You can populate this when you use Err.Raise.

To understand error handling we must first understand the different types of errors in VBA.



VBA Errors

There are three types of errors in VBA

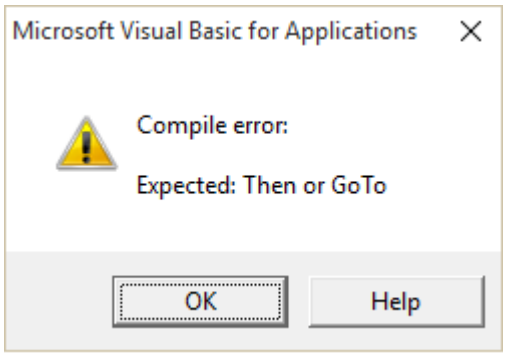
1. Syntax
2. Compilation
3. Runtime

We use error handling to deal with runtime errors. Let's have a look at each of these error types so that it is clear what a runtime error is.

Syntax Errors

If you have used VBA for any length of time you will have seen a syntax error. When you type a line and press return, VBA will evaluate the syntax and if it is not correct it will display an error message.

For example if you type **If** and forget the **Then** keyword, VBA will display the following error message



Some examples of syntax errors are

```
' then is missing
If a > b

' equals is missing after i
For i 2 To 7

' missing right parenthesis
b = left("ABCD",1
```

Syntax errors relate to one line only. They occur when the syntax of one line is incorrect.

Note: You can turn off the Syntax error dialog by going to Tools->Options and checking off “Auto Syntax Check”. The line will still appear red if there is an error but the dialog will not appear.

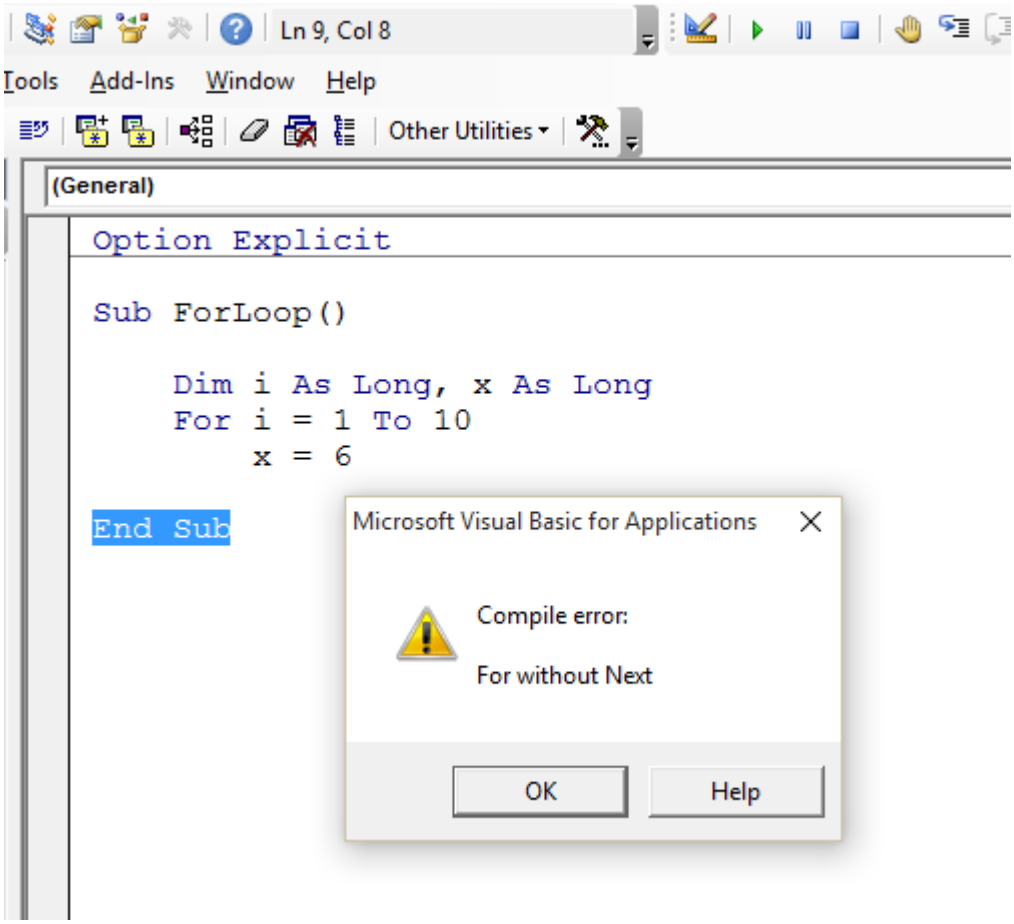
Compilation Errors

Compilation errors occur over more than one line. The syntax is correct on a single line but is incorrect when all the project code is taken into account.

Examples of compilation errors are:

- **If** statement without corresponding **End If** statement
- **For** without **Next**
- **Select** without **End Select**
- Calling a Sub or Function that does not exist
- Calling a Sub or Function with the wrong parameters
- Giving a Sub or Function the same name as a module
- Variables not declared(**Option Explicit** must be present at the top of the module)

The following screenshot shows a compilation error that occurs when a **For** loop has no matching **Next** statement.



Using Debug->Compile

To find compilation errors, we use **Debug->Compile VBA Project** from the Visual Basic menu.

When you select *Debug->Compile*, VBA displays the first error it comes across.

When this error is fixed, you can run Compile again and VBA will then find the next error.

Debug->Compile will also include syntax errors in it's search which is very useful.

If there are no errors left and you run *Debug->Compile* , it may appear that nothing happened. However, "Compile" will be grayed out in the Debug menu. This means your application has no compilation errors at the current time.

Debug->Compile Error Summary

- Debug->Compile finds compilation(project wide) errors.
- It will also find syntax errors.
- It finds one error each time you use it.
- When there are no compilation errors left the Compile option will appear grayed out in the menu.

Debug->Compile Usage

You should always use *Debug->Compile* before you run your code. This ensures that your code has no compilation errors when you run it.

If you do not run *Debug->Compile* then VBA may find compile errors when it runs. These should not be confused with Runtime errors.

Runtime Errors

Runtime errors occur when your application is running. They are normally outside of your control but can be caused by errors in your code.

For example, imagine your application reads from an external workbook. If this file gets deleted then VBA will display an error when your code tries to open it.

Other examples of runtime errors are

- a database not being available
- the user entering invalid data
- a cell containing text instead of a number

As we have seen, the purpose of error handling is to deal with runtime errors when they occur.

Expected Versus Unexpected Errors

When we think a runtime error could occur we put code in place to handle it. For example, we would normally put code in place to deal with a file not being found.

The following code checks if the file exists before it tries to open it. If the file does not exist then a user friendly message is displayed and the code exits the sub.

```
Sub OpenFile()

    Dim sFile As String
    sFile = "C:\docs\data.xlsx"

    ' Use Dir to check if file exists
    If Dir(sFile) = "" Then
        ' if file does not exist display message
        MsgBox "Could not find the file " & sFile
        Exit Sub
    End If

    ' Code will only reach here if file exists
    Workbooks.Open sFile

End Sub
```

When we think an error is likely to occur at some point, it is good practice to add code to handle the situation. We normally refer to these errors as *expected* errors.

If we don't have specific code to handle an error it is considered an *unexpected* error. We use the VBA error handling statements to handle the unexpected errors.

Runtime Errors that are not VBA Errors

Before we look at the VBA Handling there is one type of error we must mention. Some runtime errors are not considered errors by VBA but only by the user.

Let me explain this with an example. Imagine you have an application that requires you to add the values in the variables **a** and **b**

```
result = a + b
```

Let's say you mistakenly use an asterisk instead of the plus sign

```
result = a * b
```

This is not a VBA error. Your code syntax is perfectly legal. However, from your requirements point of view it is an error.

These errors cannot be dealt with using error handling as they obviously won't generate any error. You can deal with these errors using Unit Testing and Assertions. I have an in-depth post about using VBA assertions – see How to Make Your Code BulletProof (http://excelmacromastery.com/bulletproof-vba-code/).

The On Error Statement

As we have seen there are two ways to treat runtime errors

1. Expected errors – write specific code to handle them.
2. Unexpected errors – use VBA error handling statements to handle them.

The VBA **On Error** statement is used for error handling. This statement performs some action when an error occurs during runtime.

There are four different ways to use this statement

1. **On Error Goto 0** – the code stops at the line with the error and displays a message.
2. **On Error Resume Next** – the code moves to next line. No error message is displayed.
3. **On Error Goto [label]** – the code moves to a specific line or label. No error message is displayed. This is the one we use for error handling.
4. **On Error Goto -1** – clears the current error.

Let's look at each of these statements in turn.

On Error Goto 0

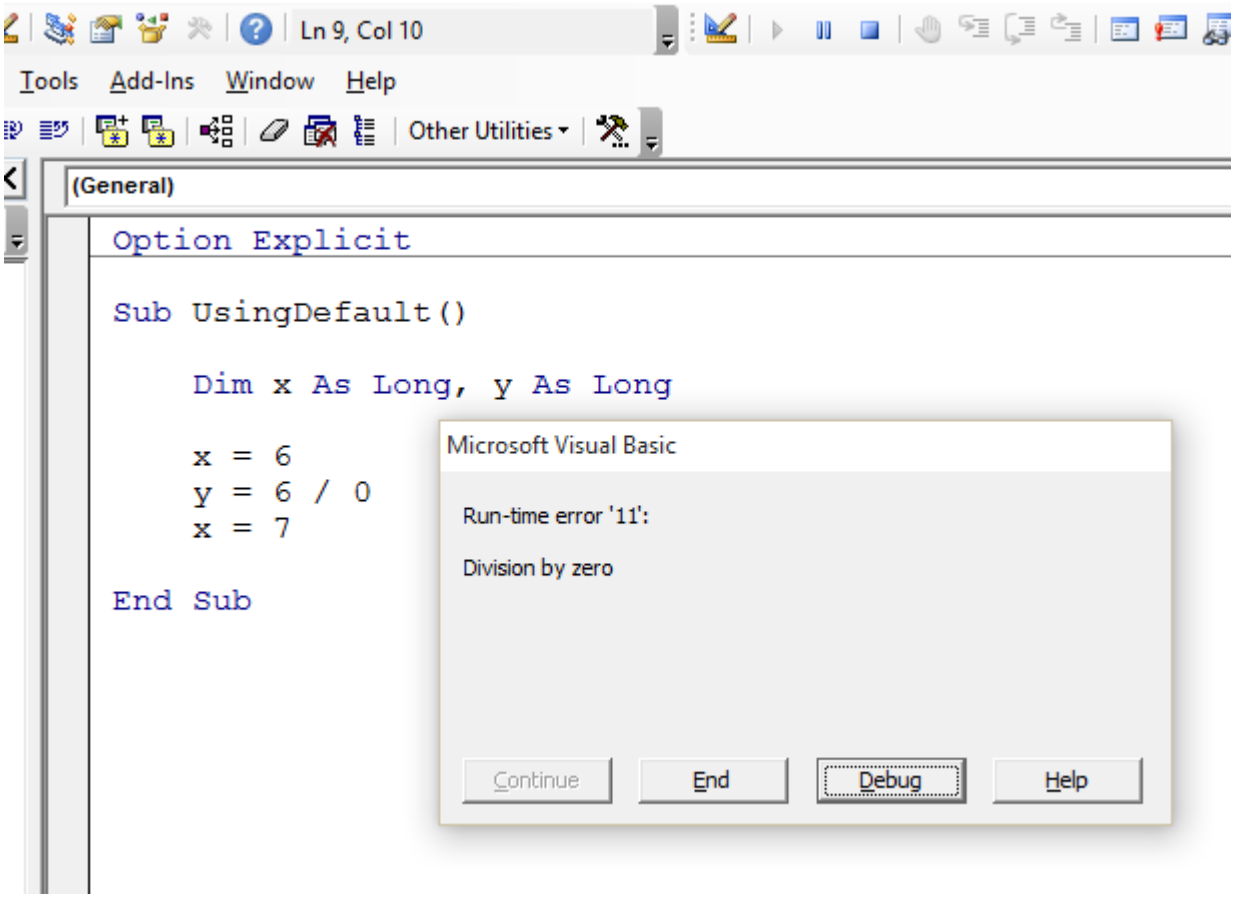
This is the default behavior of VBA. In other words, if you don't use **On Error** then this is the behavior you will see.

When an error occurs, VBA stops on the line with the error and displays the error message. The application requires user intervention with the code before it can continue. This could be fixing the error or restarting the application. In this scenario no error handling takes place.

Let's look at an example. In the following code, we have not used any **On Error** line so VBA will use the **On Error Goto 0** behavior by default.

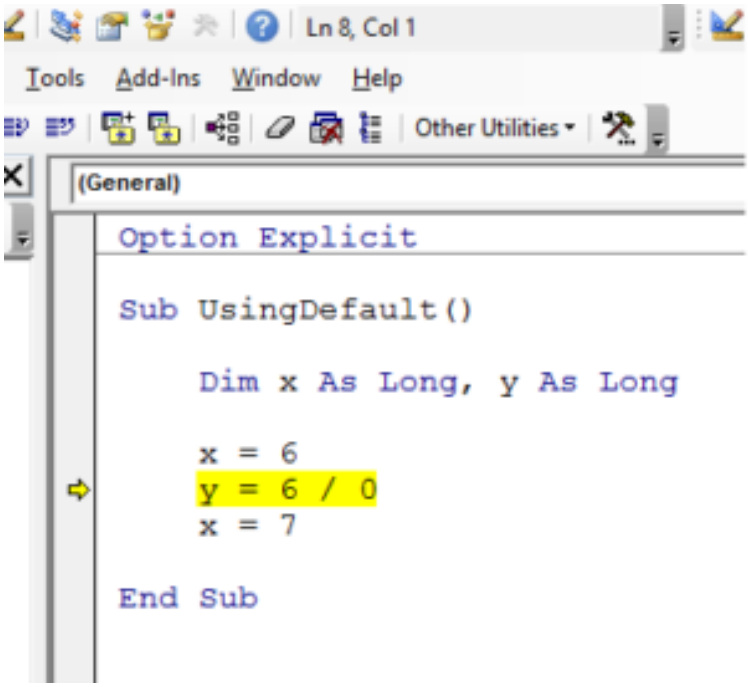
```
Sub UsingDefault()

    Dim x As Long, y As Long

    x = 6
    y = 6 / 0
    x = 7

End Sub
```

The second assignment line results in a divide by zero error. When we run this code we will get the error message shown in the screenshot below



When the error appears you can choose **End** or **Debug**

If you select **End** then the application simply stops.
If you select **Debug** the application stops on the error line as the screenshot below shows



This behaviour is fine when you are writing VBA code as it shows you the exact line with the error.

This behavior is unsuitable for an application that you are given to a user. These errors look unprofessional and they make the application look unstable.

An error like this is essentially the application crashing. The user cannot continue on without restarting the application. They may not use it at all until you fix the error for them.

By using **On Error Goto [label]** we can give the user a more controlled error message. It also prevents the application stopping. We can get the application to perform in a predefined manner.

On Error Resume Next

Using **On Error Resume Next** tells VBA to ignore the error and continue on.

There are specific occasions when this is useful. Most of the time you should avoid using it.

If we add **Resume Next** to our example Sub then VBA will ignore the divide by zero error

```
Sub UsingResumeNext()

    On Error Resume Next

    Dim x As Long, y As Long

    x = 6
    y = 6 / 0
    x = 7

End Sub
```

It is not a good idea to do this. If you ignore the error, then the behavior can be unpredictable. The error can affect the application in multiple ways.You could end up with invalid data. The problem is that you aren't aware that something went wrong because you have suppressed the error.

The code below is an example of where using **Resume Next** is valid

```
Sub SendMail()

   On Error Resume Next

    ' Requires Reference:
    ' Microsoft Outlook 15.0 Object Library
    Dim Outlook As Outlook.Application
    Set Outlook = New Outlook.Application

    If Outlook Is Nothing Then
        MsgBox "Cannot create Microsoft Outlook session." _
                   & " The email will not be sent."
        Exit Sub
    End If

End Sub
```

In this code we are checking to see if Microsoft Outlook is available on a computer. All we want to know is if it is available or not. We are not interested in the specific error.

In the code above, we continue on if there is an error. Then in the next line we check the value of the **Outlook** variable. If there has been an error then the value of this variable will be set to **Nothing**.

This is an example of when **Resume** could be useful. The point is that even though we use **Resume** we are still checking for the error. The vast majority of the time you will not need to use **Resume**.

On Error Goto [label]

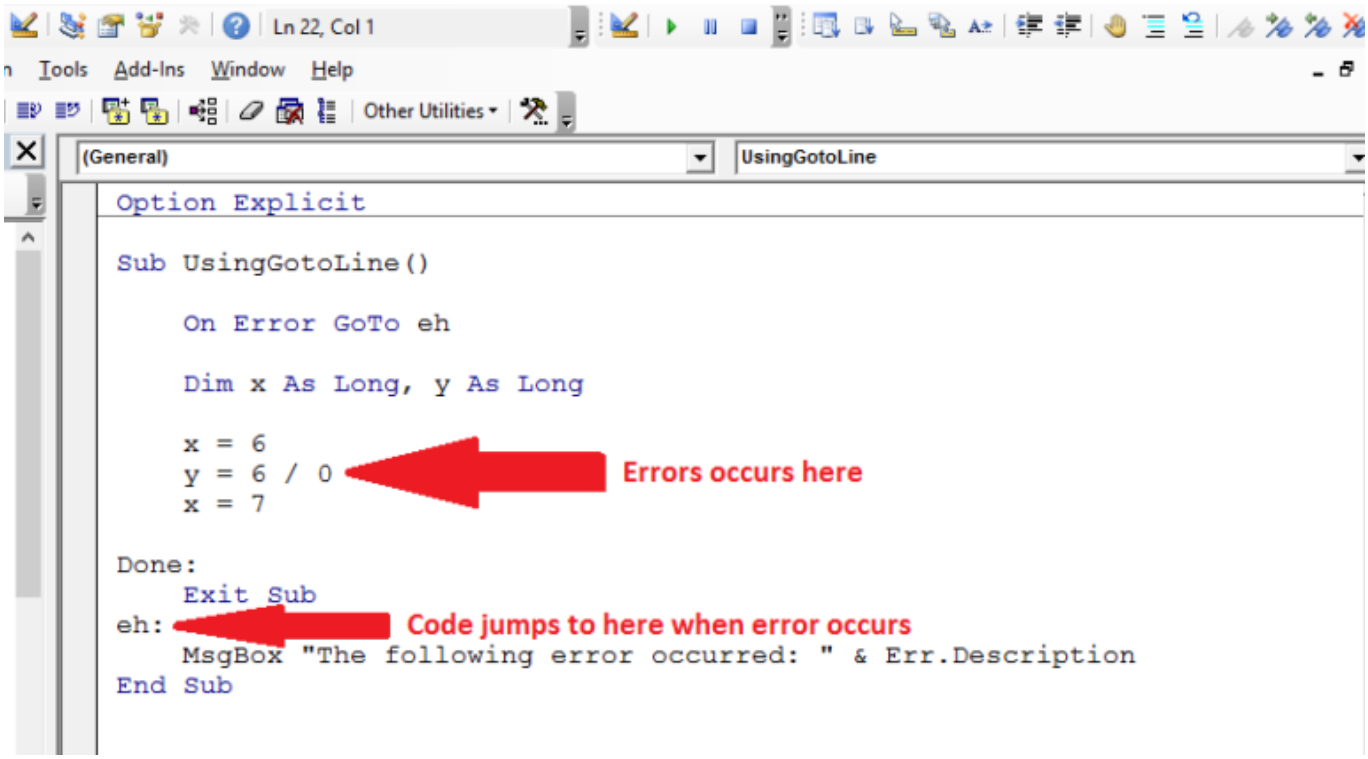
This is how we use Error Handling in VBA. It is the equivalent of the **Try** and **Catch** functionality you see in languages such as C# and Java.

When an error occurs you send the error to a specific label. It is normally at the bottom of the sub.

Let’s apply this to the sub we have been using

```
Sub UsingGotoLine()

    On Error Goto eh

    Dim x As Long, y As Long

    x = 6
    y = 6 / 0
    x = 7

Done:
    Exit Sub
eh:
    MsgBox "The following error occurred: " & Err.Description
End Sub
```

The screenshot below shows what happens when an error occurs



VBA jumps to the **eh** label because we specified this in the **On Error Goto** line.

Note 1: The label we use in the On…Goto statement, must be in the current Sub/Function. If not you will get a compilation error.

Note 2: When an error occurs when using *On Error Goto [label]*, the error handling returns to the default behaviour i.e. The code will stop on the line with the error and display the error message. See the next section for more information about this.

On Error Goto -1

This statement is different than the other three. It is used to clear the current error rather than setting a particular behaviour.

When an error occurs using *On Error Goto [label]*, the error handling behaviour returns to the default behaviour i.e. “On Error Goto 0”. That means that if another error occurs the code will stop on the current line.

This behaviour only applies to the current sub. Once we exit the sub, the error will be cleared automatically.

Take a look at the code below. The first error will cause the code to jump to the *eh* label. The second error will stop on the line with the 1034 error.

```
Sub TwoErrors()

    On Error Goto eh

    ' generate "Type mismatch" error
    Error (13)

Done:
    Exit Sub
eh:
    ' generate "Application-defined" error
    Error (1034)
End Sub
```

If we add further error handling it will not work as the error trap has not been cleared.

In the code below we have added the line

```
On Error Goto eh_other
```

after we catch the first error.

This has no effect as the error has not been cleared. In other words the code will stop on the line with the error and display the message.

```
Sub TwoErrors()

    On Error Goto eh

    ' generate "Type mismatch" error
    Error (13)

Done:
    Exit Sub
eh:
    On Error Goto eh_other
    ' generate "Application-defined" error
    Error (1034)
Exit Sub
eh_other:
    Debug.Print "ehother " & Err.Description
End Sub
```

To clear the error we use *On Error Goto -1*. Think of it like setting a mouse trap. When the trap goes off you need to set it again.

In the code below we add this line and the second error will now cause the code to jump to the *eh_other* label

```
Sub TwoErrors()

    On Error Goto eh

    ' generate "Type mismatch" error
    Error (13)

Done:
    Exit Sub
eh:
    ' clear error
    On Error Goto -1

    On Error Goto eh_other
    ' generate "Application-defined" error
    Error (1034)
Exit Sub
eh_other:
    Debug.Print "ehother " & Err.Description
End Sub
```

Note 1: There are probably rare cases where using *On Error Goto -1* is useful. I have personally never needed to use this line. Remember that once you leave the sub the error will be cleared anyway.

Note 2: The Err Object has a member *Clear*. Using *Clear* clears the text and numbers in the Err object, but it does NOT reset the error.

Using On Error

As we have seen, VBA will do one of three things when an error occurs

- Stop and display the error.
- Ignore the error and continue on.
- Jump to a specific line.

VBA will always be set to one of these behaviors. When you use **On Error**, VBA will change to the behaviour you specify and forget about any previous behavior.

In the following Sub, VBA changes the error behaviour each time we use the **On Error** statement

```
Sub ErrorStates()

    Dim x As Long

    ' Go to eh label if error
    On Error Goto eh

    ' this will ignore the error on the following line
    On Error Resume Next
    x = 1 / 0

    ' this will display an error message on the following line
    On Error Goto 0
    x = 1 / 0

Done:
   Exit Sub
eh:
    Debug.Print Err.Description
End Sub
```

The Err Object

When an error occurs you can view details of the error using the **Err** object.

When an runtime error occurs, VBA automatically fills the **Err** object with details.

The code below will print *"Error Number: 13 Type Mismatch"* which occurs when we try to place a string value in the long integer *total*

```
Sub UsingErr()

    On Error Goto eh

    Dim total As Long
    total = "aa"

Done:
    Exit Sub
eh:
    Debug.Print "Error number: " & Err.Number _
            & " " & Err.Description
End Sub
```

The **Err.Description** provides details of the error that occurs. This is the text you normally see when an error occurs e.g. "Type Mismatch"

The **Err.Number** is the ID number of the error e.g. the error number for "Type Mismatch" is 13. The only time you really need this is if you are checking that a specific error occurred and this is only necessary on rare occasions.

The **Err.Source** property seems like a great idea but it does not work for a VBA error. The source will return the project name, which hardly narrows down where the error occurred. However, if you create an error using **Err.Raise** you can set the source yourself and this can be very useful.

Getting the Line Number

The **Erl** function is used to return the line number where the error occurs.

It often causes confusion. In the following code, *Erl* will return zero

```
Sub UsingErr()

    On Error Goto eh

    Dim val As Long
    val = "aa"

Done:
    Exit Sub
eh:
    Debug.Print Erl
End Sub
```

This is because there are no line numbers present. Most people don't realise it but VBA allows you to have line numbers.

If we change the Sub above to have line number it will now print out 20

```
Sub UsingErr()

10        On Error Goto eh

          Dim val As Long
20        val = "aa"

Done:
30        Exit Sub
eh:
40        Debug.Print Erl
End Sub
```

Adding line numbers to your code manually is cumbersome. However there are tools available that will allow you to easily add and remove line numbers to a sub.

When you are finished working on a project and hand it over to the user it can be useful to add line numbers at this point. If you use the error handling strategy in the last section of this post, then VBA will report the line where the error occurred.

Using Err.Raise

Err.Raise allows us to create errors. We can use it to create custom errors for our application which is very useful. It is the equivalent of the **Throw** statement in Java\C#.

The format is as follows

```
Err.Raise [error number], [error source], [error description]
```

Let's look at a simple example. Imagine we want to ensure that a cell has an entry that has a length of 5 characters. We could have a specific message for this

```
Public Const ERROR_INVALID_DATA As Long = vbObjectError + 513

Sub ReadWorksheet()

    On Error Goto eh

    If Len(Sheet1.Range("A1")) <> 5 Then
        Err.Raise ERROR_INVALID_DATA, "ReadWorksheet" _
            , "The value in the cell A1 must have exactly 5 characters."
    End If

    ' continue on if cell has valid data
    Dim id As String
    id = Sheet1.Range("A1")


Done:
    Exit Sub
eh:
    ' Err.Raise will send code to here
    MsgBox "Error found: " & Err.Description
End Sub
```

When we create an error using *Err.Raise* we need to give it a number. We can use any number from 513 to 65535 for our error. We must use *vbObjectError* with the number e.g.

```
Err.Raise vbObjectError + 513
```

Using Err.Clear

Err.Clear is used to clear the text and numbers from the Err.Object. In other words, it clears the description and number.

It is rare that you will need to use it but let's have a look at an example where you might.

In the code below we are counting the number of errors that will occur. To keep it simple we are generating an error for each odd number.

We check the error number each time we go through the loop. If the number does not equal zero then an error has occurred. Once we count the error we need to set the error number back to zero so it is ready to check for the next error.

```
Sub UsingErrClear()

    Dim count As Long, i As Long

    ' Continue if error as we will check the error number
    On Error Resume Next

    For i = 0 To 9
        ' generate error for every second one
        If i Mod 2 = 0 Then Error (13)

        ' Check for error
        If Err.Number <> 0 Then
            count = count + 1
            Err.Clear    ' Clear Err once it is counted
        End If
    Next

    Debug.Print "The number of errors was: " & count
End Sub
```

Note 1: *Err.Clear* resets the text and numbers in the error object but it does not clear the error – see On Error Goto -1 for more information about clearing the actual error.

Logging

Logging means writing information from your application when it is running. When an error occurs you can write the details to a text file so you have a record of the error.

The code below shows a very simple logging procedure

```
Sub Logger(sType As String, sSource As String, sDetails As String)

    Dim sFilename As String
    sFilename = "C:\temp\logging.txt"

    ' Archive file at certain size
    If FileLen(sFilename) > 20000 Then
        FileCopy sFilename _
            , Replace(sFilename, ".txt", Format(Now, "ddmmyyyy hhmmss.txt"))
        Kill sFilename
    End If

    ' Open the file to write
    Dim filenumber As Variant
    filenumber = FreeFile
    Open sFilename For Append As #filenumber

    Print #filenumber, CStr(Now) & "," & sType & "," & sSource _
                               & "," & sDetails & "," & Application.UserName

    Close #filenumber

End Sub
```

You can use it like this

```
' Create unique error number
Public Const ERROR_DATA_MISSING As Long = vbObjectError + 514

Sub CreateReport()

    On Error Goto eh

    If Sheet1.Range("A1") = "" Then
       Err.Raise ERROR_DATA_MISSING, "CreateReport", "Data is missing from Cell A
    End If

    ' other code here
Done:
    Exit Sub
eh:
    Logger "Error", Err.Source, Err.Description
End Sub
```

The log is not only for recording errors. You can record other information as the application runs. When an error occurs you can then check the sequence of events before an error occurred.

Below is an example of logging. How you implement logging really depends on the nature of the application and how useful it will be.

```
Sub ReadingData()

    Logger "Information", "ReadingData()", "Starting to read data."

    Dim coll As New Collection
    ' Read data
    Set coll = ReadData

    If coll.Count < 10 Then
        Logger "Warning", "ReadingData()", "Number of data items is low."
    End If
    Logger "Information", "ReadingData()", "Number of data items is " & coll.Cour

    Logger "Information", "ReadingData()", "Finished reading data."

End Sub
```

Having a lot of information when dealing with an error can be very useful. Often the user may not give you accurate information about the error that occurred. By looking at the log you can get more accurate information about the information.

Other Error Related Items

This section covers some of the other Error Handling tools that VBA has. These items are considered obsolete but I have included them as they may exist in legacy code.

Error Function

The Error Function is used to print the error description from a given error number. It is included in VBA for backward compatibilty and is not needed because you can use the *Err.Description* instead.

Below are some examples

```
' Print the text "Division by zero"
Debug.Print Error(11)
' Print the text "Type mismatch"
Debug.Print Error(13)
' Print the text "File not found"
Debug.Print Error(53)
```

Error Statement

The Error statement allows you to simulate an error. It is included in VBA for backward compatibility. You should use **Err.Raise** instead.

In the following code we simulate a “Divide by zero” error.

```
Sub SimDivError()

    On Error Goto eh

    ' This will create a division by zero error
    Error 11

    Exit Sub
eh:
    Debug.Print Err.Number, Err.Description
End Sub
```

This statement is included in VBA for backward compatibility. You should use Err.Raise instead.

A Simple Error Handling Strategy

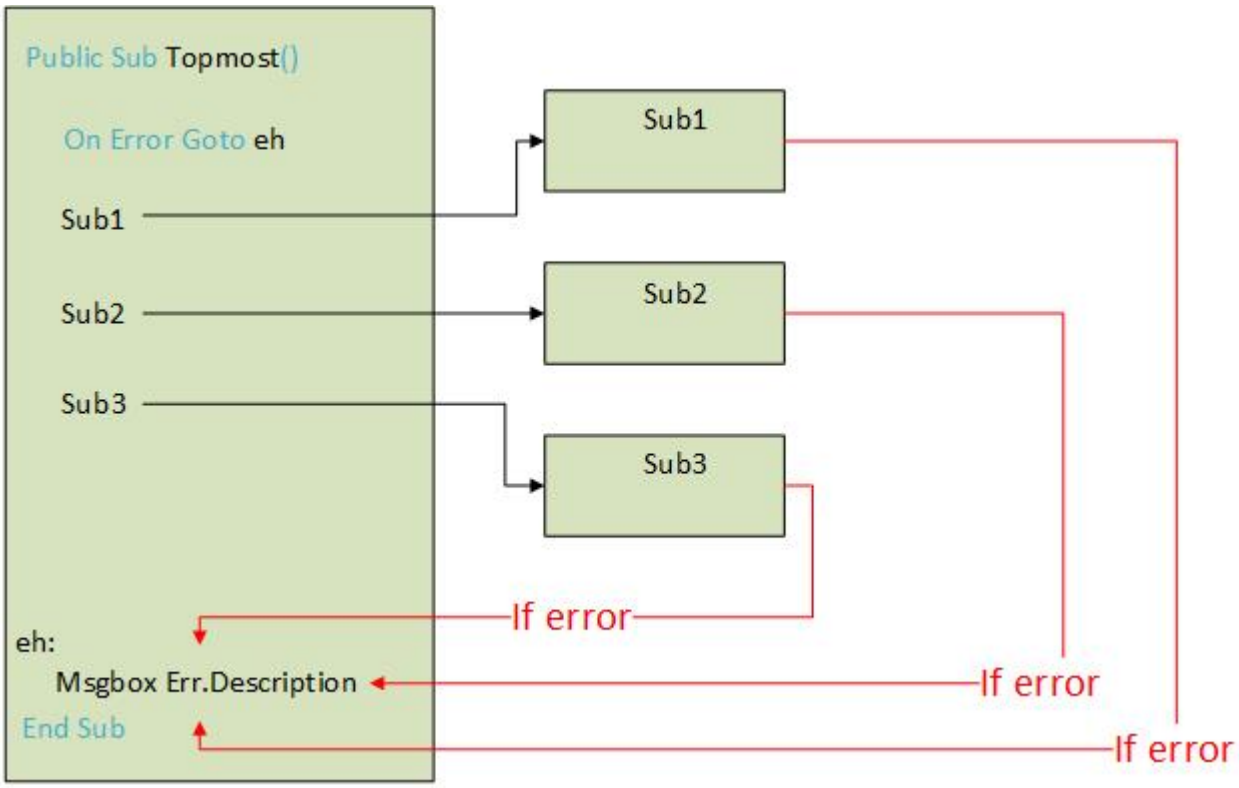
With all the different options you may be confused about how to use error handling in VBA. In this section, I'm going to show you how to implement a simple error handling strategy that you can use in all your applications.

The Basic Implementation

This is a simple overview of our strategy

1. Place the *On Error Goto Label* line at the start of our topmost sub.
2. Place the error handling *Label* at the end of our topmost sub.
3. If an expected error occurs then handle it and continue.
4. If the application cannot continue then use Err.Raise to jump to the error handling label.
5. If an unexpected error occurs the code will automatically jump to the error handling label.

The following image shows an overview of how this looks



The following code shows a simple implementation of this strategy

```
Public Const ERROR_NO_ACCOUNTS As Long = vbObjectError + 514

Sub BuildReport()

    On Error Goto eh

    ' If error in ReadAccounts then jump to error
    ReadAccounts

    ' Do something with the code

Done:
    Exit Sub
eh:
    ' All errors will jump to here
    MsgBox Err.Source & ": The following error occured  " & Err.Description
End Sub

Sub ReadAccounts()

    ' EXPECTED ERROR - Can be handled by the code
    ' Application can handle A1 being zero
    If Sheet1.Range("A1") = 0 Then
        Sheet1.Range("A1") = 1
    End If

    ' EXPECTED  ERROR - cannot be handled by the code
    ' Application cannot continue if no accounts workbook
    If Dir("C:\Docs\Account.xlsx") = "" Then
        Err.Raise ERROR_NO_ACCOUNTS, "UsingErr" _
                , "There are no accounts present for this month."
    End If

    ' UNEXPECTED ERROR - cannot be handled by the code
    ' If cell B3 contains text we will get a type mismatch error
    Dim total As Long
    total = Sheet1.Range("B3")


    ' continue on and read accounts
```

```
End Sub
```

This is a nice way of implementing error handling because

- We don't need to add error handling code to every sub.
- If an error occurs then VBA exits the application gracefully.

A Complete Error Handling Strategy

The strategy above has one drawback. It doesn't tell you where the error occurred. VBA doesn't fill **Err.Source** with anything useful so we have to do this ourselves.

In this section I am going to introduce a more complete error strategy. I have written two subs that perform all the heavy lifting so all you have to do is add them to your project.

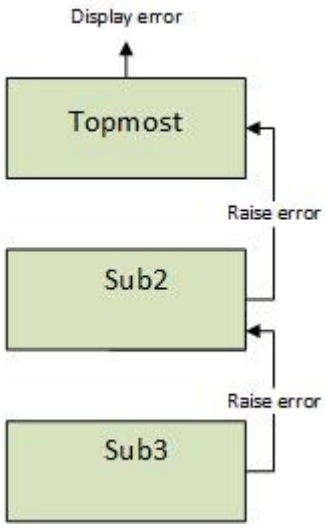
The purpose of this strategy is to provide you with the *Stack** and line number when an error exists.

*The *Stack* is the list of sub/functions that were currently in use when the error occurred.

This is our strategy

1. Place error handling in all the subs.
2. When an error occurs, the error handler adds details to the error and raises it again.
3. When the error reaches the topmost sub it is displayed.

We are simply "bubbling" the error to the top. The following diagram shows a simple visual of what happens when an error occurs in *Sub3*



The only messy part to this is formatting the strings correctly. I have written two subs that handle this, so it is taken care of for you.

These are the two helper subs

```
Option Explicit

' Reraises an error and adds line number and current procedure name
Sub RaiseError(ByVal errorno As Long, ByVal src As String _
                , ByVal proc As String, ByVal desc As String, ByVal lineno As Lon

    Dim sLineNo As Long, sSource As String

    ' Check if procedure where error occurs the line no and proc details
    If src = ThisWorkbook.VBProject.Name Then
        ' Add error line number if present
        If Erl <> 0 Then
            sSource = vbCrLf & "Line no: " & Erl & " "
        End If

        ' Add procedure to source
        sSource = sSource & vbCrLf & proc

    Else
        ' If error has already been raised then just add on procedure name
        sSource = src & vbCrLf & proc
    End If

    ' If the code stops here, make sure DisplayError is placed in the top most Su
    Err.Raise errorno, sSource, desc

End Sub

' Displays the error when it reaches the topmost sub
' Note: You can add a call to logging from this sub
Sub DisplayError(ByVal src As String, ByVal desc As String _
                  , ByVal sProcname As String)

    Dim sMsg As String
    sMsg = "The following error occurred: " & vbCrLf & Err.Description _
                & vbCrLf & vbCrLf & "Error Location is: "

    sMsg = sMsg + Err.source & vbCrLf & sProcname

    ' Display message
```

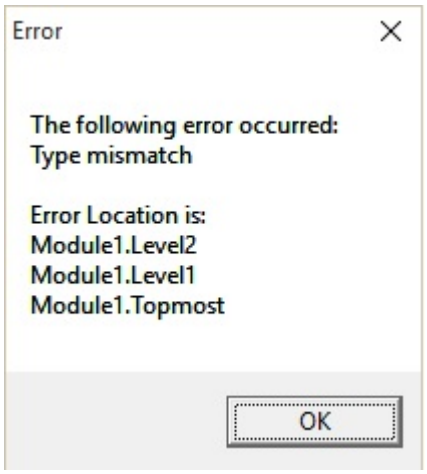
```
    MsgBox sMsg, Title:="Error"

End Sub
```

An Example of using this strategy

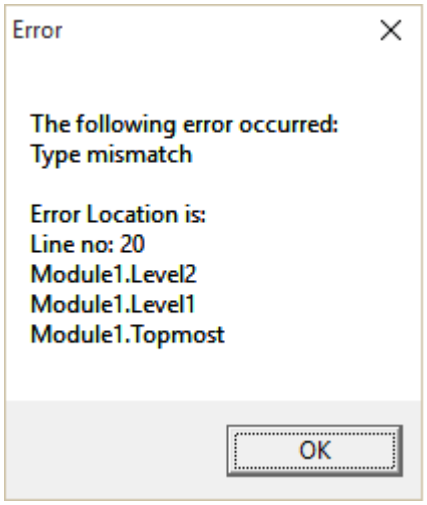
Here is a simple coding that use these subs. In this strategy, we don’t place any code in the topmost sub. We only call subs from it.

```
Sub Topmost()

    On Error Goto EH

    Level1

Done:
    Exit Sub
EH:
    DisplayError Err.source, Err.Description, "Module1.Topmost"
End Sub

Sub Level1()

    On Error Goto EH

    Level2

Done:
    Exit Sub
EH:
   RaiseError Err.Number, Err.source, "Module1.Level1", Err.Description, Erl
End Sub

Sub Level2()

    On Error Goto EH

    ' Error here
    Dim a As Long
    a = "7 / 0"

Done:
    Exit Sub
EH:
    RaiseError Err.Number, Err.source, "Module1.Level2", Err.Description, Erl
End Sub
```

The result looks like this



If your project has line numbers the result will include the line number of the error



Error Handling in a Nutshell

- Error Handling is used to handle errors that occur when your application is running.
- You write specific code to handle expected errors. You use the VBA error handling statement *On Error Goto [label]* to send VBA to a label when an unexpected error occurs.
- You can get details of the error from *Err.Description*.
- You can create your own error using *Err.Raise*.

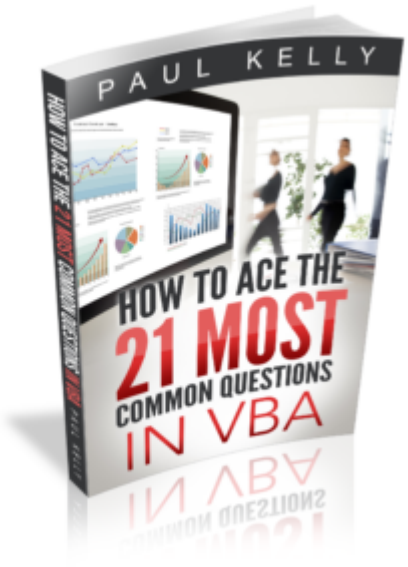
- Using one *On Error* statement in the top most sub will catch all errors in subs that are called from here.
- If you want to record the name of the Sub with the error, you can update the error and rethrow it.
- You can use a log to record information about the application as it is running.

What's Next?

If you want to read about more VBA topics you can view a **complete list of my posts** here (http://excelmacromastery.com/a-quick-guide-to-the-vba-posts/). I also have a free eBook(see below) which you will find useful if you are new to VBA.

If you are serious about mastering VBA then you may want to check out The Excel VBA Handbook (http://www.theexcelvbahandbook.com)

Get the Free eBook



(https://excelmacromastery.leadpages.co/leadbox/14791da73f72a2%3A106f25298346dc/5636318