Handling Errors in Your Access Database #1
Errors 101

Published: 20 September 2018
Author: Martin Green
Screenshots: Access 2016, Windows 10
For Access Versions: 2007, 2010, 2013, 2016

This is not intended to be a complete guide to error handling in VBA but more of an "Errors 101". In it I explain why your code needs error handlers, what they do, and how to use them. There is quite a lot more you can know about error handling and, whilst the same principles apply throughout VBA, the examples here are tailored to Microsoft Access. Most of what you read here you also applies to Excel, Word and in fact any application that can be programmed using VBA, but where appropriate the examples use Access.

Why Worry About Errors?

This may come as a surprise to some of you, but nobody writes perfect code. The aim of any database developer is to create code that is "bulletproof", code that always works as it should and never crashes. We do our best, but we can never anticipate every circumstance or prepare for every crazy thing a user might do. Sometimes we just make a mistake or leave something out. That's where *Error Handling* comes to the rescue.

An error happens when, for whatever reason, Access is unable to execute a code statement. Nothing dramatic happens. The macro just stops. Actually, it pauses, which is part of the problem. It goes into "break mode" and waits for further instructions. If those instructions haven't already been given by the developer, then the user is in trouble. This is why every procedure, from a simple one-line macro on a button click to a complex macro with hundreds of lines of code, needs an error handler.

A macro's error handler tells Access what to do in the event of something going wrong. At the very least, that simple one-line macro that couldn't possibly fail, and anyway it wouldn't matter if it did, should start with the most basic of error handlers:

```
On Error Resume Next
```

This code statement simply tells Access that, if it can't execute a code statement, it should ignore it and move on to the next one. Use it with caution! If a command that would have caused an error is ignored, something later in the same macro might not work correctly. The user will not be notified that there has been a problem and will probably assume that the macro has completed all its tasks successfully.

What Happens if there is No Error Handler?

Consider a simple navigation button on a form, like the *Previous* and *Next* buttons illustrated here (*Fig. 1*).

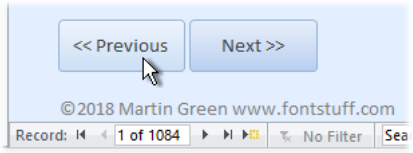

Fig. 1 A "back" button on a form.

Each needs just one code statement to do its job, for example (*Listing 1*):

Listing 1:

```
Private Sub cmdPrevious_Click()
    DoCmd.GoToRecord , , acPrevious
End Sub
```

What could possibly go wrong? Answer: the user might click the *Previous* button when the form is displaying the first record. Crash! Access can't go to the previous record because there isn't one, so it can't execute the code statement and the macro fails. If you haven't anticipated this eventuality and included an error handler, the user sees a standard error message (*Fig. 2*).

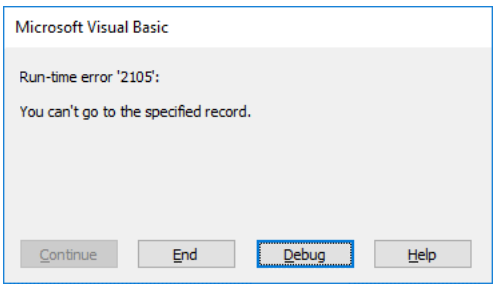

Fig. 2 The built-in error message.

That error message is for you, the developer, to help you test and debug your code, not for the poor user who is wondering what to do next. The *Debug* button is highlighted which we all know means "Click Me!" so that's what the user does and suddenly they find themselves in the Visual Basic Editor looking at your code with a macro in break mode (*Fig. 3*).

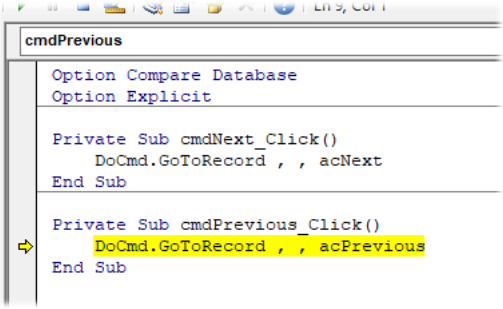

Fig. 3 The Visual Basic Editor showing a macro in Break mode.

They should have clicked the *End* button to terminate the macro but, of course, they didn't know that. Why should they? What happens next is that your phone rings and a voice says: "the database is broken".

Using On Error Resume Next

Would it really matter if the user pressed the *Previous* button on the first record and nothing happened? In this case no, so a simple addition to the code will prevent any problems (*Listing 2*).

Listing 2:

```
Private Sub cmdPrevious_Click()
    On Error Resume Next
    DoCmd.GoToRecord , , acPrevious
End Sub
```

The same applies for the *Next* button. If clicked on the last record Access will move to a new record but if clicked whilst the form is displaying a new record the code will crash.

You should only use this sort of error handler if you are certain that there will be no adverse consequences of ignoring an error. If you aren't sure, then at least remember to password protect your code.

> TIP: Always password-protect your code (in the Visual Basic Editor go to **Tools > Database Properties > Protection**). Just one of the reasons for doing this is that should it ever display the standard error message its Debug button will be disabled.

I have used a very simple example to illustrate when *On Error Resume Next* can safely be used but you can also make use of it within a more complex macro that would normally need a more sophisticated error handler. In this case you can give, and later cancel, the *On Error Resume Next* command by pairing it with the statement *On Error GoTo 0* (*Listing 3*).

Listing 3:

```
' more macro code here
    On Error Resume Next
    DoCmd.GoToRecord , , acPrevious
    On Error GoTo 0
' the macro continues here
```

In the event of such an error, this gives the instruction that, having ignored an error, if another error should occur it now has to refer to the command in Line Zero (although we don't see line numbers in VBA the code engine understands them). In Line Zero, the first line of your macro, it should find your error handler command and therefore proceed to your regular error handler. If you haven't included one the built-in error reporting system comes into effect.

Using a Full Error Handler

When you create any new code, you should try to anticipate what might go wrong. Don't fall into the trap of thinking "why would anyone do that?". You know in what circumstances and how your macros should be used but your database users might not. When testing your code, you should run it in different ways and in different circumstances and note anything that might cause it to fail. If you can't modify your code to avoid such problems arising, then a well-written error handler can help to rescue the situation if the code fails.

Here's a simple example. You have a button on a form that runs a saved query. Simple enough, but what if someone has deleted that query or changed its name (which is why my databases rarely contain saved queries). Without an error handler here's what happens (*Fig. 4*).

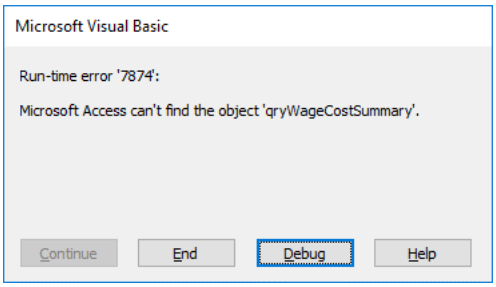

Fig. 4 The macro tries to open a non-existent query.

Again, a simple one-liner and you might be tempted to add just a basic error handler (*Listing 4*):

Listing 4:

```
Private Sub cmdWageCostSummary_Click()
    On Error Resume Next
    DoCmd.OpenQuery "qryWageCostSummary"
End Sub
```

If you do this the query simply won't open. Nothing will happen, and the user will not know why. But unlike in my earlier example, more than one thing could go wrong. Perhaps a table or field was deleted, and you forgot to modify a query that referred to it. A different error would occur with a less helpful error message (*Fig. 5*):

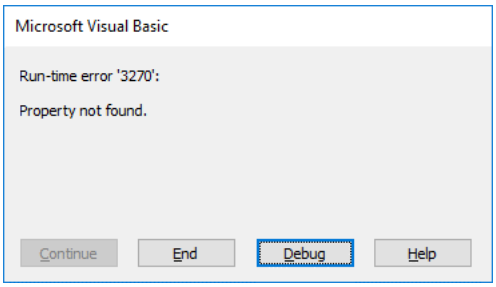

Fig. 5 Sometimes error messages are less helpful.

This simple macro needs a full error handler that lets the user know what the problem is and doesn't leave them in an embarrassing situation. Different developers have their own way of doing things. Here's how I write a full error handler (*Listing 5*):

Listing 5:

```
Private Sub cmdStaffingLevels_Click()
    On Error GoTo cmdStaffingLevels_Click_Err
    ' More code might go here
    DoCmd.OpenQuery "qryStaffingLevels"
    ' More code might go here
cmdStaffingLevels_Click_Exit:
    ' Any tidying-up code goes here
    Exit Sub
cmdStaffingLevels_Click_Err:
    Select Case Err.Number
        Case 3270
            MsgBox "The database is unable to open the query." & vbCrLf & _
                "Please ask the database manager to investigate the problem.", _
                vbCritical, "Error!"
        Case 7874
            MsgBox "The query you requested cannot be found.", vbCritical, "Error!"
        Case Else
            MsgBox "An unexpected error has occurred. If you wish to " & _
                "report the error please note the following:" & vbCrLf & _
                "Error Number: " & Err.Number & vbCrLf & _
                "Description: " & Err.Description, vbCritical, "Error!"
    End Select
    Resume cmdStaffingLevels_Click_Exit
End Sub
```

How the Error Handler Works

The code contains two bookmarks. One marks the start of my Error Handler, the other the start of the Exit Routine. Some developers simply name their bookmarks "Error Handler" and "Exit Routine" but as you can see (*Listing 5*) I prefer to include the name of the macro with the suffix "_Err" or "_Exit" as appropriate.

> NOTE: A bookmark in code is a word followed by a colon (**:**). The bookmark is not an executable code statement. It merely marks a specific place in the macro. Usual naming rules apply. When you type the name of your bookmark and enter the colon the text becomes a bookmark and moves to the left margin automatically.

The instruction indicating what to do in the case of an error should always be the first line of the macro. It takes the form of a *GoTo* statement to send the code engine to the error handler bookmark and instructs it to proceed from there. The error handler should be located at the end of your macro.

I use a *Case Statement* to deal with any known errors that my testing has highlighted and that I have not been able to prevent within the macro itself. Sometimes you might need to include more code here to deal with a particular circumstance but often all you can do is show an explanatory message as in these examples (*Fig. 6*).

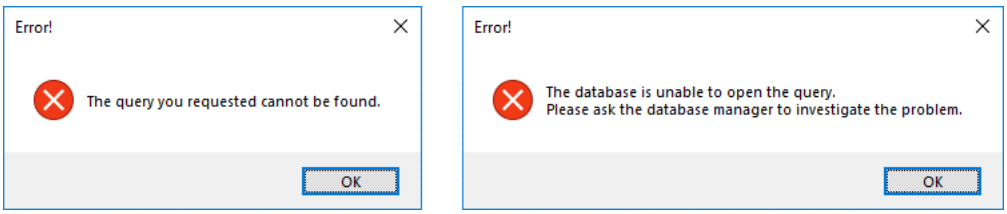

Fig. 6 The Error handler displays explanatory messages.

Each *Case* refers to a known error by its number. If you discover any further possible errors, you can return and simply add additional *Cases* to the *Case Statement*. The *Case Else* part of the Case Statement, meaning "for anything else…", is used to deal with any error that you had not anticipated. It makes use of the *Error Number* and *Error Description* that are available to you when an error occurs (*Fig. 7*). If the user makes a note of this information, or if you have included an error log in your database, it will help you diagnose and deal with the problem.

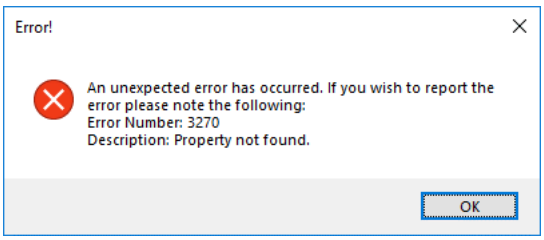

Fig. 7 A general error message.

When you handle an error yourself Access will not ever display the standard error message so there is no danger that the user will find themselves in the Visual Basic Editor. Written properly, it also ensures that the macro is safely terminated. This is very important because when code is in break mode the user is often unable to work in the host program.

Using an Exit Routine

When adding an error handler to your code you must make sure that, before the error handler's bookmark, you add the statement:

```
Exit Sub
```

This works like *End Sub*. Its purpose is to make sure that if an error has not occurred, the macro terminates at this point and does not continue into the error handler. This alone satisfies some developers, but I prefer to include it in an *Exit Routine* where any additional "tidying-up" tasks can be included.

The last statement of the error handler e.g. *Resume <Macroname>_Exit* tells the code engine that, after carrying out the necessary tasks in the *Error Handler*, it must move up to the *Exit Routine* bookmark and continue from there.

The "tidying-up" tasks could include several things, depending on what your macro was doing. Here are some examples:

- Any object variables such as a database or recordset that were declared using the *Set* keyword will need to be set to *Nothing* to release them from memory, for example:
  ```
  Dim db As DAO.Database
  Dim rst As DAO.Recordset
  ```
  Object variables need to be "unset" using statements like:
  ```
  Set db = Nothing
  Set rst = Nothing
  ```

- If you have disabled user confirmation messages for a specific action such as deleting a record or otherwise manipulating a recordset you should reinstate them. The command…
  ```
  DoCmd.SetWarnings False
  ```
  is cancelled by the statement…
  ```
  DoCmd.SetWarnings True
  ```

- It is particularly important to remember to switch on screen updating if at any point you turned it off for speed or to prevent the user seeing the macro's workings on screen by setting:
  ```
  DoCmd.Echo False
  ```
  Switch screen updating on again using…
  ```
  DoCmd.Echo True
  ```
  If you fail to do this and your macro crashes with screen updating switched off your user will be left with a blank screen so this one is really important!

Of course, you aren't going to know at which point your macro failed so would it cause a problem if your *Exit Routine* tries to "unset" an object variable that had not yet been "set"? Switching-on screen updating that was already on, or reinstating warnings that had not been disabled (in fact anything that is set/unset using *True/False*) will not cause a problem but trying to "unset" an object variable that had not yet been "set" probably will.

For this reason, it is safest to start your *Exit Routine* with the error command *On Error Resume Next*. A comprehensive *Exit Routine* for Access would like something like this (*Listing 6*):

Listing 6:

```
    ' Your macro code finishes here
cmdExportData_Click_Exit:
    On Error Resume Next
    Set rst = Nothing
    Set db = Nothing
    DoCmd.SetWarnings = True
    DoCmd.Echo True
    Exit Sub
cmdExportData_Click_Err:
    ' Your Error Handler begins here
```

Summary

Try as we might, we can't anticipate every circumstance in which our code will be used, nor can we foresee everything a user might do no matter how irrational or unexpected. Errors happen. Your task as a developer is to make sure that the unexpected doesn't turn into a disaster. Follow these rules to make your code safe and reliable:

- Add an **Error Handler** to **every** macro.
- Only use **On Error Resume Next** if you are certain that it is safe to ignore an error.
- Use **On Error GoTo 0** to cancel *On Error Resume Next* when used in a larger macro.
- Test your code and use a **Case Statement** to help you handle different kinds of errors.
- Make use of an **Exit Routine** to safely clean-up after an error.
- **Password Protect** your code to prevent problems in the event of unhandled errors.