Handling Errors in Your Access Database #2
Add an Error Log to Your Database

Published: 24 September 2018
Author: Martin Green
Screenshots: Access 2016, Windows 10
For Access Versions: 2007, 2010, 2013, 2016

Despite all our best efforts there is always the possibility of an error happening when our code is run. A competent developer will always ensure that their code is properly protected by including an Error Handler in each macro so that, in the event of the code crashing, the Error Handler takes control and deals with the situation safely, with the minimum of disruption to the user.

In the first part of this tutorial *Handling Errors in Your Access Database #1 Errors 101* (http://www.fontstuff.com/access/acctut22.htm) I showed how I deal with errors in my databases. With a known error (i.e. one that is anticipated but not preventable) my error handler displays a tailored message to the user explaining what has happened. When the error is not anticipated the error handler displays a general error message showing the *Error Number* and *Error Description* (*Fig. 1*) both of which are available through VBA. Where possible, the Error Handler will also attempt to rescue the situation to avoid damage to the database or its data.

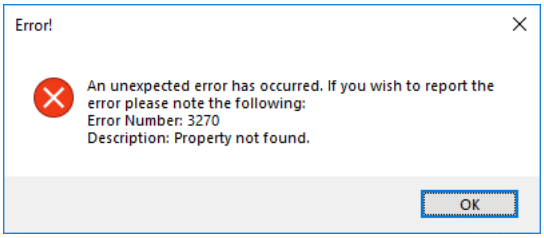

Fig. 1 A general error message.

Hopefully, should an error occur, whether anticipated or not, the user will note the details and report it so that action can be taken to rectify the problem. Unfortunately, this seldom happens. Usually there is a message like "the database keeps crashing" or "I keep getting an error message". More savvy users will note the details or even take a screenshot of the error message, which is very helpful, but diagnosing the problem will often require more information.

Additional information would be helpful, such as: Does this only happen to a particular user? Has it happened before and if so how often? Does it always happen on the same computer? What was the user doing at the time? To answer some of these questions and to make the reporting of errors easier, I always include an Error Log in my databases.

I have even discovered that users have failed to report an error, later saying something like "Oh yes, it does that…" as if they expected things to go wrong occasionally when, had I known about the problem, I could have fixed it.

What the Error Log Does

The Error Log has three main components:

- **A Table** in which to record details of an error.
- **A Macro** to gather information about the error and write it into the table.
- **A Call** in each error handler to activate the macro when an error occurs.

In addition to installing an error log in my databases I also incorporate a simple tool by which a copy of the log can be exported to, for example, Excel or can be sent to me or some other nominated person by email.

I create error handlers in my usual way, with the addition of a code statement that calls (activates) the error reporting macro, at the same time passing to it any relevant information such as the name of the macro that failed and, if appropriate, the form in which it was located. Other important details such as the Error number, Error Description, Time and Date, User Name etc. can be collected from the system.

Having collected the required information, the error reporting macro writes it into the table. Instead of relying on the users to remember the details, I can examine the error log and get the information from there.

> NOTE: Don't rely entirely on the Error Log to help you diagnose a problem. A user might have a valuable comment or observation that couldn't be picked up by the system, so it's important to listen to what they have to say!

There are four steps to build the Error Log and its reporting system.

Step 1: Build the Error Log Table

Create a new table and add as many fields as you think you will need. I usually use those shown below (*Table 1, Fig. 2*). You may wish to add more information, or collect less. If the database requires its users to Log-on I will add that information too if I can. I usually name my table *tblErrorLog*.

Table 1: Fields in the Error Log table.

Field Name	Data Type
ErrorLogID	AutoNumber (Primary Key)
DateTime	Date/Time
UserName	Short Text
ComputerName	Short Text
MacroName	Short Text
ErrorNumber	Number (Long Integer)
ErrorDescription	Short Text

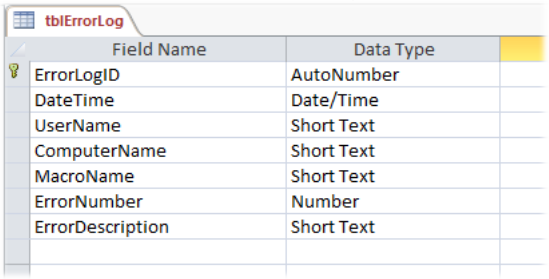

Fig. 2 The Design View of the Error Log table.

Step 2: Write the Error Reporting Macro

The error reporting macro takes the form of a procedure that requires a single parameter, the name of the macro in which the error occurred. You will see how that information is supplied in the next section.

If you don't already have something suitable you will have to create a standard module in which to place the macro. I am offering two options for the code, one using ADO to write to the Error Log table, the other using SQL. I have no particular preference. Here's the ADO version (*Listing 1*):

Listing 1:

```
Sub ReportError(MacroName As String)
' Error handling and logging routine
    Dim cnn As New ADODB.Connection
    Dim rst As ADODB.Recordset
    MsgBox "ERROR!" & vbCrLf _
        & vbCrLf & "An unexpected error has occurred." _
        & vbCrLf & "Details of the error have been logged. " _
        & "If you wish to report the error please quote the following information:" _
        & vbCrLf & "Procedure: " & MacroName _
        & vbCrLf & "Error Number: " & Err.Number _
        & vbCrLf & "Description: " & Err.Description _
        , vbCritical, "ERROR!"
    Set cnn = CurrentProject.Connection
    Set rst = New ADODB.Recordset
    rst.Open "SELECT * FROM tblErrorLog", cnn, adOpenKeyset, adLockOptimistic
    With rst
        .AddNew
        ![DateTime] = Now
        ![UserName] = Environ("USERNAME")
        ![ComputerName] = Environ("COMPUTERNAME")
        ![MacroName] = MacroName
        ![ErrorNumber] = Err.Number
        ![ErrorDescription] = Err.Description
        .Update
        .Close
    End With
    Set rst = Nothing
    Set cnn = Nothing
End Sub
```

How the ADO Code Works

When using ADO (ActiveX Data Objects) code, depending which version of Access you are using, you might need to set a reference to the ActiveX object library. If you aren't sure whether or not you need to do this try compiling the code (**Debug > Compile...**). If the compiler displays an error on the ADODB variable declaration then you need to set the reference and compile again.

> TIP: To set a reference to ADO, in the Visual Basic Editor go to **Tools > References**. In the *References* dialog find and check *Microsoft ActiveX Data Objects 2.8 Library* then click **OK**.

Between the parentheses that follow the macro's name (I've named the macro *ReportError*) is the name and data type of the required parameter which I've named *MacroName*. This information is supplied when the macro is "called" by the error handler.

This example uses ADO code to add a new record to the *tblErrorLog* table. Alternatively, I could have constructed a SQL statement and used SQL to add the record. SQL is often faster than ADO but when dealing with a single record it makes no difference.

The first two statements declare the variables that represent the ADO connection and the ADO recordset that will be opened to receive the new data. These are followed by the standard error message I show the user. You can add whatever wording you like here.

Two statements then add values to the previously declared variables, citing the current database and the recordset which opens the Error Log table into memory.

The *With Statement* contains a series of commands which add a new record to the table, populate its various fields, then save the record and close the recordset. A final pair or commands clear the two object variables.

The SQL method has the advantage that it does not require a reference to ADO. Otherwise both methods work equally well. Here's how I do it using SQL (*Listing 2*):

Listing 2:

```
Sub ReportError(MacroName As String)
' Error handling and logging routine
    Dim strSQL As String
    MsgBox "ERROR!" & vbCrLf _
        & vbCrLf & "An unexpected error has occurred." _
        & vbCrLf & "Details of the error have been logged. " _
        & "If you wish to report the error please quote the following information:" _
        & vbCrLf & "Procedure: " & MacroName _
        & vbCrLf & "Error Number: " & Err.Number _
        & vbCrLf & "Description: " & Err.Description _
        , vbCritical, "ERROR!"
    strSQL = "INSERT INTO tblErrorLog ([DateTime], [UserName], [ComputerName], " & _
             "[MacroName], [ErrorNumber], [ErrorDescription]) VALUES (" & _
             Chr(35) & Format(Now, "MM/DD/YYYY hh:nn") & Chr(35) & ", " & _
             Chr(34) & Environ("USERNAME") & Chr(34) & ", " & _
             Chr(34) & Environ("COMPUTERNAME") & Chr(34) & ", " & _
             Chr(34) & MacroName & Chr(34) & ", " & _
             Err.Number & ", " & _
             Chr(34) & Err.Description & Chr(34) & ");"
    DoCmd.SetWarnings False
    DoCmd.RunSQL strSQL
    DoCmd.SetWarnings True
End Sub
```

How the SQL Code Works

My code creates a SQL statement and stores it in a variable before executing it so the code starts with a variable declaration for the string that will represent the SQL. The usual message box follows, then the building of an *Insert Into* SQL statement. Access creates a SLQ statement like this when you build an *Append Query* in the Access Query Builder.

SQL requires the use of *Data Qualifiers* when a SQL statement refers to data values. A text value must be enclosed in quotes (‘ or “), a date/time value in hash marks (#), whilst numerical values don’t require anything. Since I am writing my SQL in VBA (which also requires the use of data qualifiers) the SQL statement, itself being a text string, needs to be enclosed in quotes too. Trying to put quotes inside quotes can lead to problems so to avoid this I use the *Chr()* function to generate the characters I need. A quote mark is created by *Chr(34)* whilst *Chr(35)* generates a hash mark.

Having created a suitable SQL statement the code then uses the *DoCmd.RunSQL* command to execute the SQL. Since all “action” queries cause Access to prompt the user for permission, I have enclosed the *DoCmd.RunSQL* command in a pair of *DoCmd.SetWarnings* commands to deactivate then reinstate the prompts. We don’t need to ask the user’s permission to report the error, nor do we want them to say “No” if asked.

Step 3: Add a Call Statement to Your Error Handlers

One of the pieces of information that the error reporting macro notes is the name of the macro in which the error occurred. This is supplied as a parameter when the error reporting macro is “called”. Although not absolutely necessary I use the keyword *Call* when calling one macro from another since it makes it clear to anyone reading the code exactly what the code is doing e.g.:

```
Call ReportError(<Macro Name Goes Here Enclosed in Quotes>)
```

The parameter takes the form of a text string so it must be enclosed in quotes. It can be anything you like as long as it accurately defines the macro you want to report. If the reported macro is a *Private Sub* on a form, such as a button click, I also add the name of the form to the parameter since procedures on different forms might have the same name. For example a macro named *cmdWageCostSummary_Click* on a form named *frmHome* would be reported with:

```
Call ReportError("frmHome_cmdWageCostSummary_Click")
```

In the first part of this tutorial I gave an example of errors that could arise from a macro that opened a stored query, together with a sample of code showing how I create an error handler that dealt with the various types of error that could occur (see: http://www.fontstuff.com/access/acctut22.htm). Here’s how I would amend that error handler when using the Error Log (*Listing 3*):

Listing 3:

```
Private Sub cmdWageCostSummary_Click()
    On Error GoTo cmdWageCostSummary_Click_Err
    DoCmd.OpenQuery "qryWageCostSummary"
cmdWageCostSummary_Click_Exit:
    Exit Sub
cmdWageCostSummary_Click_Err:
    Call ReportError("frmHome_cmdWageCostSummary_Click")
    Resume cmdWageCostSummary_Click_Exit
End Sub
```

Note that I have not included a Case Statement for dealing individually with different kinds of error. You might choose to do this or not, depending on the circumstances. You may decide that some types of error simply need to be explained to the user by means of a message, whilst others need to be logged as well.

When called the Error Log macro writes the information it has gathered into the Error Log table (*Fig. 3*).

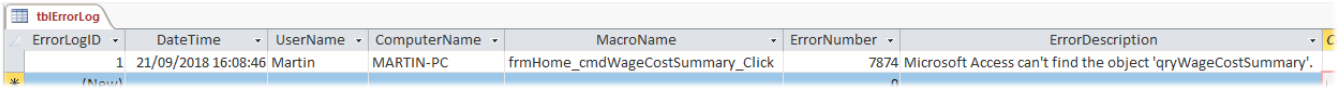

Fig. 3 An Error recorded on the Error Log table.

Step 4: Viewing the Error Log

I like to offer a few choices for gaining access to the Error Log and usually present these on a *Database Maintenance* screen along with other tools useful to the users. In this example I give three different ways to view or share the log (*Fig. 4*).

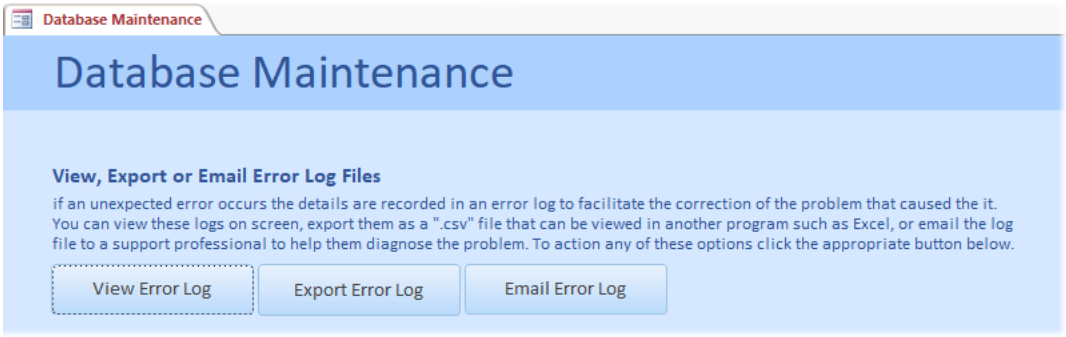

Fig. 4 Offering the facility to view the Error Log.

View the Error Log Table

This option example simply opens the table. I don't normally allow users direct access to tables so when opening the Error Log table I also make it read-only (*Listing 4*).

Listing 4:

```
Private Sub cmdViewErrorLog_Click()
    On Error Resume Next
' Open Error Log table read-only
    DoCmd.OpenTable "tblErrorLog", , acReadOnly
End Sub
```

Export the Error Log

Access VBA allows data to be exported from a table or query in various different formats. I prefer the flexibility of a .csv file which can easily be read by Excel and other programs.

My code (*Listing 5*) constructs a filename which it stores temporarily in a string variable. The filename includes a timestamp accurate to the second. This avoids an error in the event of the user running the macro a second time when, otherwise, it would attempt to create multiple files with the same name. The variable string also includes the full path to the user's desktop, obtained with the help of the *Environ()* function. You could export elsewhere, for example to the user's Documents folder, or even ask the user to choose a location. I decided to keep it simple.

Listing 5:

```
Private Sub cmdExportErrorLog_Click()
    On Error GoTo cmdExportErrorLog_Click_Err
' Export Error Log as *.csv file to user's desktop
    Dim strFilename As String
    strFilename = Environ("UserProfile") & "\Desktop\" & _
        "ErrorLog_" & Format(Now, "yymmddhhnnss") & ".csv"
    DoCmd.TransferText acExportDelim, , "tblErrorLog", strFilename, True
    MsgBox "The Error log file has been exported to:" & vbCrLf & _
        strFilename, vbInformation, "Export complete"
cmdExportErrorLog_Click_Exit:
    Exit Sub
cmdExportErrorLog_Click_Err:
    Call ReportError("frmDatabaseMaintenance_cmdExportErrorLog_Click")
    Resume cmdExportErrorLog_Click_Exit
End Sub
```

The string variable holding the export filename and path is used both to perform the export and to subsequently notify the user via a message box (*Fig. 5*). I have use the *DoCmd.TransferText* command to export the entire table. You could add the facility to export a specific date range, and create a query to export instead of the whole table.

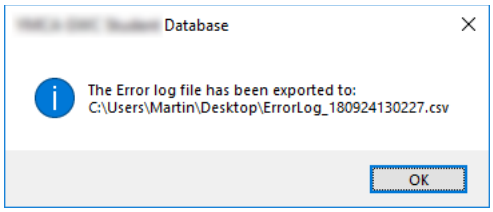

Fig. 5 A message confirms that the Log has been exported.

Email the Error Log

Most useful for the developer is the ability for the user to email the error log to them. My code (*Listing 6*) uses the *DoCmd.SendObject* command which employs the user's default email client and doesn't require any additional programming (e.g. for Microsoft Outlook). In this example the code sends the entire table but, as I suggested earlier, you could send a query specifying a chosen date range.

The *DoCmd.SendObject* command includes several options. I have hard-coded a recipient email address. If this argument is left blank Access prompts the user for a recipient when the macro is run. The code also allows for Cc and Bcc to be specified. The parameter *True* in this example gives the user the opportunity to edit the email message before sending by opening it in the default email program (*False* would send the message straight away).

> NOTE: If you offer the user the opportunity to edit the message by setting the *EditMessage* parameter to *True*, an error would occur if the user subsequently cancelled the message. You can see in my example (*Listing 6*) that this eventuality is accounted for in the macro's Error Handler.

Listing 6:

```
Private Sub cmdEmailErrorLog_Click()
    On Error GoTo cmdEmailErrorLog_Click_Err
' Email a copy of the Error Log file
    DoCmd.SendObject acSendTable, "tblErrorLog", acFormatXLS, & _
        "martin@fontstuff.com", , , "Acme Database Error Log", & _
        "See Error Log attached.", True
cmdEmailErrorLog_Click_Exit:
    Exit Sub
cmdEmailErrorLog_Click_Err:
    Select Case Err.Number
        Case 2501
            MsgBox "You cancelled the email message.", vbInformation, "Cancelled"
        Case Else
            Call ReportError("frmDatabaseMaintenance_cmdEmailErrorLog_Click")
    End Select
    Resume cmdEmailErrorLog_Click_Exit
End Sub
```

Appendix: The Environ() Function

You will have seen that, on several occasions, I have made use of the VBA *Environ()* function to gather information about the user and the environment in which they were working. This function requires a single argument which can be a text string or a number. I recommend that you use the Text string rather than its index number because anyone later reading your code will understand how it is being used. Some common examples are shown here (*Table 2*):

Table 2: Examples of the Environ() function.

Environ(Argument)	Returns
`Environ("USERNAME")`	The current user's Windows login name. e.g. **JohnDoe**
`Environ("USERPROFILE")`	The path to the current user's User folder e.g. **C:\Users\JohnDoe**
`Environ("COMPUTERNAME")`	The name of the host computer e.g. **ADMIN-PC4**
`Environ("TEMP")`	The path to the current user's Temp folder e.g. **C:\Users\JohnDoe\AppData\Local\Temp**

In addition to saving information into the Error Log you can use the *Environ()* function to specify the path to the user's Desktop or Documents folders, for example:

```
Environ("UserProfile") & "\Desktop\"
```

Would construct a path like:

```
C:\Users\JohnDoe\Desktop\
```

To allow you to perhaps save a file to the current user's, desktop as I did in an earlier example (*Listing 5*).

The *Environ()* function is available to all your VBA programs. Use this simple macro (*Listing 7*, *Fig. 6*) in any of your programs to create a list of its various arguments. The macro uses a loop to send a list of examples of the 37 available arguments to the Visual Basic Editor's Immediate Window (open it via **View>Immediate Window** or Keys: **[Ctrl]+G**).

Listing 7:

```
Sub ListEnviron()
    Dim i As Integer
    For i = 1 To 37
        Debug.Print Environ(i)
    Next i
End Sub
```

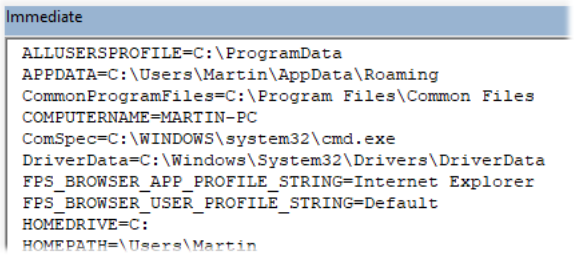

Fig. 6 Environ() arguments sent to the Immediate Window.

Alternatively, use this macro (*Listing 8*, *Fig. 7*) to write the list to an empty worksheet in Excel:

Listing 8:

```
Sub ListEnviron()
    Dim i As Integer
    For i = 1 To 37
        Range("A" & i).Value =
Environ(i)
    Next i
End Sub
```

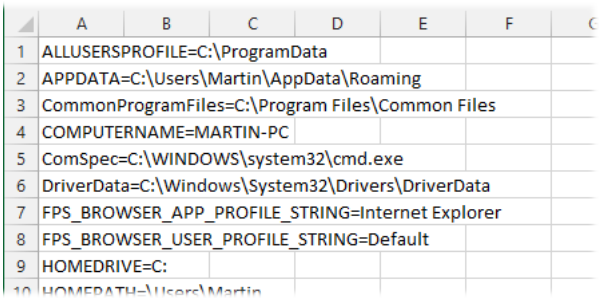

Fig. 7 Environ() arguments written to an Excel worksheet.