vb123.com

Smart Access

Solutions for Microsoft® Access® Developers

Error Trapping with a Code Builder

Keith Bombard

97 2000 2002 DOWNLOAD

If you code for a living, you know that error trapping is a drag. But it's also an essential part of any serious Access development project. Here Keith Bombard introduces a new tool to automatically insert error-handling code into your Access applications.

WHY is this article so important to you? If you perform error trapping on as much code as I do (my last Access application had 950 error traps), error trapping becomes critically important. Error trapping imparts professionalism, structure, and (arguably) a level of safety into your production data environment. The presentation of precise, considerate, and unambiguous Situation Notification messages goes a long way toward establishing and maintaining user confidence. In my experience, users crave reliability and consistency of format, and my clients will pay for the additional time spent to do it right, especially if I take the time to explain the benefits to them.

Of course, I didn't hand-type 950 error traps into that application. I used a code builder to quickly drop pre-formatted error trapping text into event procedures, subroutines, and functions.

Anatomy of an error trap

An error trap routine consists of three parts:

- *The heading*—Where comments and the On Error Goto statement are found.
- *The middle section*—Left blank for you to put your code in.
- *The bottom section*—Where the error is trapped and an error message is displayed.

Continues on page 4

November 2002

Volume 10, Number 11

1 **Error Trapping with a Code Builder**
Keith Bombard

3 **Editorial: Moving to the Web**
Peter Vogel

8 **Company Profile: Superior Software for Windows (SSW)**
Danny J. Lesandrini

10 **Product Review: Access Reporter .NET for IIS**
Danny J. Lesandrini

13 **"Just-In-Time" Queries**
Frank Kegley

16 **No More Write-only Code**
Peter Vogel

20 **November 2002 Source Code**

95 Applies to Access 95 | 97 Applies to Access 97 | 2000 Applies to Access 2000 | 2002 Applies to Access 2002

DOWNLOAD

www.vb123.com/smart

In code, an underscore (_) as the last character of a line indicates that the line has been wrapped for layout purposes. In Access 95 and up you can use the code as it appears, but in Access 2.0 you must recombine the wrapped lines.

increase they saw.

Of course, with each user retrieving hundreds of records to the Web server, the application had substantial scalability problems. We'll pass over that problem in silence.

The right thing to do would have been to rewrite the Access application to retrieve only the records that the user wanted. The cost of modifying the Access application would probably have been measured in hours. But, now, they had a Web application. ▲

Error Trapping...

Continued from page 1

My error handling code expects to be passed the module name, the procedure name, and the procedure type (form, report, or global module), but that's taken care of now by the builder, sparing me the trouble of typing that information (that's the main benefit of using a builder—you don't need to type anything). Passing information from the calling program helps me to identify exactly where the error has occurred. This sounds so basic that you're probably wondering why I mention this. You'd be surprised how many applications I've seen where errors are trapped and only the standard Err.Description property gets displayed. When a user calls to tell you about the error armed only with this information, you have no clue where to look. It's always a good idea to provide as much information as possible to the user in an error message.

The builder

Let's assume that you agree with me about error trapping, and let's also assume that you've searched high and low and just can't find a suitable (read: no cost) add-in code builder that works in Access 2000/2002. I couldn't find one either, which brings me to this article's *raison d'être*: how to create an Access code module error-trap builder for Access 2000. For simplicity, I'll refer exclusively to Access 2000, but the builder will also work in Access 2002. Try the code out and use it with my blessing.

This adventure was precipitated by my employer's requirement to upgrade several very large Access 97 applications to Access 2000. I had a very nice set of shareware add-ins that worked fine in the 97 world, but wouldn't work at all in Access 2000. I really needed a code builder to continue development in 2000—and I hate to purchase software when I can write it myself. In my spare time I came up with this tool.

The technique that I use builds on a new feature found in Access 2000: Code Macros. A Code Macro is a subroutine (not a function) that you can call directly from any VB code module Edit panel. For those of you who aren't familiar with this feature, you can find a description in the Help system under the Macros sub-choice of the Tools menu in the code window. If you run a Code Macro from the Tools menu, the subroutine simply runs. This mechanism turned out to be perfect for this code builder application.

The Access 97 version of this routine (also included in this month's Download file, which is available at www.smartaccessnewsletter.com) uses an Autokeys macro (attached to the F12 key) to call a similar error trap function. However, I soon discovered that Autokeys macros won't work in the Access 2000 code window. Code Macros work around this limitation quite easily.

How it works

The code inserts strings from the table tblErrorTrapMaster directly into your code module. Error trap string data is stored in the memo field CodeString in the table. The text in each CodeString record represents one complete error trap that will be inserted as one piece into your procedure. CodeString records have two key fields: CodeKey (more on this shortly) and FindString.

To implement the error trap insertion into a module, you just have to perform two steps:

1. Enter the appropriate FindString text into the new module code where the error trap code should be inserted.
2. Execute the A_ETrap Code Macro from the Tools | Macros code window menu.

Behind the scenes, the code builder searches the current open module and if a FindString identifier is found, that line is deleted and the text in the corresponding CodeString field is inserted. Some key substitutions are made to the CodeString text just before the insertion. These changes are based on a setup form where the module name, procedure name, and procedure type are substituted into the CodeString text. The cursor is then positioned properly into the middle of the routine (using the venerable SendKeys).

This technique allows you to have several error traps defined and choose which one you want to use. You select which error trap to add to your code by inserting the appropriate FindString text into your module code.

The call to the A_ETrap Code Macro begins by defining the following variables:

```
Dim MDL As Module
Dim StartLine As Long
Dim EndLine As Long
Dim StartCol As Long
Dim EndCol As Long
Dim OType As Variant
Dim DateStr As String
Dim ProcType As String
Dim NoOfLines As Long
```

```
    Dim NoOfFinds As Long
    Dim Findstr As String
    Dim Db As DAO.Database
    Dim rs As DAO.Recordset
    Dim MName As Variant
    Dim NumOfModules As Long
    Dim i As Long
```

The code first tests to make sure that there are active FindStrings to search for. The value of CodeKey in the sample table is currently set to "CS1". If you're coding in different venues / applications and you need to insert different looking error trap strings into your code, add those strings to tblErrorTrapMaster and set CodeKey to an alternate value to differentiate between those strings. The code then checks for the number of open modules and reads the error trapping table:

```
NoOfFinds = DCount("*", "tblErrorTrapMaster", _
      "CodeKey = 'CS1' and Active = True")

If NoOfFinds > 0 Then
    NumOfModules = Application.Modules.Count
    Set Db = CurrentDb
    Set rs = Db.OpenRecordset( _
        "Select * from tblErrorTrapMaster  " & _
        "Where CodeKey = 'CS1' and Active = True", _
         dbOpenSnapshot)
```

The routine now loops through the open modules using the Modules collection, searching and finding the first open module with the first FindString string:

```
For i = 0 To NumOfModules - 1
   rs.MoveFirst
   MName = Modules(i).Name
   OType = Modules(i).Type
```

While you may not be aware of it, the modules that you work with in Access can be manipulated as objects. As my code demonstrates, you can hold references to code modules in object variables. The Application object also has the Modules collection that I'm using to process all of the code in my application. The properties of the module object, like the ProcBodyLine method, let you determine the relationships between procedures and positions in the module.

The next step moves the selected module into a variable for use later in the module:

```
Set MDL = Modules(MName)
```

Now the code tries to find any of the FindString string markers in open module code. I loop through the recordset of active FindStrings until I get a hit in the module:

```
Do While Not rs.EOF
    Findstr = rs!FindString
    glbVisibleSetupForm = rs!VisibleSetupForm
```

The module's Find method does all the work. The Find method will return True if FindStr is found in the module that it's currently processing. The found line number is reported back in StartLine. In the following code, if the FindString is found, execution drops through the For loop and continues:

```
      If MDL.Find(Findstr, StartLine, StartCol, _
          EndLine, EndCol) = True Then
        Exit For
        End If
      rs.MoveNext
    Loop
  Next
  rs.Close
  Set Db = Nothing
End If
```

If no string is found, the routine reports this back to you and exits:

```
    If StartLine = 0 Or NoOfFinds = 0 Then
      MsgBox "Sorry, a valid String Marker was " & _
       "not found or the Error-trap String " & _
       "table records are inactive. " & _
       "Please place a String Marker into your " & _
       "module first.", _
       vbOKOnly + vbInformation, MsgHeader
    End If
  Exit Sub
End If
```

Now that a string is found, a predefined set of Global variables are initialized to pass the settings to the Add Header Setup Form. The variable glbProcKind is initialized also, using the Module object's ProcOfLine method (this returns the procedure type at the CodeString's line position):

```
glbProcName = MDL.ProcOfLine(StartLine, glbProcKind)
glbModName = MName
glbObjectType = OType
glbFindString = Findstr
glbFoundLine = StartLine
Set glbModule = MDL
```

The setup form

With all the data gathered, the setup form is opened and passed the current object type. The setup form is a view-only form used to display the information to be inserted into your code.

I used the OpenArgs variable to pass the data to the form to prevent users from opening this form from the Database window (and getting a bunch of errors if they do). I also use the variable glbVisibleSetupForm here. A True in this variable enables the display of the Error-Trap setup form just prior to the code insertion. If this variable is False, then the setup form opens in hidden mode and module code is inserted without displaying the form. Whether the setup form is displayed is, in the end, controlled by the field VisibleSetupForm in tblErrorTrapMaster, which sets the glbVisibleSetupForm variable.

```
If glbVisibleSetupForm = True Then
    DoCmd.OpenForm "frmAddErrorHeader", , , , , _
            acDialog, "A_ETrap_Caller"
Else
    DoCmd.OpenForm "frmAddErrorHeader", , , , , _
```

```
            acHidden, "A_ETrap_Caller"
    Forms!frmAddErrorHeader.CmdAddit_Click
End If
```

The load event of the setup form sets the stage for the insertion of the error trap. Most of the variables are initialized from the variables passed to the form:

```
Dim LineNo As Long
Dim ProcType As String
Me.ModName = glbModName
Me.ProcName = glbProcName
Me.FindString = glbFindString
Me.FoundLine = glbFoundLine
```

At this point, the object type of the procedure is used to set a radio button on the form to indicate the type of module:

```
Me.optType = 1
Select Case glbObjectType
   Case 3
      Me.optType = 2
   Case 5
      Me.optType = 3
End Select
```

Now I pass the procedure name and type (GlbProcName and glbProcKind) in another call to ProcOfLine to get the procedure's line number:

```
LineNo = glbModule.ProcBodyLine( _
         glbProcName, glbProcKind)
```

Here's an interesting twist that complicated matters a bit—to determine the kind of procedure, I use the Access 2000 hidden class object ProcKind. This class isn't hidden in Access 97. To see the members in this class, go to the object browser and turn on the Show Hidden Members option using the right mouse shortcut menu. If you do a search for ProcKind, you'll see its methods, properties, and events. Access 2000 Help doesn't appear to have been updated from the Access 97 world: The Access 97 VBA constants vbext_pk_Get, vbext_pk_Let, and so on are described in Access 2000 Help but cannot be used in 2000 code. I used the numeric values of those constants so that this section of code would run in both 97 and 2000.

Subroutines and functions return the same ProcKind value, so I have to examine the name of the procedure to determine whether it's a function or a procedure.

```
Select Case glbProcKind
  Case 0   'Procedure
      ProcType = glbModule.Lines(LineNo, 1)
      If InStr(ProcType, " Sub ") > 0 Then
         Me.optProcType = 1
         ProcType = "Sub"
      ElseIf InStr(ProcType, "Sub ") = 1 Then
         Me.optProcType = 1
         ProcType = "Sub"
      Else
         ProcType = "Function"
         Me.optProcType = 2
      End If
  Case 1     ' Property Let
      Me.optProcType = 3
      ProcType = "Property"
  Case 3     ' Property Get
      Me.optProcType = 4
      ProcType = "Property"

  Case 2     ' Property Set
      Me.optProcType = 5
      ProcType = "Property"
End Select

glbProcType = ProcType
```

The end result of all this code is that I can load the global variable glbProcType with the correct procedure description string and pass it into the error trap insertion routine.

Inserting the error trap code

The setup form inserts the code into your module with the InsertErrTrap function. This function is implemented from the click event of the form's cmdAddit button. Most of the work is done before this call is made. Key variables are passed into the function when it's called:

```
If InsertErrTrap(glbModName, glbProcType, _
   glbFindString, glbProcName, glbFoundLine, _
   glbModule) Then
```

The InsertErrTrap function begins like this:

```
Function InsertErrTrap(Modulename As Variant, _
    ProcType As Variant, _
    FindString As Variant, _
    ProcName As Variant, _
    FoundLine As Variant, _
    MDL As Module) As Boolean

     Dim OutStr As String
     Dim DateStr As String
     Dim LinesToMoveUp As Long
     Dim StartLine As Long
     Dim EndLine As Long
     Dim StartCol As Long
     Dim EndCol As Long
```

I then get the current date for insertion into the comments section of the code:

```
DateStr = Format$(Date, "MMMM DD, YYYY")
```

OutStr holds the main code string that's inserted into the procedure:

```
OutStr = Nz(DLookup("CodeString", _
   "tblErrorTrapMaster", _
   "CodeKey = 'CS1' and FindString = '" & _
   FindString & "'"), "")
```

OutStr has four special strings that are replaced with key data fields. These fields are passed in from the procedure header and are subsequently inserted into your error trap.

I place these special strings (XDS, XPN, XPT, and XMN) in the error trap text and swap in the correct module and procedure names and so forth using the VBA Replace function (the 97 version uses my Substitute

Continues on page 18

Error Trapping...

Continued from page 6

function). This technique, even though it's rather low-tech, is quite effective. If you don't like my XDS/XPN convention you can use any strings you want for this. Just take care to use a string combination you're sure not to find in normal text (otherwise, the replacement may occur when you don't want it to). If you choose to use your own replacement strings, remember to change the following code so the replace command will work:

```
OutStr = Replace(OutStr, "XDS", DateStr)
OutStr = Replace(OutStr, "XPN", ProcName)
OutStr = Replace(OutStr, "XPT", ProcType)
OutStr = Replace(OutStr, "XMN", Modulename)
```

The updated OutStr text is now inserted into the module at the original FoundLine position, using the powerful InsertLines method of the Module object:

```
MDL.InsertLines FoundLine, OutStr
```

Now I find the FindString that triggered the process and delete that line using the DeleteLines method:

```
If MDL.Find(FindString, StartLine, StartCol, _
                EndLine, EndCol) = True Then
     MDL.DeleteLines StartLine, 1
End If
```

Once the text is inserted, it's time to reposition the cursor down to the middle section of the error trap so that the user can insert their code. Doing this means moving up to the appropriate line, as the cursor was moved down with the insert of the error trap code. There's one assumption you'll need to consider to determine whether it's appropriate to you: The error trap text has a fixed header size of seven lines. If your header depth changes, you should adjust the 7 in the following code in order to get the cursor placed properly. The technique that I use to determine how far I have to move is to count the number of carriage returns (hence lines) in OutStr, and then subtract the number of lines in the header:

```
LinesToMoveUp = CharCount(Chr(13), OutStr) - 7
If LinesToMoveUp < 0 Then
   LinesToMoveUp = 0
End If
```

Finally, I use SendKeys to reposition the cursor. Using SendKeys is always risky, though this use is fairly benign. The cursor should end up just after the header in the error trap:

```
SendKeys "{HOME}" & "{UP " & _
         CStr(LinesToMoveUp) & "}"
InsertErrTrap = True
```

The error trap's lines are now inserted with the cursor positioned where it should be, awaiting your keystrokes to add all the rest of the code in your function.

Dedication: This article is dedicated to my good and very open-minded friend Norbert Foigelman, who has recently spent many hours retrofitting error traps into his code in response to my tirades on the subject. ▲

ERRTRP.ZIP at www.smartaccessnewsletter.com

Keith Bombard is a contract programmer employed by Systems Group, specializing in large Access implementations. He's been developing Office-based solutions for multiple clients since 1993. Prior to that, Keith was a systems manager in the financial industry. He's currently on an extended assignment building seven large Access databases for the State of Connecticut, Department of Environmental Protection. Keith.Bombard@PO.STATE.CT.US.

Notes on Builder Implementation

- *Access 2000 or 2002*—Import all the objects from the ETrap2000.mdb into your application (the application you want to use the builder with). In any new event procedure, procedure or function, enter the FindString zzz1 into the code after the procedure heading and before the End Sub (or End Function) line. Run the Code Macro A_ETrap (Tools | Macros menu). The setup form should appear. Click the Add Trap button to add the error trapping code into your module. If you enter in zzz3 instead, the code will be inserted without the setup form.
- *Access 97*—Import all the objects from the ETrap97.mdb into your application (the application you want to use the builder with). The 97 version uses an Autokeys macro, using F12 to call a similar A_ETrap function. Note: If your application already has the Autokeys macro, this ETrap97 Autokeys macro will import into your app as Autokeys1. Make sure you copy the code for the F12 key from the Autokeys1 macro into your Autokeys macro so that it will take effect. If you're already using the F12 key, try to find another key combination to use for the builder. It shouldn't matter which key combination you use. In any new event procedure, procedure or function, enter the FindString zzz1 into the code after the procedure heading and before the End Sub (or End Function) line. Click F12 (or whatever key you assigned). The setup form should appear. Click the Add Trap button to add the error trapping code into your module. If you enter in zzz3 instead, the code will be inserted without the setup form.