Fehlerbehandlung mit vbWatchdog

André Minhorst

Fehlerbehandlung ist für viele eine lästige Pflicht. Wenn man es richtig machen will, muss man jede einzelne Routine mit einer Fehlerbehandlung versehen. Dazu kommen noch Zeilennummern, Angabe des Modul- und Prozedurnamens und Informationen wie die Fehlernummer und Beschreibung. Die damit zusammenhängende Schreibarbeit kann man sich auf elegante Weise vom Hals schaffen – und mehr. Erreichen können Sie dies mit vbWatchdog – vier Klassenmodule und ein paar Zeilen Code sorgen für eine professionelle Fehlerbehandlung und wesentlich schlankere Prozeduren.

vbWatchdog gibt es in zwei Versionen: Standard und Enterprise. Die Enterprise-Version enthält einige Features mehr, die den aktuellen Unterschied von 50% Aufschlag zum Preis der Standardversion mehr als rechtfertigen. Normalerweise werden in **Access im Unternehmen** keine kostenpflichtigen Tools vorgestellt, aber wenn eines soviel Arbeit wie in diesem Fall sparen kann, machen wir eine Ausnahme.

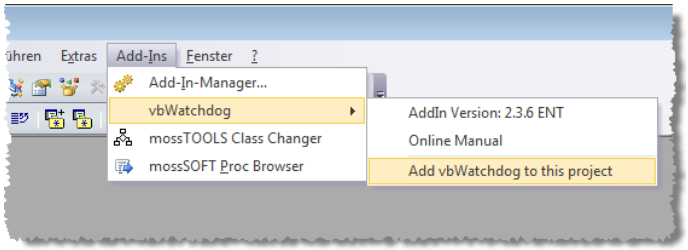


Abb. 1: Einfügen des vbWatchdog in ein VBA-Projekt

Bis zum 31.7.2010 erhalten Sie als Access im Unternehmen-Leser **vbWatchdog** in der Enterprise-Version zum Preis von *,** Euro. **vbWatchdog** können Sie in beliebig viele Anwendungen einbauen. Preise für Mehrbenutzer-Lizenzen finden Sie auf der Webseite unter **www.acciu.de/watchdog**.

Die Testversion von **vbWatchdog** finden Sie unter dem gleichen Link. Nach dem Download starten Sie die **.exe**-Datei, die ein COM-Add-In für den VBA-Editor installiert. Dieses fügen Sie im VBA-Editor über den Menüeintrag **Add-Ins|vbWatchdog|Add vbWatchdog to this project** zum aktuellen VBA-Projekt hinzu (s. Abb. 1).

Dies fügt dem aktuellen Projekt vier Klassenmodule hinzu, welche die komplette Funktionalität enthalten – es sind keine externen DLLs et cetera nötig (s. Abb. 2). Sie geben **vbWatchdog** also als Teil der Anwendung weiter.

Und damit kann die Fehlerbehandlung schon beginnen: Sie brauchen **vbWatchdog** nur zu aktivieren und schon kümmert sich das Tool um alle

Zusammenfassung
Verwenden Sie vbWatchdog, um Ihre herkömmliche Fehlerbehandlung durch eine globale und moderne Fehlerbehandlung abzulösen.
Techniken
VBA, Fehlerbehandlung
Voraussetzungen
Access 2000 und höher
Beispieldateien
vbWatchdog.mdb
Shortlink
www.access-im-unternehmen.de/****

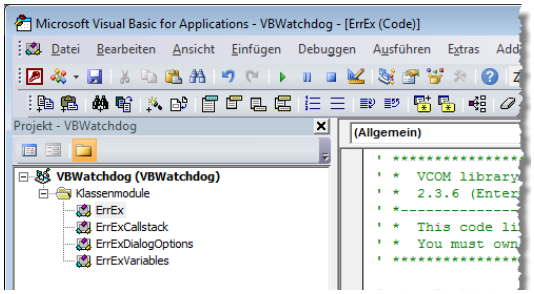


Abb. 2: Die vier Klassen des vbWatchdog

auftretenden Laufzeitfehler. Die Aktivierung erfolgt im einfachsten Fall mit der folgenden Anweisung (eine Instanzierung der Klasse ErrEx ist nicht nötig):

```
Public Sub vbWatchdogAktivieren()
    Call ErrEx.Enable("")
End Sub
```

Damit sorgen Sie dafür, dass **vbWatchdog** im Fehlerfall seinen eingebauten Dialog anzeigt. Nehmen wir an, Sie starten bei aktiviertem **vbWatchdog** die folgende Prozedur:

```
Public Sub Testfehler()
    Debug.Print 1 / 0
End Sub
```

Die resultierende Fehlermeldung sieht dann wie in Abb. 3 aus. Wenn Sie nicht mehr mit **vbWatchdog** arbeiten möchten, deaktivieren Sie ihn mit der folgenden Anweisung:

```
ErrEx.Disable
```

Wenn Sie nicht die eingebaute Fehlermeldung von **vbWatchdog** sehen wollen, können Sie selbst eine globale Fehlerbehandlungsroutine angeben. Dazu aktivieren Sie **vbWatchdog** wie oben, geben aber den Namen der Prozedur an, die im Falle eines Fehlers aufgerufen werden soll:

```
Public Sub vbWatchdogMitEigenerMeldung()
    Call ErrEx.Enable("aiuFehlerbehandlung")
End Sub
```

Die eigene Fehlerbehandlung soll in der Prozedur **aiuFehlerbehandlung** erfolgen und sich zunächst einfach per Meldungsfenster zu Wort melden:

```
Public Sub aiuFehlerbehandlung()
    MsgBox "Fehler!"
End Sub
```

Lösen Sie nun erneut einen Fehler aus, erscheint erst das Meldungsfenster und dann erneut der eingebaute Dialog von **vbWatchdog**.

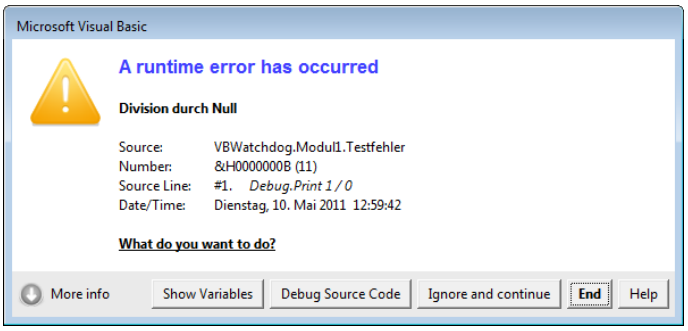
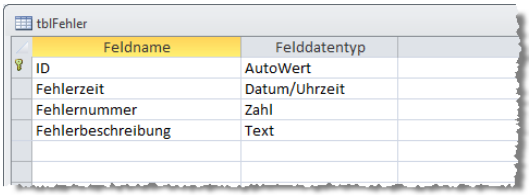


Abb. 3: Eine einfache Fehlermeldung des vbWatchdog

Fehler protokollieren

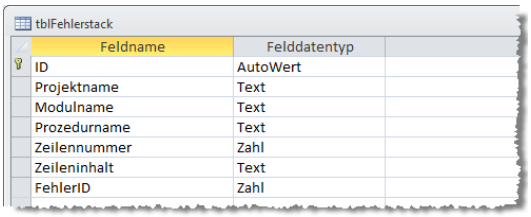
Die Fehlerbehandlungsprozedur **aiuFehlerbehandlung** können Sie beispielsweise nutzen, um die Fehler in Tabellen zu dokumentieren. Dazu legen Sie zunächst eine einfache Fehlertabelle wie in Abb. 4 an. Normalerweise würden Sie hier auch Informationen wie Projektname, Prozedurname, Modulname, Zeilennummer oder den Inhalt der fehlerhaften Zeile eintragen.

Nun bietet **vbWatchdog** allerdings auch noch die Möglichkeit, Informationen aller Routinen auszugeben, die zum Auslösen des Fehlers geführt haben. Wenn also die Prozedur **Testfehler2** die Prozedur **Testfehler1** aufruft und dieser wiederum die Prozedur **Testfehler**, in der schließlich ein Fehler auftritt, dann kann es für den Entwickler von großem Nutzen sein, wenn er diesen Hergang kennt. Die Informationen dieses sogenann-


tblFehler

Feldname	Felddatentyp
ID	AutoWert
Fehlerzeit	Datum/Uhrzeit
Fehlernummer	Zahl
Fehlerbeschreibung	Text

Abb. 4: Tabellen zum Speichern von Fehlern ...


tblFehlerstack

Feldname	Felddatentyp
ID	AutoWert
Projektname	Text
Modulname	Text
Prozedurname	Text
Zeilennummer	Zahl
Zeileninhalt	Text
FehlerID	Zahl

Abb. 5: ... und den Schritten, die dazu führten.

Listing 1: Speichern von Fehlerinformationen in drei Tabellen

```
Public Sub aiuFehlerbehandlung()
    Dim db As DAO.Database
    Dim lngFehlerID As Long
    Dim lngStackID As Long
    Dim strSQLError As String
    Dim strSQLErrorstack As String
    Dim strSQLVariablen As String
    Set db = CurrentDb
    With ErrEx
        strSQLError = "INSERT INTO tblFehler(Fehlerzeit, Fehlernummer, Fehlerbeschreibung) VALUES(" _
            & ISODatum(Now) & ", " & .Number & ", '" & .Description & "')"
        db.Execute strSQLError, dbFailOnError
        lngFehlerID = db.OpenRecordset("SELECT @@IDENTITY").Fields(0)
        With ErrEx.Callstack
            Do
                strSQLErrorstack = "INSERT INTO tblFehlerstack(Projektname, Modulname, Prozedurname, " _
                    & "Zeilennummer, Zeileninhalt, FehlerID) VALUES('" & .ProjectName & "','" _
                    & .ModuleName & "','" & .ProcedureName & "'," & .LineNumber & ",'" & .LineCode _
                    & "'," & lngFehlerID & ")"
                db.Execute strSQLErrorstack, dbFailOnError
                lngStackID = db.OpenRecordset("SELECT @@IDENTITY").Fields(0)
                With ErrEx.Callstack.VariablesInspector
                    .FirstVar
                    Do While Not .IsEnd
                        strSQLVariablen = "INSERT INTO tblVariablen(Variablenname, Variablenwert, " _
                            & " Datentyp, Gueltigkeitsbereich, StackID) VALUES('" & .Name & "','" _
                            & .Value & "','" & .Type & "','" & .Scope & "'," & lngStackID & ")"
                        db.Execute strSQLVariablen, dbFailOnError
                        .NextVar
                    Loop
                End With
            Loop While .NextLevel
        End With
    End With
End Sub
```

ten Fehlerstacks liefert **vbWatchdog** ebenfalls, weshalb wir diese Informationen in eine weitere, per 1:n-Beziehung mit der Tabelle **tblFehler** verknüpfte Tabelle namens **tblFehlerstack** auslagern (s. Abb. 5).

Variablen zum Fehlerzeitpunkt archivieren

Wenn wir schon beim Archivieren sind, legen wir auch gleich noch eine dritte Tabelle zum Speichern der Variablen und ihrer Inhalte zum Zeitpunkt des Fehlers an. Diese sieht wie in Abb. 7 aus und ist über das Feld **FehlerstackID** mit der Tabelle **tblFehlerstack** verknüpft.

Archivieren von Fehlern, Stack und Variablen

Nun passen wir die Fehlerbehandlungs-Prozedur so an, dass sie die gewünschten Daten in die beiden Tabellen schreibt. Das Ergebnis sieht wie in Listing 1 aus. Die Prozedur verwendet zunächst die beiden Eigenschaften **Number** und **Description** des **ErrEx**-Objekts und schreibt diese in die Tabelle **tblFehler**. Dann durchläuft es eine **Do... Loop**-Schleife und schreibt in jedem Durchlauf

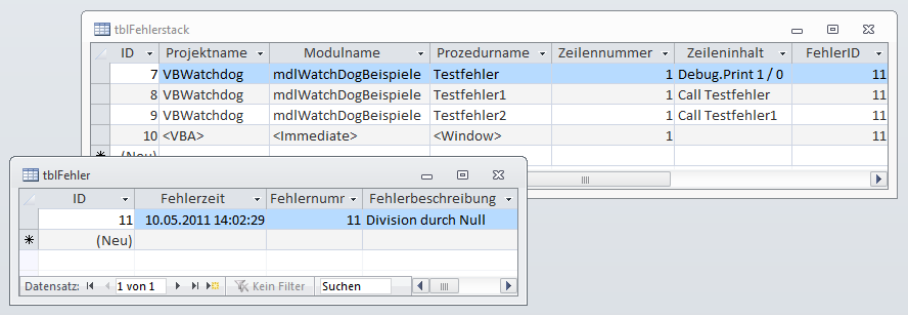


Abb. 6: Dokumentation eines einfachen Fehlers

die Eigenschaften **ProjectName**, **ModuleName**, **ProcedureName**, **LineNumber** und **LineCode** des **CallStack**-Elements in die zweite Tabelle.Außerdem trägt es dort die ID des soeben hinzugefügten Eintrags der Tabelle **tblFehler** hinzu, damit die Stack-Informationen dem Fehler zugeordnet werden können. Abb. 6 zeigt, wie das für einen einfachen Fehler aussehen kann, der über drei Prozeduren angestoßen wurde.

Die Prozedur erledigt noch einen Schritt mehr (nur für die Enterprise-Version): Der **vbWatchdog** liefert auch noch die Variablenwerte aller Variablen der Prozedur, die den Fehler ausgelöst hat sowie aller Prozeduren, die sich im Stack befinden.

Das Objekt für den Zugriff auf diese Informationen heißt **ErrEx.Callstack.VariablesInspector**. Die erste Methode **FirstVar** lädt die Informationen der ersten Variable, sodass diese über die Eigenschaften **Name**, **Value**, **TypeDesc** (Datentyp) und **Scop** (Gültigkeitsbereich) ausgelsen werden können. In einer **Do While**-Schleife wird so oft die **NextVar**-Methode ausgeführt, bis die Eigenschaft **IsEnd** den Wert **True** enthält – dann sind alle Variablen in die Tabelle **tblVariablen** geschrieben.

Die Fehlerinformationen können Sie natürlich auch in eine Text- oder XML-Datei schreiben oder im Direktfenster ausgeben.

Fehlermeldung unterdrücken

Die Fehlermeldung des **vbWatchdog** unterdrücken Sie genauso, wie Sie es mit herkömmlichen Fehlermeldungen machen: Sie stellen der Zeile, die einen Fehler auslösen könnte, die folgende Anweisung voran:

```
On Error Resume Next
```

Dies führt aber dennoch dazu, dass eine eventuell angegebene Errorhandler-Prozedur aufgerufen wird. Sie können also Fehler übergehen und dennoch die Fehlerinformationen aufzeichnen.

Das gleiche gilt, wenn Sie wie in folgendem Beispiel eine benutzerdefinierte Fehlermeldung ansteuern:

```
Public Sub TestfehlerMitOnErrorGoto()
    On Error GoTo Fehler
    Debug.Print 1 / 0
Fehler:
    MsgBox Err.Number & " " & Err.Description
End Sub
```

Fehlerbehandlungen, die etwa das Einfügen doppelter Werte in eindeutig indizierten Tabellenfeldern abfangen sollen, behalten Sie so bei wie zuvor.

Zeilennummern anzeigen

Bei herkömmlicher Fehlerbehandlung konnten Sie mit der undokumentierten **Erl**-Funktion die Nummer einer fehlerhaften Zeile abfragen, wenn

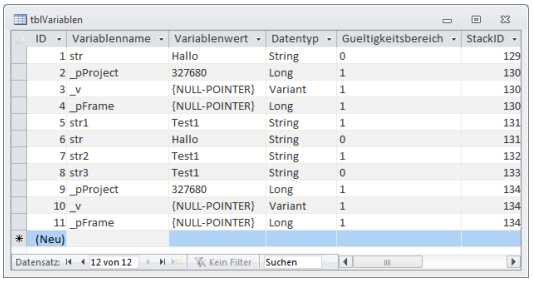


Abb. 7: Speichern der Variablen zum Zeitpunkt eines Fehlers

die Zeilen der betroffenen Prozedur entsprechende Zeilennummern enthielten.

vbWatchdog liefert beim Auftreten eines Fehlers automatisch eine Zeilennummer, obwohl der VBA-Code gar keine Zeilennummern enthält.

Wie kann das sein und muss ich jetzt von Hand durchzählen, in welcher Zeile der Fehler aufgetreten ist?

Nein: Sie brauchen einfach nur mit dem Menüeintrag **Ansicht|Symbolleisten|vbWatchdog.LineNumberViewer** eine neue Symbolleiste hervorzuzaubern und den einzigen dort enthaltenen Befehl anzuklicken.

Schon zeigt der VBA-Editor die Zeilennummern an (s. Abb. 8).

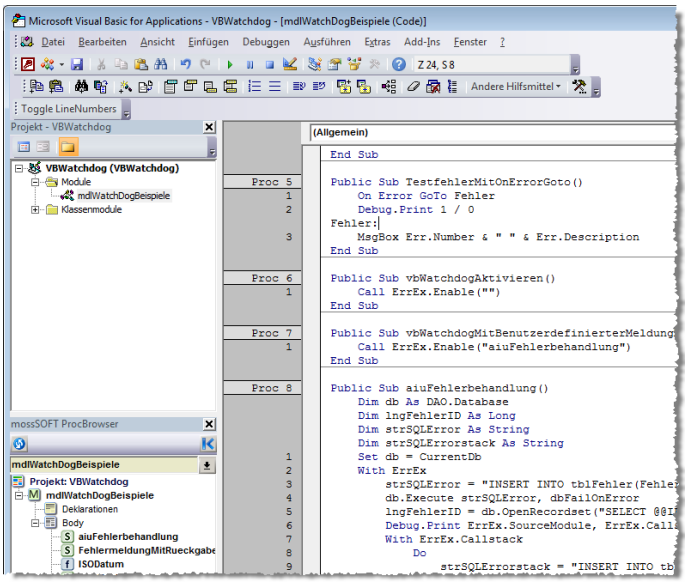


Abb. 8: Zeilennummern einblenden

Den Fehlerdialog von **vbWatchdog** können Sie nach Ihren Wünschen anpassen. Am einfachsten gelingt das, wenn Sie die ebenfalls auf der Webseite des Herstellers verfügbare Beispieldatenbank namens **vbWatchdog_Sample.mdb** starten und dort im Hauptformular auf **Customize** klicken.

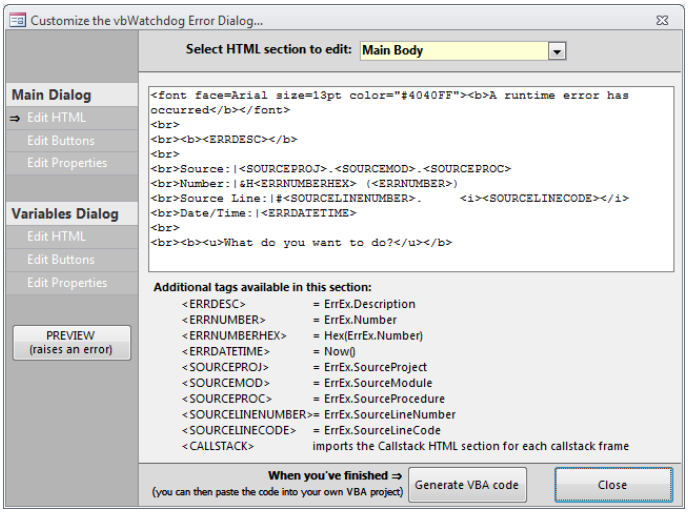


Abb. 9: Zusammenstellen eines benutzerdefinierten Fehlerdialogs

Eigenen Fehlerdialog programmieren

Es erscheint ein Dialog, mit dem Sie alle verfügbaren Einstellungen vornehmen können (s. Abb. 9).

Alle Einstellungen werden auf Knopfdruck zu einer Reihe von VBA-Anweisungen zusammengestellt und in die Zwischenablage kopiert.

Sie fügen diese Anweisungen am besten in die gleiche Prozedur ein, die auch die Aktivierung des **vbWatchDog** vornimmt.

Die notwendigen Einstellungen nehmen Sie per VBA über zwei Objekte vor:

- **ErrEx.DialogOptions**: Ändert das Aussehen und Verhalten des Standarddialogs.
- **ErrEx.VariablesDialogOptions**: Ändert das Aussehen des Dialogs zur Anzeige der Variablen zum Zeitpunkt des Auftretens eines Fehlers (s. Abb. 10).

Die einzelnen Einstellungen können Sie natürlich auch manuell zusammenstellen. Wenn Sie etwa den HTML-Ausdruck für das Aussehen des oberen Bereichs ändern möchten, lassen Sie sich diesen zunächst als Vorlage im Direktfenster ausgeben:

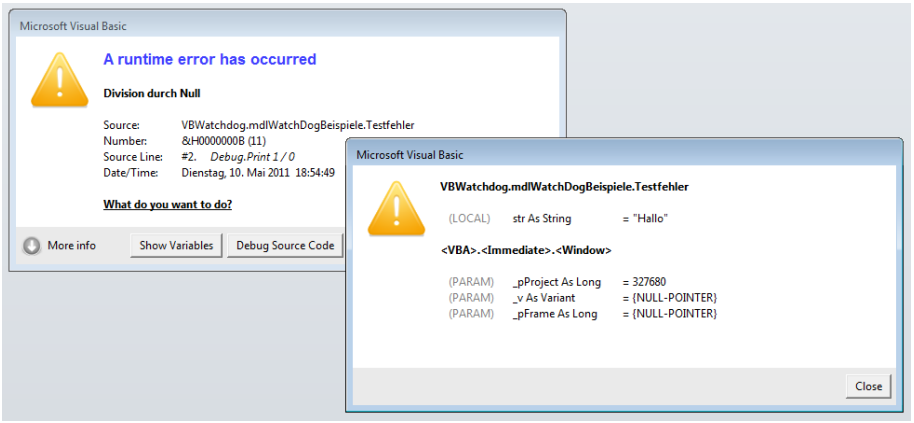


Abb. 10: Ein weiterer Dialog kann die Variablen und ihre Inhalte zum Zeitpunkt des Fehlers anzeigen.

```
? ErrEx.DialogOptions.HTML_MainBody
<font face=Arial size=13pt
color="#4040FF"><b>A runtime error has oc-
curred</b></font><br><br><b><ERRDESC></b>
<br><br>Source:|<SOURCEPROJ>.<SOURCEMOD>
.<SOURCEPROC><br>Number:|&H<ERRNUMBERHEX>
(<ERRNUMBER>)<br>Source Line:|#<SOURCELINENUMBER>.
<i><SOURCELINECODE></i><br>Date/Time:|<ERRDATETIM
E><br><br><b><u>What do you want to do?</u></b>
```

Danach ändern Sie den Bereich nach eigenem Wunsch und schreiben den HTML-Ausdruck durch Zuweisung an die Eigenschaft **DialogOptions.HTML_MainBody** wieder zurück. Bei der nächsten Fehlermeldung finden Sie Ihre eigene Version vor.

Noch einfacher geht es natürlich, wenn Sie die gewünschten Einstellungen in eine Prozedur schreiben, die auch noch weitere Einstellungen vornimmt.

Für eine sehr rudimentäre, aber deutschsprachige Fehlermeldung führen Sie die Prozedur aus Listing 2 aus und provozieren dann einen Fehler.

Die entsprechende Meldung sieht dann wie in Abb. 11 und Abb. 12 aus.

Beim Zusammenstellen der HTML-Eigenschaften wie **HTML_MainBody**, **HTML_MoreInfoBody**, **HTML_CallStackItem** und **HTML_VariableItem** kommt es neben der richtigen Auszeichnung mit HTML-Tags und den Text auf zwei Dinge an:

- Fehlerinformationen können mit entsprechenden Platzhaltern wie etwa **<ERRDESC>** eingefügt werden (siehe unten).
- Das Pipe-Zeichen (|) entspricht einem Tabulator.

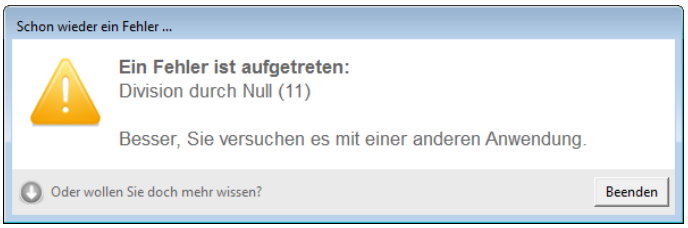


Abb. 11: Eine benutzerdefinierte Fehlermeldung ...

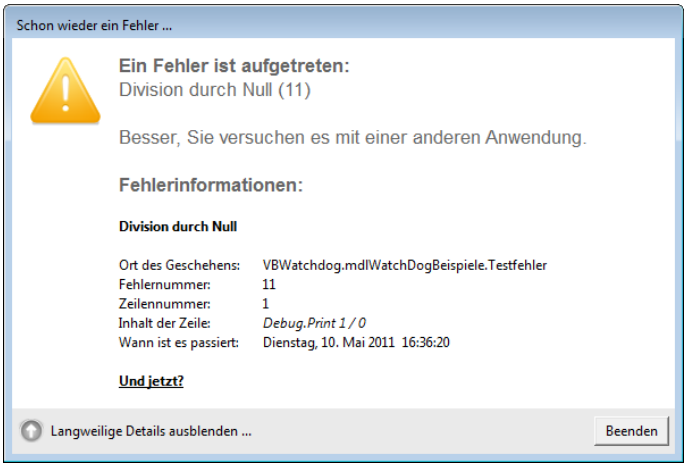


Abb. 12: ... mit aufklappbaren Details.

Listing 2: Definieren des Aussehens einer Fehlermeldung

```
Public Sub vbWatchdogMitBenutzerdefinierterMeldungAktivieren()
    Call ErrEx.Enable("aiuFehlerbehandlung")
    With ErrEx.DialogOptions
        .HTML_MainBody = "<font face=Arial size=13pt color=""#666666""><b>Ein Fehler ist " _
            & "aufgetreten:</b><br><ERRDESC> (<ERRNUMBER>)<br><br>Besser, Sie versuchen es mit " _
            & "einer anderen Anwendung."
        .HTML_MoreInfoBody = "<font face=Arial size=13pt color=""#666666""><br>" _
            & "<b>Fehlerinformationen:</b></font><br><br><b><ERRDESC></b><br>" _
            & "<br>Ort des Geschehens:|<SOURCEPROJ>.<SOURCEMOD>.<SOURCEPROC><br>Fehlernummer:|" _
            & "<ERRNUMBER><br>Zeilennummer:|<SOURCELINENUMBER><br>Inhalt der Zeile:|<i>" _
            & "<SOURCELINECODE></i><br>Wann ist es passiert:|<ERRDATETIME><br><br>" _
            & "<b><u>Und jetzt?</u></b>"
        .RemoveAllButtons
        .AddButton "Beenden", BUTTONACTION_ONERROREND
        .MoreInfoCaption = "Oder wollen Sie doch mehr wissen?"
        .LessInfoCaption = "Langweilige Details ausblenden ..."
        .WindowCaption = "Schon wieder ein Fehler ..."
    End With
End Sub
```

Es gibt die folgenden Platzhalter für Fehlerinformationen:

- **<ERRDESC>**: Fehlerbeschreibung
- **<ERRNUMBER>**: Fehlernummer
- **<ERRNUMBERHEX>**: Fehlernummer (hexadezimal)
- **<ERRDATETIME>**: Datum und Zeit
- **<SOURCEPROJ>**: Projektname
- **<SOURCEMOD>**: Modulname
- **<SOURCEPROC>**: Prozedurname
- **<SOURCELINENUMBER>**: Zeilennummer
- **<SOURCELINECODE>**: Inhalt der Zeile
- **<CALLSTACK>**: Liste der Prozeduraufrufe, siehe unten

In unserem Beispiel haben wir neben der Änderung der HTML-Bereiche noch andere Elementen manipuliert.

Die Methode **RemoveAllButtons** entfernt alle bestehenden Schaltflächen. Die Methode **AddButton** fügt Schaltflächen hinzu, wobei Sie einen benutzerdefinierten Text angeben und die Aktion festlegen können.

Dafür gibt es die folgenden Konstanten:

- **<BUTTONACTION_SHOWVARIABLES>**: Variablen anzeigen
- **<BUTTONACTION_ONERRORDEBUG>**: Code debuggen
- **<BUTTONACTION_ONERRORRESUMENEXT>**: Code fortsetzen
- **<BUTTONACTION_SHOWHELP>**: Access-Hilfe zu diesem Fehler anzeigen
- **<BUTTONACTION_ONERROREND>**: Prozedur beenden

Außerdem haben wir die folgenden Eigenschaften eingestellt:

- **MoreInfoCaption**: Beschriftung der Schaltfläche zum Ausklappen der Details

- **LessInfoCaption**: Beschriftung der Schaltfläche zum Einklappen der Details
- **WindowCaption**: Fenstertitel der Fehlermeldung

Callstack in der Fehlermeldung

Die Callstack-Informationen können Sie in einem der beiden Bereiche der Fehlermeldung unterbringen. Dazu fügen Sie das Tag **<CALLSTACK>** hinzu.

Wie der Inhalt dieses sich für alle Aufrufe wiederholenden Bereichs aussieht, legen Sie mit der Eigenschaft **HTML_CallStackItem** fest. Dieser kann folgende Variablen enthalten:

- **<SOURCEPROJ>**: Projektname
- **<SOURCEMOD>**: Modulname
- **<SOURCEPROC>**: Prozedurname
- **<SOURCELINENUMBER>**: Zeilennummer
- **<SOURCELINECODE>**: Inhalt der Zeile
- **<VARIABLES>**: Inhalt der Variablen

Variablen in der Fehlermeldung

Die Variablen zum Zeitpunkt des Fehlers können Sie ebenfalls in der Fehlermeldung ausgeben. Dazu legen Sie die Eigenschaft **HTML_VariablesItem** des **ErrEx**-Objekts fest.

Die Variableninhalte werden für jedes Element des Stack ausgegeben, wenn dort das **<VARIABLES>**-Tag enthalten ist. Für die Variablen gibt es folgende Platzhalter:

- **<VARSCOPE>**: Gültigkeitsbereich
- **<VARNAME>**: Name der Variablen
- **<VARTYPE>**: Datentyp
- **<VARVALUE>**: Variablenwert

Benutzerdefinierte Platzhalter

Auch wenn Sie das Aussehen des Fehlerdialogs bereits festgelegt haben, können Sie zur Laufzeit den Inhalt benutzerdefinierter Platzhalter ändern.

Einen solchen Platzhalter legen Sie genauso wie einen der eingebauten Platzhalter an – mit dem kleinen Unterschied, dass er ein führendes Ausrufezeichen enthalten muss:

```
<!Benutzerdefinierter>
```

Wenn Sie diesen Platzhalter belegen möchten, erledigen Sie dies mit der folgenden Anweisung:

```
ErrEx.CustomVars("Benutzerdefiniert") = "Bla"
```

Try...Catch (nur Enterprise-Version)

Wenn Sie Fehler abfangen möchten, brauchen Sie nicht mehr die **On Error Goto <Marke>**-Syntax wie in diesem Beispiel:

```
Public Sub TryCatchAlt()
    On Error GoTo Fehler
    Debug.Print 1 / 0
    Exit Sub
Fehler:
    MsgBox "Dieser Fehler wurde behandelt."
End Sub
```

Sie definieren einfach am Ende der Prozedur einen Block, der wie folgt aussieht:

```
Public Sub TryCatch()
    Debug.Print 1 / 0
ErrEx.Catch 11
    MsgBox "Dieser Fehler wurde per Try...Catch
abgefangen."
End Sub
```

Die **Catch**-Anweisung sorgt dafür, das bei jedem Auftreten eines Fehlers bis hierhin der danach folgende Block ausgeführt wird. Tritt kein Fehler auf, endet die Prozedur mit der ersten **Catch**-Anweisung. Wenn Sie verschiedene Fehler mit der gleichen **Catch**-Methode abfangen möchten, verwenden Sie die folgende Syntax:

```
ErrEx.Catch 11, 3022
```

Für verschiedene Fehler haben Sie bisher vermutlich eine **Select Case**-Anweisung verwendet:

```
Public Sub TryCatchMitSelectCaseAlt()
    On Error GoTo Fehler
    Debug.Print 1 / 0
    Exit Sub
Fehler:
    Select Case Err.Number
        Case 11
            MsgBox "Dieser Fehler wurde behandelt."
        Case 3022
            '...
    End Select
End Sub
```

Mit **vbWatchdog** verwenden Sie einfach mehrere Catch-Anweisungen:

```
Public Sub TryCatch()
    Debug.Print 1 / 0
ErrEx.Catch 11
    MsgBox "Dieser Fehler wurde per Try...Catch
abgefangen."
ErrEx.Catch 3022
    '...
End Sub
```

Das Abfangen aller Fehler erledigen Sie mit der **CatchAll**-Methode:

```
ErrEx.CatchAll
```

Und wenn Sie nach dem Auftreten eines Fehlers noch dringend einige Aufgaben erledigen müssen, sah das ohne **vbWatchdog** so aus:

```
Public Sub TryCatchAltMitAufraeumen()
    On Error GoTo Fehler
    Debug.Print 1 / 0
    Exit Sub
Ende:
    MsgBox "Geschafft ..."
Fehler:
    MsgBox "Dieser Fehler wurde behandelt."
    GoTo Ende
End Sub
```

Mit **vbWatchdog** schreiben Sie die auch nach einem Fehler noch durchzuführenden Anweisungen einfach im Anschluss an den Aufruf der **ErrEx.Finally**-Anweisung:

```
Public Sub TryCatchMitAufraeumen()
    Debug.Print 1 / 0
ErrEx.Catch 11
    MsgBox "Dieser Fehler wurde behandelt."
ErrEx.Finally
    MsgBox "Geschafft!"
End Sub
```

Die dort befindlichen Anweisungen werden mit oder ohne Fehler auf jeden Fall ausgeführt.

Zusammenfassung und Ausblick

vbWatchdog ist eines der besten Tools, die es für VBA auf dem Markt gibt. Es spart eine große Menge Programmierarbeit und somit viel Zeit. Noch dazu wird der Code durch die globale Fehlerbehandlung wesentlich besser lesbar: Sie benötigen keine klassische Fehlerbehandlung mehr und es sind auch keine Zeilennummern mehr nötig.

Das Tool arbeitet in **.mde**- und **.accde**-Dateien genauso wie unter der Runtime-Version von Access.