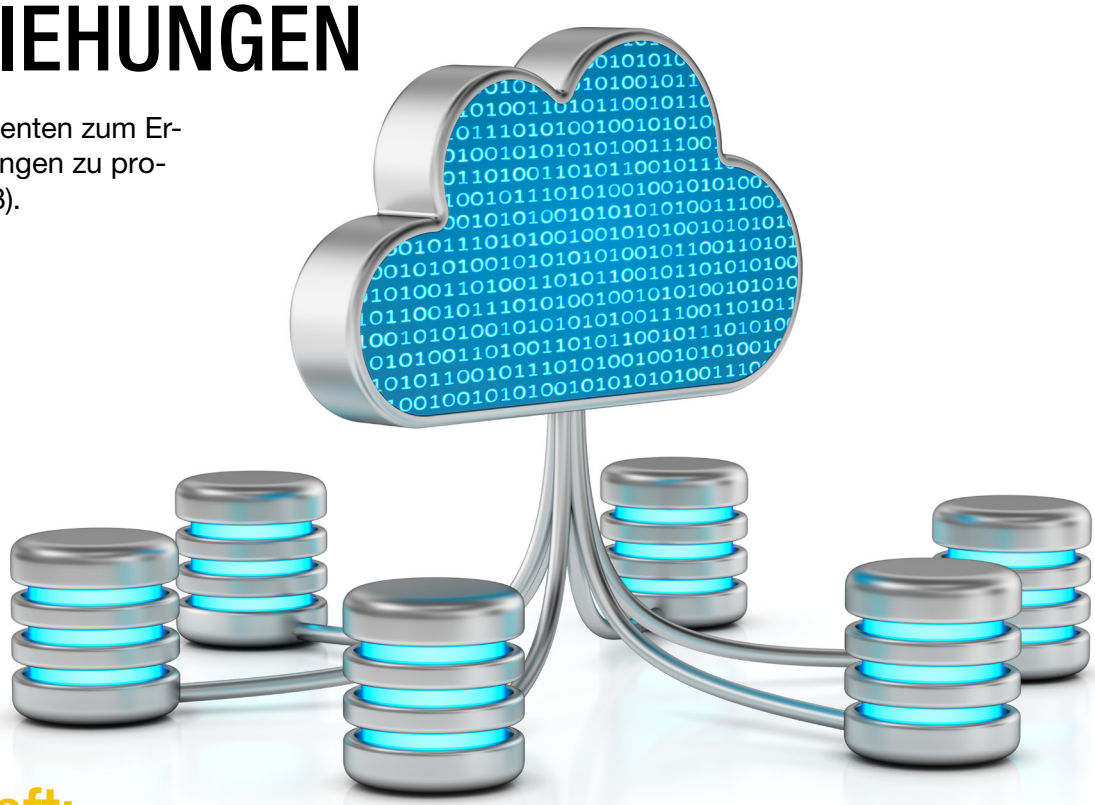
HAUFE.
Ausgabe 06/2021

ACCESS

IM UNTERNEHMEN

ASSISTENT FÜR M:N-BEZIEHUNGEN

Lernen Sie, einen Assistenten zum Erstellen von m:n-Beziehungen zu programmieren (ab Seite 53).



In diesem Heft:

ACCESS-ANWENDUNGEN WEITERGEBEN

Stellen Sie sicher, dass die installierte Anwendung als vertrauenswürdig eingestuft wird.

SEITE 30

FEHLERRESISTENTES RIBBON

Verhindern Sie, dass unbehandelte Laufzeitfehler die Ribbonfunktion beeinträchtigen.

SEITE 5

FORMULARE PER VBA ERSTELLEN

Programmieren Sie VBA-Code, um Formulare per Mausklick zu erstellen!

SEITE 16

m:n-Beziehungen leicht gemacht

Wer in die Datenbankentwicklung einsteigt und damit auch in den Entwurf von Datenmodellen, für den kann nach der 1:n-Beziehung die m:n-Beziehung die nächste Herausforderung sein. 1:n-Beziehung baut man unter Access ganz einfach mit dem Nachschlage-Assistenten. Für die Erstellung von m:n-Beziehungen und die dafür benötigte Verknüpfungstabelle sucht man vergeblich nach einem Tool. Kein Problem: Wir haben eines programmiert und zeigen die notwendigen Aktionen in einer ausführlichen Schritt-für-Schritt-Anleitung!

Eigentlich ist eine m:n-Beziehung ja nur eine Kombination zweier 1:n-Beziehungen. Allerdings benötigen wir ausgehend von den beiden zu verknüpfenden Tabellen nicht nur zwei Beziehungen, sondern auch noch eine Verknüpfungstabelle, welche die beiden Beziehungen zusammenführt. Unser Assistent nimmt Ihnen nicht nur die Aufgabe ab, die Beziehungen zu erstellen, sondern definiert auch gleich noch die Verknüpfungstabelle. Sie brauchen nur noch die beiden Tabellen auszuwählen, die verknüpft werden sollen, sowie die Primärschlüsselfelder, die an der Verknüpfung beteiligt sind – den Rest erledigt der Assistent für uns.

Wie Sie diesen programmieren, lernen Sie im Beitrag **Assistent für m:n-Beziehungen** ab Seite 53.

Viele Entwickler wollen, dass der Benutzer die Anwendung genau so nutzt, wie sie es vorgesehen haben – und dass der Benutzer keinen Zugriff auf Elemente wie beispielsweise den Navigationsbereich hat. Solche Elemente kann man ausschalten, sodass sie normalerweise nicht angezeigt werden. Betätigt der Benutzer jedoch beim Öffnen der Datenbank die Umschalttaste, umgeht er diese Einstellungen. Um das zu verhindern, können Sie die Funktion der Umschalttaste beim Öffnen blockieren. Wie das gelingt, lesen Sie unter **Umschalttaste sperren mit AllowByPassKey** ab Seite 2.

Wenn Sie Ihre Anwendung mit einer benutzerdefinierten Ribbondefinition ausgestattet haben, kann es zu Funktionsstörungen kommen, nachdem ein unbehandelter Laufzeitfehler aufgetreten ist. Welche Schritte nötig sind, damit das Ribbon dann immer noch fehlerfrei funktioniert, erfahren Sie im Beitrag **Ribbonvariable fehlerresistent machen** (ab Seite 5).

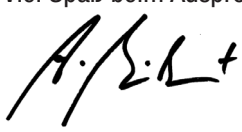
Ein zweiter Beitrag zum Thema Ribbon erläutert, wie Sie beim Auswählen eines der Ribbonreiter durch den Benutzer immer direkt ein dazu passendes Formular anzeigen können (**Bei Tab-Wechsel im Ribbon ein Formular anzeigen**, ab Seite 11).

Das Thema Setup für Access-Anwendungen behandeln wir weiter im Beitrag **Setup für Access: Vertrauenswürdige Speicherorte** (ab Seite 30). Hier schauen wir uns an, wie Sie direkt bei der Installation dafür sorgen können, dass die Access-Anwendung als vertrauenswürdig eingeordnet wird.

Ein wichtiges Objekt bei der Programmierung von Access-Anwendungen ist das Application-Objekt. Der Beitrag **Das Application-Objekt** zeigt ab Seite 36 alle Eigenschaften und Methoden dieses Objekts.

Schließlich liefert der Beitrag **Formulare per VBA erstellen** Informationen darüber, wie Sie programmgesteuert Formulare erstellen können – was beispielsweise für die Programmierung von Assistenten zum Erstellen von Formularen wichtig sein kann (ab Seite 16).

Viel Spaß beim Ausprobieren!



Ihr André Minhorst

Umschalttaste sperren mit AllowByPassKey

Unter Access können Sie einige Einstellungen vornehmen, damit der Benutzer kaum noch etwas davon sieht, dass es sich bei der Anwendung um eine Access-Anwendung handelt. Sie können beispielsweise den Navigationsbereich ausblenden oder die eingebauten Ribbon- und Backstage-Einträge mit einer benutzerdefinierten Ribbon-Definition ausblenden. Pfiffige Anwender finden allerdings schnell heraus, wie sich diese Änderungen beim Öffnen einer Datenbank blockieren lassen: durch einfaches Drücken und Halten der Umschalttaste. Also zeigen wir in diesem Beitrag noch einen Weg, wie Sie auch das noch ein wenig erschweren können. Eine wichtige Rolle spielt dabei eine Eigenschaft namens AllowByPassKey.

Das Öffnen einer Datenbankanwendung bei gedrückter Umschalttaste ist ein gängiges Mittel, um sämtliche vom Entwickler programmierten Aktionen, die beim Starten der Anwendung ausgeführt werden sollen, zu unterbinden. Unter anderem blockieren Sie die folgenden Aktionen:

- das Öffnen des als Startformular angegebenen Formulars
- das Ausführen des **AutoExec**-Makros
- die Anwendung einer benutzerdefinierten Ribbondefinition aus der Tabelle **USysRibbons**, die über die Option **Name des Menübands** ausgewählt wurde
- das Ausblenden des Navigationsbereichs über die Access-Optionen für die aktuelle Datenbank
- Verwenden der Spezialtasten von Access wie beispielsweise **F11** zum Anzeigen des Navigationsbereichs

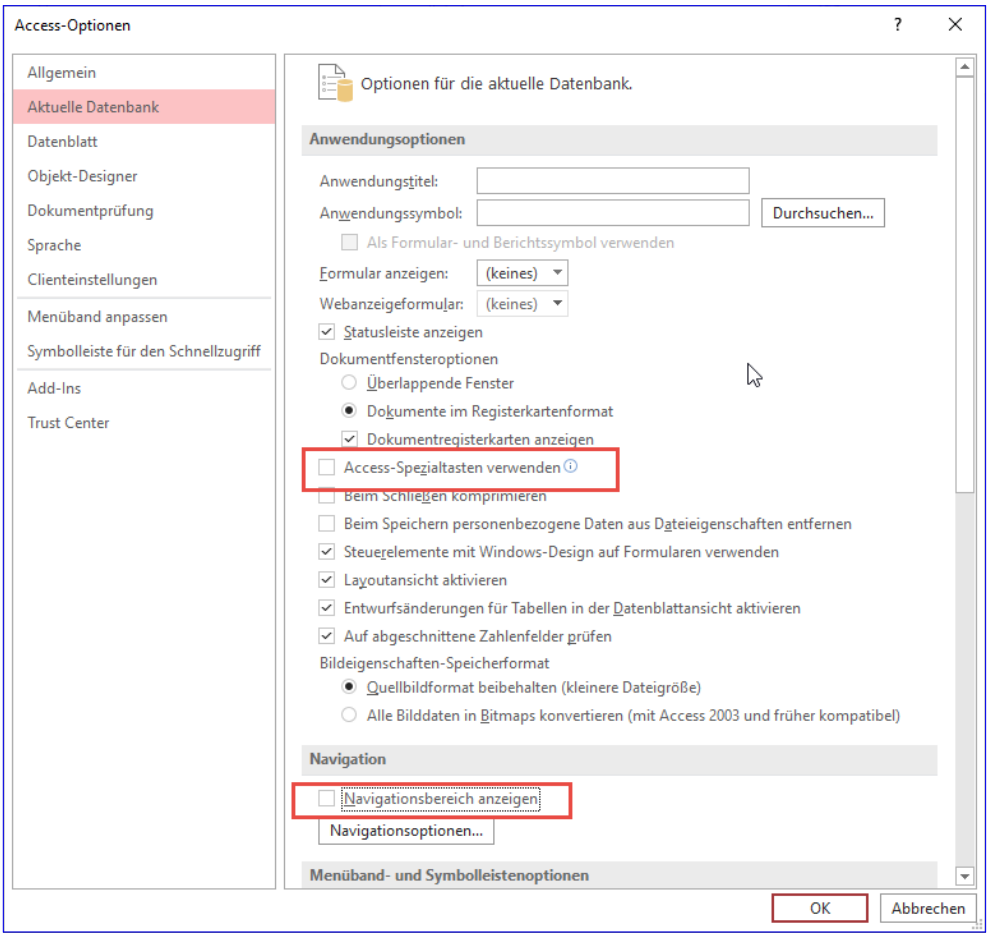


Bild 1: Zwei Beispiele für Optionen, die man per Umschalttaste aushebeln kann

Hält der Benutzer die Umschalttaste beim Öffnen gedrückt, wird die Datenbank so geöffnet, als ob all diese Änderungen nicht vorgenommen worden wären. Also wollen wir die Wirkung des Drückens der Umschalttaste unterbinden.

```
Public Sub UmschalttasteErlauben(Optional bolSperren As Boolean = True)
    Dim db As DAO.Database
    Dim prp As DAO.Property
    Set db = CurrentDb
    On Error Resume Next
    Set prp = db.Properties("AllowBypassKey")
    If Err.Number = 3270 Then
        On Error GoTo 0
        Set prp = db.CreateProperty("AllowBypassKey", dbBoolean, bolSperren)
        db.Properties.Append prp
    Else
        prp.Value = bolSperren
    End If
End Sub
```

Listing 1: Aktivieren oder Deaktivieren der Umschalttaste

Wir haben in den Optionen der Beispieldatenbank die beiden Einstellungen aus Bild 1 vorgenommen. Diese sorgen dafür, dass erstens der Navigationsbereich nicht angezeigt wird und zweitens die Spezialtasten von Access nicht genutzt werden können, um beispielsweise mit **F11** den Navigationsbereich doch noch zu aktivieren.

Die Eigenschaft AllowByPassKey

Es gibt eine Eigenschaft namens **AllowByPassKey**, welche die Funktion der Umschalttaste beim Öffnen einer Access-Anwendung erlaubt – oder auch nicht. Diese Eigenschaft wird als Element der **Properties**-Auflistung der aktuellen Datenbank angelegt. Dies können wir nur per VBA erledigen, weshalb wir die Prozedur aus Listing 1 angelegt haben.

Diese heißt **UmschalttasteErlauben** und erwartet einen **Boolean**-Wert als Parameter:

- **True**: Erlaubt die Funktion der Umschalttaste beim Öffnen der Datenbankanwendung.
- **False**: Sperrt die Funktion der Umschalttaste beim Öffnen der Datenbankanwendung.

Die Prozedur versucht bei deaktivierter Fehlerbehandlung, auf ein Element namens **AllowByPassKey** der Auflistung **Properties** für das mit **CurrentDb** ermittelte **Database**-Objekt der aktuellen Datenbank zuzugreifen. Löst dies einen Fehler aus, ist das Element noch nicht vorhanden und wir legen es mit der **CreateProperty**-Methode an. Diese erhält den Namen der zu erstellenden Property, den Datentyp (**dbBoolean**) und den Wert aus dem Prozedurparameter **bolSperren** als Parameter. Anschließend wird die Property noch mit der **Append**-Methode an die Liste der vorhandenen Properties angehängt.

Ist die Property bereits vorhanden, löst der Zugriff darauf keinen Fehler aus und wir können ihr einfach den gewünschten Wert zuweisen.

Der Aufruf zum Verhindern der Funktion der Umschalttaste, abgesetzt beispielsweise im Direktbereich, lautet:

```
UmschalttasteErlauben False
```

In beiden Fällen können wir den Wert der Eigenschaft anschließend mit der folgenden Anweisung über den Direktbereich des VBA-Editors abfragen:

```
? CurrentDb.Properties("AllowByPassKey")
```

Wenn diese Eigenschaft den Wert **False** liefert, wird die Wirkung der Umschalttaste beim nächsten Öffnen dieser Accessdatenbank unterbunden.

Ribbonvariable fehlerresistent machen

In VBA-Projekten von Access-Datenbanken (und in VBA im Allgemeinen) gibt es das Problem, dass das Auftreten von unbehandelten Laufzeitfehlern dazu führt, dass Objektvariablen geleert werden. Das ist insbesondere dann nachteilig, wenn Sie mit dem Ribbon arbeiten und dieses zwischendurch mit der Invalidate-Methode ungültig machen müssen, damit die Attribute mit get...-Prozeduren erneut eingelesen werden können. Der Aufruf von Invalidate führt dann unweigerlich zu einem Laufzeitfehler. Dieser Beitrag beschreibt das grundlegende Beispiel und liefert eine Lösung, mit der Sie sich keine Sorgen mehr um Objektvariablen machen müssen, die durch Laufzeitfehler geleert werden.

Leeren von Objektvariablen nach Laufzeitfehlern reproduzieren

Als Erstes schauen wir uns an, mit welchem Problem wir es überhaupt zu tun haben. Das Problem betrifft nicht nur die Variablen, die in Zusammenhang mit dem Ribbon erstellt werden. Allerdings haben Objektvariablen üblicherweise eine überschaubare Gültigkeitsdauer – sie werden in der Regel innerhalb einer Prozedur deklariert und verlieren ihre Gültigkeit nach dem Beenden der Prozedur auch wieder.

Grundsätzlich ist es ohnehin nicht empfehlenswert, mit global deklarierten Variablen zu arbeiten, da diese nicht nur von überall gelesen, sondern auch von überall geändert werden können. Dabei treten schnell Probleme auf, wenn man von mehreren Stellen aus auf solche Variablen zugreift und nicht genau weiß, was man da tut.

Spätestens, wenn mal jemand anderer die Anwendung übernimmt und anpasst und sich nicht bewusst ist, das er dort mit global deklarierten Variablen arbeitet, könnte es zu Problemen kommen.

Beispiel für Datenverlust nach Laufzeitfehlern

Schauen wir uns also zunächst einmal den grundsätzlichen Effekt an, um zu sehen, womit wir es zu tun haben. Dazu deklarieren wir in einem Standardmodul die folgende Variable:

```
Public db As DAO.Database
```

db ist nun öffentlich deklariert und kann von überall innerhalb der Anwendung gesetzt und gelesen werden. Wir setzen diese in einer kleinen Prozedur im gleichen Modul:

```
Public Sub DbSetzen()
    Set db = CurrentDb
End Sub
```

Wir können nun im Direktbereich von Access über die Objektvariable **db** auf das referenzierte Objekt zugreifen und so beispielsweise den Pfad der aktuellen Anwendung ermitteln:

```
? db.Name
C:\Users\...\RibbonvariableFehlersicherMachen.accdb
```

Nun provozieren wir einen unbehandelten Laufzeitfehler, indem wir die folgende Prozedur auslösen:

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

Dies ruft die Fehlermeldung aus Bild 1 hervor. Klicken wir hier auf **Beenden**, vervollständigen wir den unbehandelten Laufzeitfehler. Durch einen Klick auf **Debuggen** und gege-

benenfalls Änderungen im Code könnten wir den Fehler behandeln, was wir aber an dieser Stelle nicht wollen. Stattdessen klicken wir schlicht auf **Beenden**.

Dies führt zunächst einmal zu keiner sichtbaren Folge – davon abgesehen, dass die Prozedur an dieser Stelle einfach abgebrochen wird.

Es gibt jedoch noch einen weiteren Nebeneffekt: Die Variable **db** ist nun leer und zeigt nicht mehr auf das soeben referenzierte Objekt des Typs **Database**. Wie können wir das herausfinden? Indem wir einfach nochmal versuchen, den Pfad der aktuellen Anwendung im Direktbereich auszugeben.

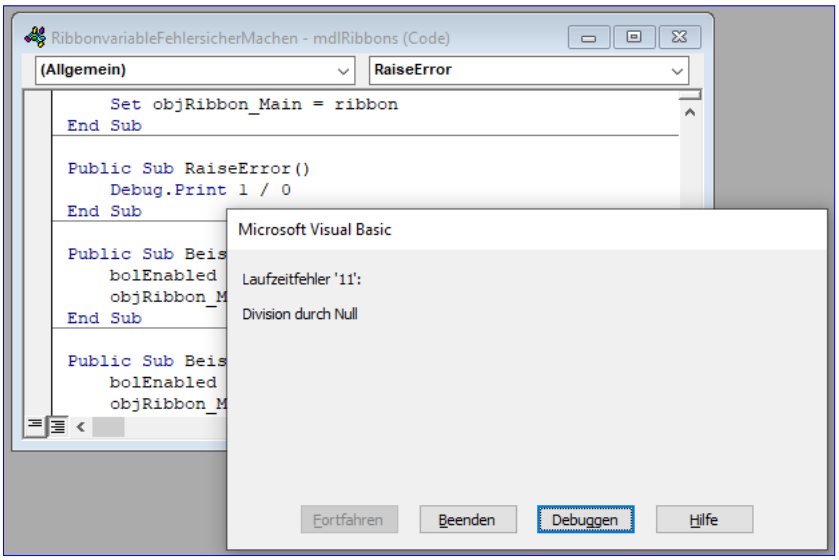


Bild 1: Unbehandelter Laufzeitfehler

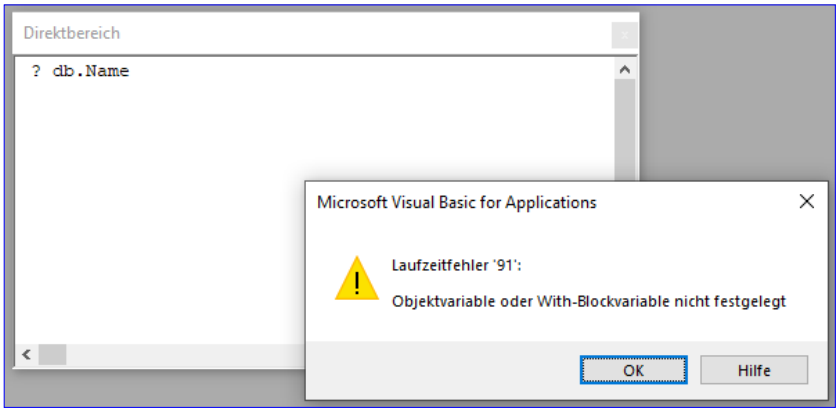


Bild 2: Die Variable **db** ist geleert.

Dies führt nun zu dem Fehler aus Bild 2. Die Variable enthält also nicht mehr den Verweis auf das **Database**-Objekt.

Wozu globale Ribbonvariablen?

Doch zurück zur eigentlichen Problemstellung, die solche Variablen betrifft, die global deklariert sind und Verweise auf Ribbon-Definitionen speichern.

Warum machen wir das überhaupt? Nun: Der Verweis auf eine Ribbon-Definition wird nur einmal beim Anwenden der Ribbon-Definition geliefert, und zwar mit der Prozedur, die Sie für das Ereignisattribut **onLoad** hinterlegen können. Diese enthält einen Parameter namens **ribbon** mit dem Datentyp **IRibbonUI**.

Damit wir später auf diese Variable zugreifen können, müssen wir diese als globale Variable speichern. Dazu verwenden wir die folgende Variable, die wir in einem Standardmodul beispielsweise namens **mdlRibbons** deklarieren:

```
Public objRibbon_Main As IRibbonUI
```

In der Ribbon-Definition legen wir für das Element **CustomUI** ein Attribut namens **onLoad** an, das den Namen der Prozedur enthält, die beim Laden der Ribbondefinition ausgeführt werden soll – außerdem hinterlegen wir für das **button**-Element **btn** noch ein Callbackattribut namens **getEnabled**, mit dem wir gleich prüfen können, ob die **Invalidate**-Methode des Ribbons funktioniert:

```
<?xml version="1.0"?>
<customUI ... onLoad="OnLoad_Main">
  <ribbon>
```

```
    <tabs>
      <tab id="tab" label="Beispieltab">
        <group id="grp" label="Beispielgruppe">
          <button id="btn" getEnabled="getEnabled"
            label="Beispielbutton"/>
        </group>
      </tab>
    </tabs>
  </ribbon>
</customUI>
```

Die Prozedur für das Attribut **onLoad** enthält lediglich eine Anweisung, die den mit dem Parameter **ribbon** übergebenen Verweis in der Variablen **objRibbon_Main** speichert:

```
Sub onLoad_Main(ribbon As IRibbonUI)
    Set objRibbon_Main = ribbon
End Sub
```

Außerdem hinterlegen wir im Modul **mdlRibbons** noch eine Variable für den **Enabled**-Zustand der Schaltfläche **btn**:

```
Public bolEnabled As Boolean
```

Für das Attribut **getEnabled** des **button**-Elements **btn** hinterlegen wir die folgende Prozedur, welche mit dem Parameter **enabled** den Wert der Variablen **bolEnabled** zurückgibt:

```
Sub getEnabled(control As IRibbon-
Control, ByRef enabled)
    enabled = bolEnabled
End Sub
```

Testen, ob die Ribbon-Variable bei Fehler geleert wird

Wenn Sie die Ribbon-Definition wie hier beschrieben und in der Beispielanwendung umgesetzt angelegt haben, sollte nach einem Neustart der Anwendung das Tab mit der Beschriftung **Beispieltab** mit einer Gruppe **Beispielgruppe** und einer Schaltfläche **Beispielbutton** erscheinen (siehe Bild 3).

Die Schaltfläche ist nicht aktiviert, da die Variable **bolEnabled** standardmäßig den Wert **False** hat und noch nicht auf **True** eingestellt wurde.

Also liefert die Prozedur **getEnabled** den Wert **False** für den Parameter **enabled**, der dann an das aufrufende **button**-Element weitergegeben wird.

Nun probieren wir, ob auch das Zuweisen des **IRibbonUI**-Objekts funktioniert hat. Dazu rufen wir die folgende Prozedur auf. Diese stellt zunächst **bolEnabled** auf **True** ein. Beim nächsten Aufruf von **getEnabled** sollte also die Schaltfläche **btn** aktiviert werden. Außerdem ruft die Prozedur die **Invalidate**-Methode von **objRibbon_Main** auf:

```
Public Sub BeispielbuttonAktivieren()
    bolEnabled = True
    objRibbon_Main.Invalidate
End Sub
```

Dies führt dazu, dass die Schaltfläche **btn** nun aktiviert dargestellt wird (siehe Bild 4). Mit der folgenden Prozedur können Sie diese wieder deaktivieren:

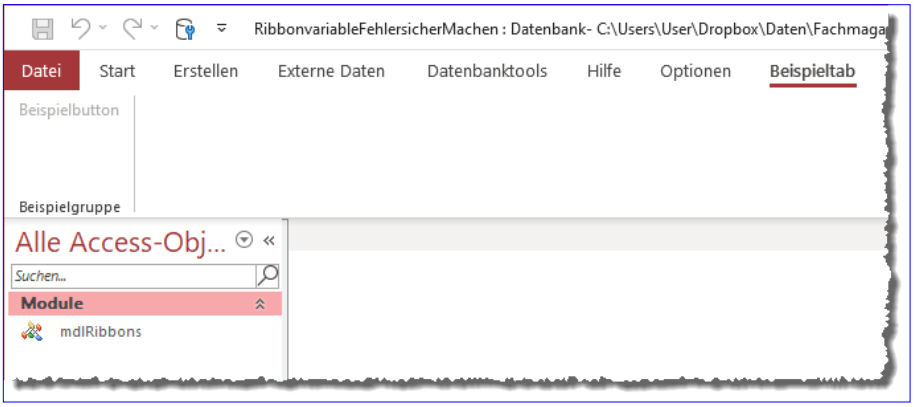


Bild 3: Das Beispielribbon in Aktion

Bei Tab-Wechsel im Ribbon ein Formular anzeigen

Microsoft verwöhnt den Office-Entwickler nicht gerade mit Ereignissen im Ribbon. Es gibt ein bis zwei Ereignisattribute für einige Steuerelemente, eines, das beim Anzeigen einer Ribbondefinition ausgeführt wird – und das war es schon fast. Was aber, wenn Sie andere Ereignisse benötigen? Zum Beispiel, um beim Wechsel von tab-Elementen ein spezielles Formular für das jeweilige tab-Element einzublenden? Hierfür gibt es kein eingebautes Ereignis. Also müssen wir ein wenig improvisieren. Wie das gelingt, zeigen wir in diesem Beitrag!

Ausgangsfrage

Dieser Beitrag ist eine Antwort auf eine Leseranfrage: Der Leser wünschte sich, dass beim Anklicken eines Tabs im Ribbon immer direkt ein passendes Formular geöffnet wird.

Wir haben daraus folgende Beispielanforderung abgeleitet:

Unser Ribbon hat drei Tabs namens **tab1**, **tab2** und **tab3**. Dazu enthält die Lösung drei Formulare namens **frm1**, **frm2** und **frm3**. Direkt beim Öffnen der Anwendung wird das tab-Element **tab1** angezeigt und damit auch das Formular **frm1** (siehe Bild 1).

Wenn der Benutzer auf das **tab**-Element **tab2** klickt, soll sich direkt das Formular **frm2** öffnen (siehe Bild 2), klickt er auf **tab3**, soll das Formular **frm3** erscheinen.

Ist eines der Formulare bereits geöffnet und der Benutzer klickt auf ein **tab**-Element, das gerade nicht aktiviert ist, dann soll das entsprechende Formular aktiviert werden.

Problem: kein passendes Ereignisattribut

Wenn wir uns nun die Attribute des **tab**-Elements ansehen, stellen wir fest, dass dieses kein Attribut anbietet, für das wir beispielsweise beim Anzeigen oder Aktivieren eines **tab**-Elements ein Ereignis auslösen können. Generell bietet das Ribbon quasi keine Ereignisse, die durch

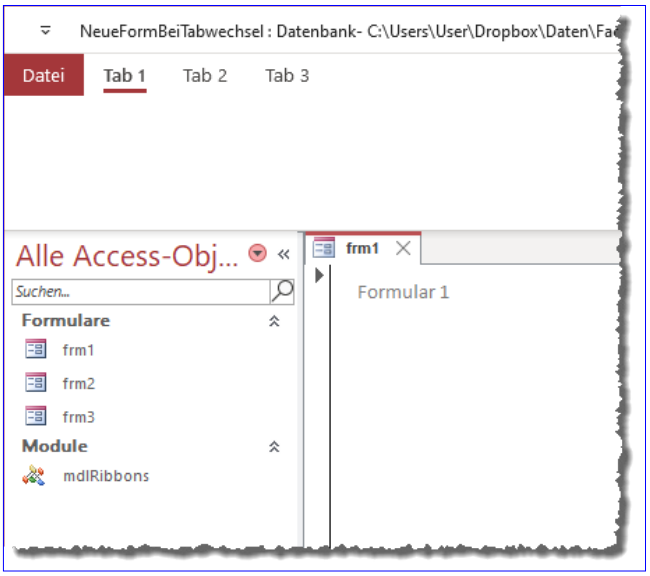


Bild 1: Anzeige des ersten Formulars gleich beim Öffnen

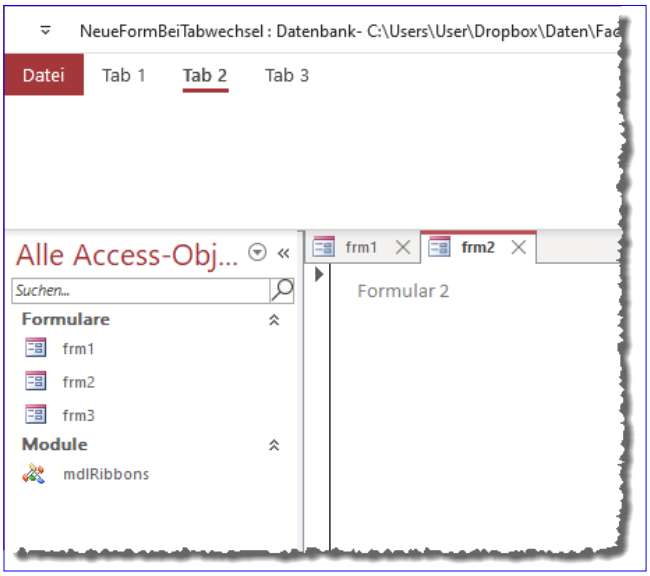


Bild 2: Anzeige des zweiten Formulars beim Klick auf das zweite Tab

```
<?xml version="1.0"?>
<customUI xmlns="http://schemas.microsoft.com/office/2009/07/customui" onLoad="OnLoad_Main">
  <ribbon startFromScratch="true">
    <tabs>
      <tab id="tab1" tag="1" label="Tab 1">
        <group id="grp1" label="Beispielgruppe 1">
          <button label="Beispielschaltfläche 1" getVisible="getVisible" id="btn1" tag="frm1"/>
        </group>
      </tab>
      <tab id="tab2" tag="2" label="Tab 2">
        <group id="grp2" label="Beispielgruppe 2">
          <button label="Beispielschaltfläche 2" getVisible="getVisible" id="btn2" tag="frm2"/>
        </group>
      </tab>
      <tab id="tab3" tag="3" label="Tab 3">
        <group id="grp3" label="Beispielgruppe 3">
          <button label="Beispielschaltfläche 3" getVisible="getVisible" id="btn3" tag="frm3"/>
        </group>
      </tab>
    </tabs>
  </ribbon>
</customUI>
```

Listing 1: Definition des Ribbons zum automatischen Öffnen von Formularen

das Anklicken oder Wechseln eines **tab**-Elements ausgelöst werden. Das **tab**-Element bietet nur die folgenden **get...**-Callbackattribute:

- **getKeytip**
- **getLabel**
- **getVisible**

Diese Attribute erwarten Prozeduren, welche die Werte für die Attribute **Keytip**, **Label** und **Visible** des **tab**-Elements liefern.

Diese Prozeduren werden dann aufgerufen, wenn die **tab**-Elemente erstmalig angezeigt werden oder nachdem die **Invalidate**-Methode für die Ribbondefinition oder die **InvalidateControl**-Methode für ein einzelnes Ribbonelement aufgerufen wurde.

Das **customUI**-Element bietet ein Ereignisattribut namens **onLoad** an, aber das wird nur einmalig beim Laden der Ribbon-Definition ausgelöst.

Das hilft alles nur wenig weiter – wir benötigen ja schließlich ein Ereignis, das immer beim Aktivieren eines **tab**-Elements ausgelöst wird, damit wir dann das jeweilige Formular anzeigen können.

Kombiniere, kombiniere ...!

Da die **get...**-Callbackattribute immer beim ersten Anzeigen ausgelöst werden (oder beim erneuten Anzeigen, wenn zwischendurch die **Invalidate**- oder die **InvalidateControl**-Methode aufgerufen wurde), könnten wir sie eigentlich auch nutzen! Wir müssen hier nur den Befehl zum Öffnen des jeweiligen Formulars unterbringen und dafür sorgen, dass die **tab**-Elemente immer nach dem Auswählen eines anderen **tab**-Elements wieder »invalidiert« werden.

Allerdings kann es sein, dass Sie die drei **get...**-Attribute des **tab**-Elements für andere Zwecke benötigen.

Also fügen wir einfach zu jedem **tab**-Element ein **button**-Element ein, wobei das **button**-Element ausschließlich der Anzeige des Formulars zum jeweiligen **tab**-Element dient.

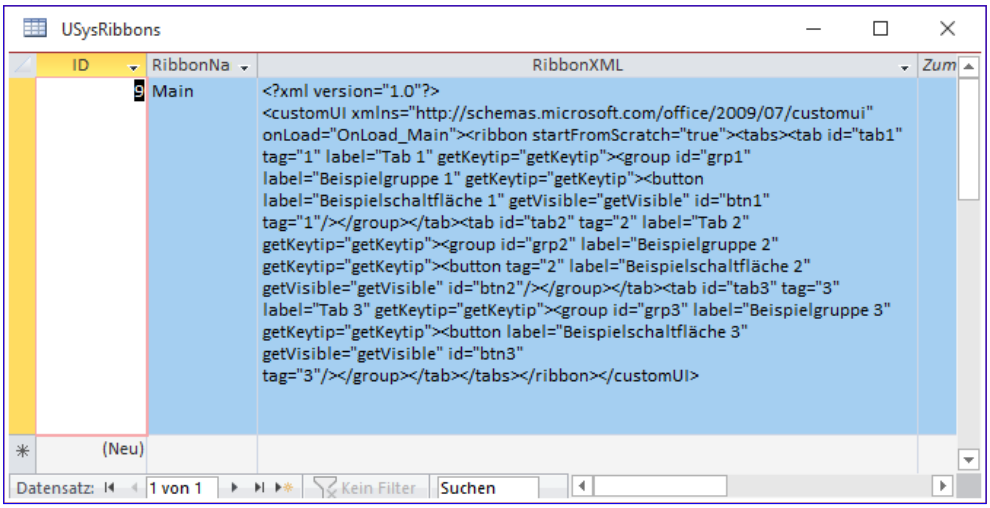


Bild 3: Tabelle zum Speichern des Ribbons

Die Ribbon-Definition gestalten wir wie in Listing 1.

Für ein einzelnes **tab**-Element sieht das so aus:

```
<tab id="tab1" tag="1" label="Tab 1">
  <group id="grp1" label="Beispielgruppe 1">
    <button label="Beispielschaltfläche 1"
      getVisible="getVisible" id="btn1" tag="frm1"/>
  </group>
</tab>
```

Das wichtigste Element ist hier das **button**-Element mit seinem **getVisible**-Attribut. Davon abgesehen, dass es uns hilft, das passende Formular zum **tab**-Element zu öffnen, hat das **button**-Element keine Funktion.

Beachten Sie auch, dass wir für alle **button**-Elemente das **tag**-Attribut gesetzt haben. Dieses nimmt jeweils den Namen des Formulars auf, das beim Einblenden des **tab**-Elemente geöffnet werden soll. Wie dieses ausgewertet wird, sehen wir weiter unten.

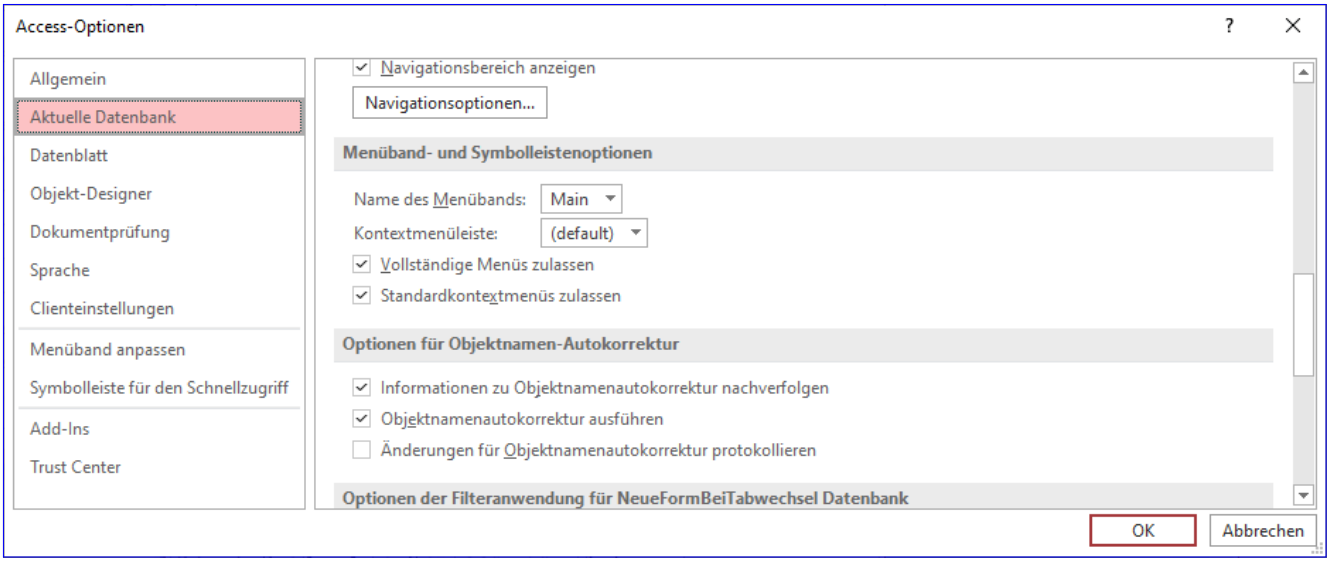


Bild 4: Festlegen des Ribbons als Anwendungsribbon

Formulare per VBA erstellen

Warum sollte man Formulare per VBA erstellen, wenn Microsoft Access doch die gute alte, etwas in die Jahre gekommene Entwurfsansicht dafür bereitstellt? Ganz einfach: Weil es für den effizient arbeitenden Access-Entwickler immer wieder Aufgaben gibt, die er einfach nicht von Hand erledigen möchte. Oder weil der Access-Entwickler immer wiederkehrende Aufgaben in ein Access-Add-In oder ein COM-Add-In auslagern möchte. Und dort gibt es nun einmal keine Entwurfsansicht – dort ist VBA-Code gefragt, um neue Formulare zu erstellen und die gewünschten Steuerelemente auf das Formular zu bringen. Dieser Beitrag liefert alle Techniken, die zum Erstellen von Formularen und zum Ausstatten mit Steuerelementen notwendig sind.

Neues, leeres Formular erstellen

Mit einer ersten Funktion wollen wir ein neues, leeres Formular mit einem vorher festgelegten Namen erstellen. Das erledigt die Funktion **CreateNewForm**. Diese erwartet als Parameter den Namen, den das Formular nach dem Erstellen erhalten soll:

```
Public Sub CreateNewForm(strName As String)
    Dim frm As Form
    Dim strNameTemp As String
    Set frm = CreateForm()
    strNameTemp = frm.Name
    DoCmd.Close acForm, frm.Name, acSaveYes
    DoCmd.Rename strName, acForm, strNameTemp
End Sub
```

Die Prozedur deklariert Variablen für das neue **Form**-Element (**frm**) und für den temporären von Access beim Erstellen vergebenen Namen (**strNameTemp**). Dann erstellt es mit der **CreateForm**-Funktion ein neues Formular. **CreateForm** liefert einen Verweis auf das neu erstellte **Form**-Objekt zurück, welches wir mit der Variablen **frm** referenzieren. Wir würden nun gern den Namen des Formulars einstellen, aber das ist nicht ohne weiteres möglich: Die Eigenschaft **Name** des **Form**-Objekts ist nämlich schreibgeschützt. Wie also dem Formular den gewünschten Namen geben? Klar: Wir könnten es einfach nach dem Speichern und Schließen umbenennen, dafür gibt es die **DoCmd.Rename**-Methode. Allerdings erwartet die auch den vorherigen Namen des Formulars, und den kennen wir nicht. Jedoch können wir diesen zuvor mit **frm.Name** auslesen und speichern ihn in der Variablen **strNameTemp**. Danach schließen wir dann das Formular mit der Methode **DoCmd.Close** und den Parametern **acForm** für den Objekttyp, **frm.Name** für den Namen des zu schließenden Objekts und **acSaveYes**, damit die Änderungen an dem neu erstellten Formular ohne Rückfrage gespeichert werden.

Nach dem Schließen rufen wir dann **DoCmd.Rename** auf und übergeben mit dem ersten Parameter den neuen Namen, mit dem zweiten den Objekttyp (**acForm**) und mit dem dritten den vorherigen Objektnamen.

Rufen wir diese Prozedur nun wie folgt auf, legt dies ein neues, leeres Formular unter dem angegebenen Namen an, das direkt im Navigationsbereich erscheint:

```
CreateNewForm "frmNeuesFormular"
```

Formular bereits vorhanden?

Wenn man diese Prozedur zwei Mal hintereinander mit dem gleichen Formularnamen aufruft, erhält man die Meldung aus Bild 1. Diese wird durch den Versuch ausgelöst, dem neuen Formular den Namen eines bereits vorhandenen Formulars zu übergeben. Dem können wir

bereits zuvor vorbeugen, indem wir abfragen, ob bereits ein Formular dieses Namens vorhanden ist. Ob ein Formular bereits vorhanden ist, können wir auf verschiedene Arten prüfen. Wir könnten zum Beispiel versuchen, es zu öffnen, oder wir durchlaufen eine der Auflistungen der Formulare.

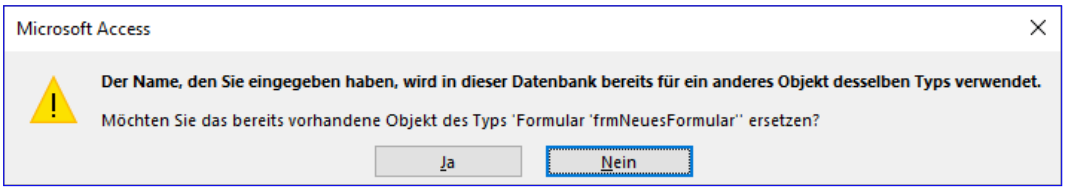


Bild 1: Meldung beim Versuch, einem Formular einen bereits vorhandenen Namen zu geben

Wir wählen letztere Variante. Die Funktion heißt **ExistsForm** und erwartet den Formularnamen als Parameter. Sie liefert das Ergebnis als Booleanwert zurück. Hier deklarieren wir für die Formulare eine Variable des Typs **AccessObject**. Warum nun **AccessObject** und nicht **Form** wie oben in der Prozedur **CreateNewForm**? Weil die dort verwendete Funktion **CreateForm** ein Objekt des Typs **Form** erstellt, die Auflistung **CurrentProject.AllForms**, die wir durchsuchen wollen, jedoch Elemente des Typs **AccessObjects** liefert.

Mit **objForm** durchlaufen wir in einer **For Each**-Schleife alle Elemente der Auflistung **CurrentProject.AllForms** und prüfen, ob das aktuell mit **objForm** referenzierte Formular den mit **strForm** übergebenen Namen hat. Falls ja, stellen wir den Funktionswert auf **True** ein und verlassen die Funktion. Wenn die Funktion alle Elemente aus **CurrentProject.AllForms** durchläuft, ohne dass das passende Formular gefunden wird, liefert sie den Rückgabewert **False** (also den Standardwert für Rückgabewerte des Datentyps **Boolean**):

```
Public Function ExistsForm(strForm As String) As Boolean
    Dim objForm As AccessObject
    For Each objForm In CurrentProject.AllForms
        If objForm.Name = strForm Then
            ExistsForm = True
            Exit Function
        End If
    Next objForm
End Function
```

Formular neu erstellen?

Wenn das Formular bereits existiert, können wir den Benutzer fragen, ob das vorhandene Formular überschrieben werden soll. Ist das der Fall, löschen wir das Formular und erstellen es neu. Dazu erweitern wir die Prozedur **CreateNewForm** wie in Listing 1. Hier fügen wir direkt hinter der Deklaration der Variablen einen Aufruf der Funktion **ExistsForm** ein.

Liefert diese den Wert **True**, dann zeigen wir ein Meldungsfenster an, das den Benutzer fragt, ob er das Formular überschreiben möchte. Falls ja, rufen wir eine Funktion namens **DeleteForm** auf (siehe weiter unten) und übergeben den Namen des zu löschenden Formulars. Diese könnte fehlschlagen, weil beispielsweise das zu löschende Formular gerade geöffnet ist. In diesem Fall soll die Prozedur einfach verlassen werden – die passende Fehlermeldung soll die Funktion **DeleteForm** liefern. Wenn der Benutzer die Frage, ob das vorhandene Formular gelöscht werden soll, mit **Nein** antwortet, soll die Prozedur auch verlassen werden.

Nur wenn das Formular noch nicht existiert oder gelöscht werden konnte, werden die anschließenden Schritte zum Erstellen des neuen Formulars ausgeführt.

Löschen eines vorhandenen Formulars

Wenn das Formular bereits vorhanden ist und der Benutzer es überschreiben möchte, müssen wir es löschen. Dazu haben wir die Funktion **DeleteForm** definiert, die den Namen des zu löschenden Formulars entgegennimmt und einen **Boolean**-Wert zurückliefert.

Die Funktion probiert bei deaktivierter eingebauter Fehlerbehandlung aus, ob das Formular mit **DoCmd.DeleteOb-**

```
Public Function CreateNewForm(strName As String) As Boolean
    Dim frm As Form
    Dim strNameTemp As String
    If ExistsForm(strName) Then
        If MsgBox("Formular '" & strName & "' ist bereits vorhanden. Überschreiben?", vbYesNo + vbExclamation, _
                "Formular überschreiben?") = vbYes Then
            If DeleteForm(strName) = False Then
                Exit Function
            End If
        Else
            Exit Function
        End If
    End If
    Set frm = CreateForm()
    strNameTemp = frm.Name
    frm.Visible = True
    DoCmd.Close acForm, frm.Name, acSaveYes
    DoCmd.Rename strName, acForm, strNameTemp
     CreateNewForm = True
End Function
```

Listing 1: Erstellen eines neuen Formulars

ject gelöscht werden kann. Falls ja, erhält die Funktion den Rückgabewert **True**. Anderenfalls gibt die Funktion die entsprechende Fehlermeldung aus **Err.Description** aus und gibt den Standardwert **False** als Funktionsergebnis an die aufrufende Prozedur zurück:

```
Public Function DeleteForm(strName As String) As Boolean
    On Error Resume Next
    DoCmd.DeleteObject acForm, strName
    If Err.Number = 0 Then
        DeleteForm = True
    Else
        MsgBox Err.Description, vbCritical + vbOKOnly, _
            "Löschen fehlgeschlagen"
    End If
End Function
```

Damit haben wir bereits eine Funktion zum Erstellen eines neuen Formulars, das allerdings noch komplett leer ist und für das wir auch noch keine Eigenschaften eingestellt haben.

Formular weiterverarbeiten

Nun stehen zwei Aufgaben an: Die erste ist das Einstellen verschiedener Eigenschaften für das Formular. Die zweite ist das Erstellen von Steuerelementen im Formular. Letzteres beschreibt der bald erscheinende Beitrag **Steuerelemente erstellen** (**www.access-im-unternehmen.de/1336**). Um die Eigenschaften kümmern wir uns jetzt gleich.

Um Eigenschaften einstellen zu können, benötigen wir einen Verweis auf das Formular in der Entwurfsansicht. Das heißt also, dass wir das Formular auch öffnen müssen. Da stellt sich die Frage: Könnten wir mit der Funktion **CreateNewForm** nicht direkt einen Verweis auf das neu erstellte und in der Formularansicht geöffnete Formular zurückgeben? Das könnten wir tun, wir müssten dazu das Formular allerdings auch in dieser Funktion wieder öffnen.

Das Schließen ist dort unbedingt nötig, da wir es sonst nicht umbenennen können. Da wir nicht wissen, ob und wie das Formular nach dem Erstellen direkt weiterverarbeitet werden soll, geben wir einfach nur einen Boole-

an-Wert zurück, der angibt, ob das Formular erfolgreich erstellt wurde.

Eigenschaften des Formulars einstellen

In der Prozedur, welche die Funktion **CreateNewForm** aufgerufen hat, können wir nun das Formular in der Entwurfsansicht öffnen und die gewünschten Änderungen vornehmen.

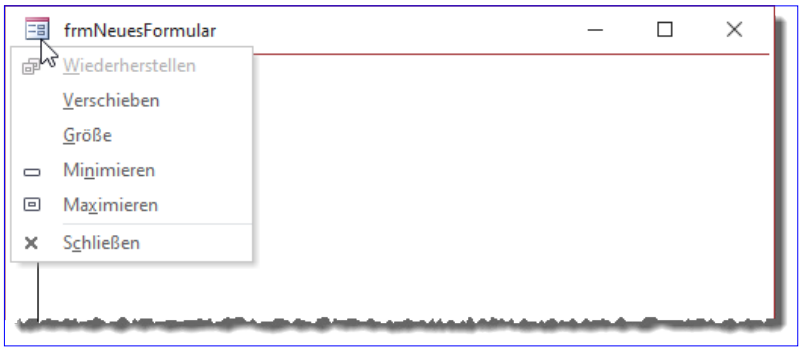


Bild 2: Elemente der Titelleiste eines Formulars

Wie bereits erwähnt, wollen wir uns in diesem Beitrag zunächst um das Einstellen der verschiedenen Eigenschaften des Formulars kümmern und schauen uns diese dabei im Detail an. Das Öffnen gestalten wir wie folgt:

```
Public Sub Test_CreateNewForm()
    Dim frm As Form
    Dim strForm As String
    strForm = "frmNeuesFormular"
    If CreateNewForm(strForm) = True Then
        DoCmd.OpenForm strForm, acDesign
        Set frm = Forms(strForm)
        With frm
            'Eigenschaften einstellen
        End With
    End If
End Sub
```

Hier schreiben wir den Namen des zu erstellenden Formulars direkt in eine Variable, weil wir diesen mehr als einmal benötigen und anschließende Änderungen so nur an einer Stelle erfolgen müssen. Dann erstellen wir das neue Formular mit der Funktion **CreateNewForm**.

Ist dies erfolgreich, öffnen wir das neu erstellte Formular mit **DoCmd.OpenForm** in der Entwurfsansicht. Dazu stellen wir den zweiten Parameter **View** auf **acDesign** ein.

Damit wir danach komfortabel die Eigenschaften des Formulars einstellen können, referenzieren wir das Formular dann über die **Forms**-Auflistung und den Namen des Formulars mit der **Form**-Variablen **frm**. Für **frm** stellen wir dann die nachfolgend beschriebenen Eigenschaften ein.

Einstellungen für die Titelleiste

Die Titelleiste des Formulars (siehe Bild 2) verwendet gleich mehrere Eigenschaften.

- Die Titelleiste des Formulars stellen wir mit der Eigenschaft **Caption** ein.
- Ob das Formular das gleiche Icon anzeigen soll wie die Hauptanwendung, legen Sie übrigens nicht mit einer Formular-Eigenschaft fest, sondern in den Access-Optionen mit der Option **Als Formular- und Berichtssymbol verwenden**. Dies gilt dann für alle Formulare und Berichte.
- Ob die **Schließen**-Schaltfläche aktiviert ist, stellen Sie mit der Eigenschaft **CloseButton** ein. Legen Sie den Wert auf **False** fest, wird die Schaltfläche ausgegraut dargestellt und der Benutzer kann sie nicht betätigen.
- Die Schaltflächen in der Titelleiste zum Minimieren und Maximieren des Fensters können Sie über die Benutzeroberfläche mit der Eigenschaft **MinMaxSchaltflächen** einstellen, über VBA mit **MinMaxButtons**. Die möglichen Werte sind: **0** (**Keine**), **1** (**Min vorhanden**), **2** (**Max vorhanden**) oder **3** (**Beide vorhanden**).

- Die Eigenschaft **Mit Systemfeldmenü** (**ControlBox**) stellt ein, ob überhaupt Steuerelemente in der Titelleiste angezeigt werden sollen. Falls nicht, sieht die Titelleiste wie in Bild 3 aus.

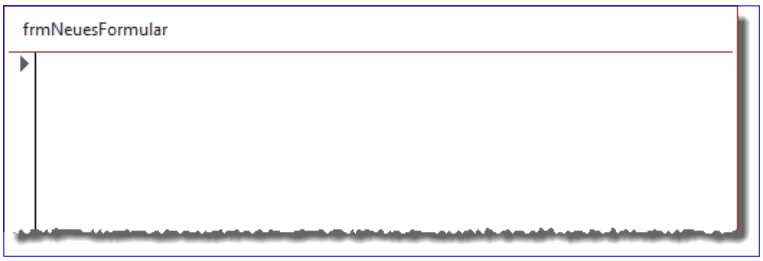


Bild 3: Titelleiste nur mit Überschrift

Abmessungen

Die Abmessungen des Formulars wie die Höhe und die Breite stellen Sie mit Eigenschaften für verschiedene Elemente ein. Die **Breite** ist eine Formular-Eigenschaft, diese entspricht der VBA-Eigenschaft **Width**. Eine Eigenschaft namens **Höhe** beziehungsweise **Height** vermissen wir allerdings im Eigenschaftenblatt. Der Grund ist einfach: Ein Formular kann mehrere Bereiche enthalten, die jeweils eine eigene Höhe haben – zum Beispiel der Detailbereich oder Formularkopf und -Fuß. Die Einstellungen dieser Bereiche besprechen wir weiter unten.

Standardansicht

Die Standardansicht stellt die Eigenschaft **DefaultView** ein. Diese nimmt die Werte der Enumeration **acDefView** entgegen:

- **acDefViewContinuous** (**1**): Endlosformular
- **acDefViewDatasheet** (**2**): Datenblatt
- **acDefViewSingle** (**0**): Einzelnes Formular
- **acDefViewSplitForm** (**5**): Geteiltes Formular

Beispiel zum Einstellen der Eigenschaft **Standardansicht** auf **Datenblatt**:

```
frm.DefaultView = acViewDatasheet
```

Erlaubte Ansichten einstellen

Wenn Sie verhindern wollen, dass die Benutzer zwischen verschiedenen Ansichten wechseln, nutzen Sie die Eigenschaft **ViewsAllowed** dazu.

Mögliche Werte:

- **0**: Formularansicht und Datenblattansicht möglich
- **1**: Kein Wechsel von der Formularansicht zur Datenblattansicht möglich
- **2**: Kein Wechsel von der Datenblattansicht zur Formularansicht möglich

Sichtbarkeit

Mit der Eigenschaft **Sichtbar** (**Visible**) können Sie das Formular ein- und ausblenden.

Eigenschaften für geteilte Formulare

Wenn das Formular über die Eigenschaft **DefaultView** als geteiltes Formular angezeigt wird, werden die folgenden Eigenschaften ausgewertet:

- **Größe des geteilten Formulars** (**SplitFormSize**) gibt je nach der mit **SplitFormOrientation** gewählten Position die Höhe oder Breite des Detailbereichs des geteilten Formulars an. Diese wird im Eigenschaftenblatt in Zentimeter und unter VBA in Twips angegeben.
- **Ausrichtung des geteilten Formulars** (**SplitFormOrientation**): Gibt an, wo das Datenblatt angezeigt werden soll. Es gibt die folgenden Werte: **acDatasheetOnBottom** (**1**), **acDatasheetOnLeft** (**2**), **acDatasheetOnRight** (**3**) und **acDatasheetOnTop** (**0**).
- **Teilerleiste des geteilten Formulars** (**SplitFormSplitterBar**): Gibt an, ob die Leiste zwischen den beiden

Formularbereichen angezeigt werden soll. Wenn der Benutzer die Höhe/Breite der Bereiche nicht verändern soll, stellen Sie diese Eigenschaft auf **False** ein.

- **Datenblatt des geteilten Formulars** (**SplitFormDatasheet**): Hier können Sie festlegen, ob Bearbeitungen im Datenblatt zulässig sein sollen. Es gibt die beiden Werte **Bearbeitungen zulassen** (**acDatasheetAllowEdits**) oder **Schreibgeschützt** (**acDatasheetReadOnly**.).

- **Drucken des geteilten Formulars** (**SplitFormPrinting**): Hiermit legen Sie fest, welcher Bereich des geteilten Formulars im Fall der Fälle gedruckt werden soll. Die Eigenschaft nimmt die Werte **Nur Datenblatt** (**acGridOnly**) oder **Nur Formular** (**acFormOnly**) entgegen.

- **Position der Teilerleiste speichern** (**SaveSplitterBarPosition**): Legt fest, ob die Position der Teilerleiste gespeichert werden soll, wenn diese beim Bearbeiten des geöffneten Formulars mit der Maus verschoben wurde. Beim Schließen wird dann abgefragt, ob die Position so wie geändert erhalten werden soll.

Einstellungen für die Anzeige des Formulars

Die folgenden Einstellungen blenden verschiedene Formularelemente ein oder aus, außerdem enthalten sie Informationen über den Zustand zu modalen und Popup-Fenstern.

Die nachfolgend beschriebenen Elemente finden Sie teilweise im Screenshot aus Bild 4.

- **Automatisch zentrieren** (**AutoCenter**): Legt fest, ob das Formular beim Öffnen automatisch im Access-Fenster zentriert werden soll.

- **Datensatzmarkierer** (**RecordSelectors**): Aktiviert die Anzeige des Datensatzmarkierers (siehe Screenshot unter 1)

- **Navigationsschaltflächen** (**NavigationButtons**): Aktiviert die Anzeige der Navigationsschaltflächen (siehe Screenshot unter 3).

- **Navigationsbeschriftung** (**NavigationCaption**): Stellt die Beschriftung der Navigationsschaltflächen ein. Diese Beschriftung wird dort angezeigt, wo normalerweise **Datensatz:** steht (siehe Screenshot unter 2).

- **Trennlinien** (**DividingLines**): Legt fest, ob im Endlosformular Trennlinien zwischen den Datensätzen angezeigt werden.

- **Bildlaufleisten** (**ScrollBars**): Stellt ein, ob Bildlaufleisten angezeigt werden sollen. Es gibt die folgenden Einstellungen: **0**: **Nein**, **1**: **Nur horizontal**, **2**: **Nur vertikal** und **3**: **In beide Richtungen** (siehe Screenshot unter 4)

- **Rahmenart** (**BorderStyle**): Gibt an, welche Rahmenart für Titelzeile, Formularrahmen et cetera verwendet werden soll. Es gibt die Werte **0** (**Keine**), **1** (**Extra dünn**), **2** (**Veränderbar**) und **3** (**Dialog**).

Außerdem gibt es noch zwei Eigenschaften, welche beide auf **True** eingestellt werden, wenn Sie das Formular mit **DoCmd.OpenForm** mit dem Wert **acDialog** für den Parameter **WindowMode** öffnen:

```
DoCmd.OpenForm "frmBeispiel", WindowMode:=acDialog
```

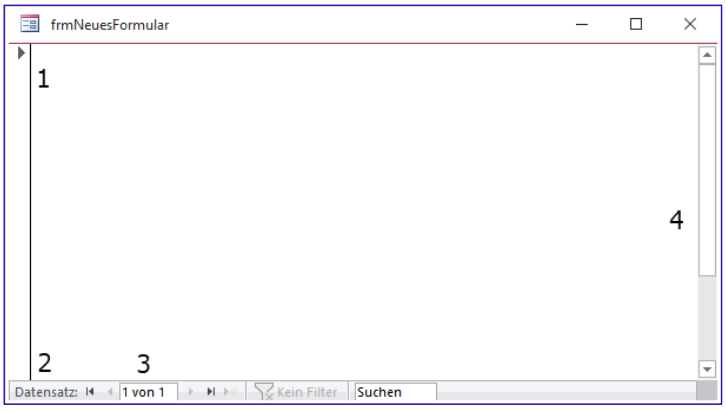


Bild 4: Ein- und ausblendbare Elemente im Formular

Die Eigenschaften lauten **Popup** (im Eigenschaftenblatt und unter VBA) und **Modal** (nur unter VBA verfügbar). Wenn Sie ein Formular wie oben öffnen, weisen diese beiden Eigenschaften den Wert **True** auf. Wenn Sie beide beim Erstellen auf **True** einstellen, wird das Formular immer als modaler Dialog geöffnet.

Ribbon und Menüs

Auch zum Festlegen des Ribbons, das mit einem Formular angezeigt werden soll, und zu den Menüs, die mit Access 2003 abgekündigt wurden, gibt es einige Eigenschaften. Das Ribbon eines Formulars stellen Sie mit der Eigenschaft **Name des Menübands** (**RibbonName**) ein. Hier geben Sie den Namen eines Eintrags der Tabelle **USysRegInfo** ein. Die übrigen Eigenschaften wollen wir nur der Vollständigkeit halber erwähnen. Mit **Menüleiste** (**MenuBar**) und **Symbolleiste** (**Toolbar**) geben Sie die mit einem Formular anzuzeigenden Menüs an. Die Eigenschaft **Kontextmenüleiste** (**ShortcutMenuBar**) hingegen können Sie auch unter Access 2007 und neueren Versionen noch verwenden. Voraussetzung ist, dass Sie beispielsweise mit dem folgenden Code eine solche Kontextmenüleiste erstellt haben:

```
Dim cbr As CommandBar
Dim cbc As CommandBarButton
On Error Resume Next
CommandBars("cbrKontextmenue").Delete
On Error GoTo 0
Set cbr = CommandBars.Add("cbrKontextmenue", 7
                                msoBarPopup, False, True)
Set cbc = cbr.Controls.Add(msoControlButton)
With cbc
    .Caption = "Beispielbutton"
End With
```

Danach können Sie das Kontextmenü aus der Liste im Eigenschaftenblatt auswählen und es auch der Eigenschaft **ShortCutMenuBar** zuweisen:

```
frm.ShortCutMenuBar = "cbrKontextmenue"
```

Mit der Eigenschaft **Kontextmenü** (**ShortcutMenu**) legen Sie fest, ob das Formular überhaupt ein Kontextmenü anzeigen soll. Damit deaktivieren Sie sowohl eingebaute als auch benutzerdefinierte Kontextmenüs.

Dateneinstellungen

Die Eigenschaft **Datensatzquelle** (**RecordSource**) stellt die Tabelle oder Abfrage ein, deren Daten im Formular angezeigt werden können.

Mit der Eigenschaft **Zyklus** (**Cycle**) stellen Sie ein, ob beim Verlassen des letzten Feldes zum ersten Feld der Aktivierreihenfolge oder umgekehrt etwa mit der Tabulator- oder der Eingabetaste der Datensatz gewechselt werden soll. Die Eigenschaft kann die Werte **0** (**Alle Datensätze**), **1** (**Aktueller Datensatz**) und **2** (**Aktuelle Seite**) annehmen.

Die Eigenschaft **Standardwerte abrufen** (**FetchDefaults**) gibt an, ob Standardwerte für Datensätze schon vor dem Speichern angezeigt werden sollen.

Datenoperationen erlauben oder verbieten

Die folgenden Eigenschaften dienen dazu, verschiedene Operationen mit Daten zu erlauben oder zu unterbinden:

- **Daten eingeben** (**DataEntry**): Legt fest, ob das Formular beim Öffnen nur einen neuen, leeren Datensatz anzeigt (**True**) oder ob alle Datensätze angezeigt werden sollen (**False**).
- **Anfügen zulassen** (**AllowAdditions**): Legt fest, ob neue Datensätze hinzugefügt werden können.
- **Löschen zulassen** (**AllowDeletions**): Legt fest, ob Datensätze gelöscht werden können.
- **Bearbeitungen zulassen** (**AllowEdits**): Legt fest, ob Datensätze bearbeitet werden können.
- **Filter zulassen** (**AllowFilters**): Legt fest, ob Filter gesetzt werden dürfen.

Setup für Access: Vertrauenswürdige Speicherorte

Christoph Jüngling, https://www.juengling-edv.de

Das in diesem Artikel beschriebene Konzept hat das Ziel, die Registrierung des Installationsverzeichnisses als vertrauenswürdigen Speicherort zu automatisieren. Dadurch entfällt die Notwendigkeit für den Anwender, dies in Access selbst einzutragen. Vor allem vermeiden wir durch die spezifische Festlegung nur eines Verzeichnisses als vertrauenswürdig das Risiko, dass der Anwender unnötig viele Unterverzeichnisse quasi nebenher als vertrauenswürdig einstuft (was sie vielleicht nicht sein sollten).

Automatisierung des Eintrags für die vertrauenswürdigen Speicherorte

Das Konzept der "vertrauenswürdigen Speicherorte" ist seit längerem für (meines Wissens) alle Office-Programme von Microsoft gängig. Wie schon im vorherigen Artikel erläutert, möchte Microsoft damit verhindern, dass irgendeine Datei mit VBA-Code auf unsere Festplatte gelangt und ohne weitere Kontrolle ausgeführt wird.

Ich habe von Leuten gehört, die der Einfachheit halber **C:** und alle Unterverzeichnisse als vertrauenswürdig einstufen, aber das scheint mir ein scheunentorgroßes Loch in die Sicherheit unseres Rechners zu schlagen. Jedes Verzeichnis, auch das Standardverzeichnis für temporäre Dateien, würde es dann erlauben, dass VBA-Makros ausgeführt werden! Das kann nicht in unserem Sinne sein.

Bei Word und Excel zum Beispiel gibt es einen Trick: die digitale Signatur von VBA-Makros. Diese sichert die VBA-Makros der betreffenden Datei einerseits gegen Veränderungen ab, andererseits macht sie den Autor des Makros überprüfbar.

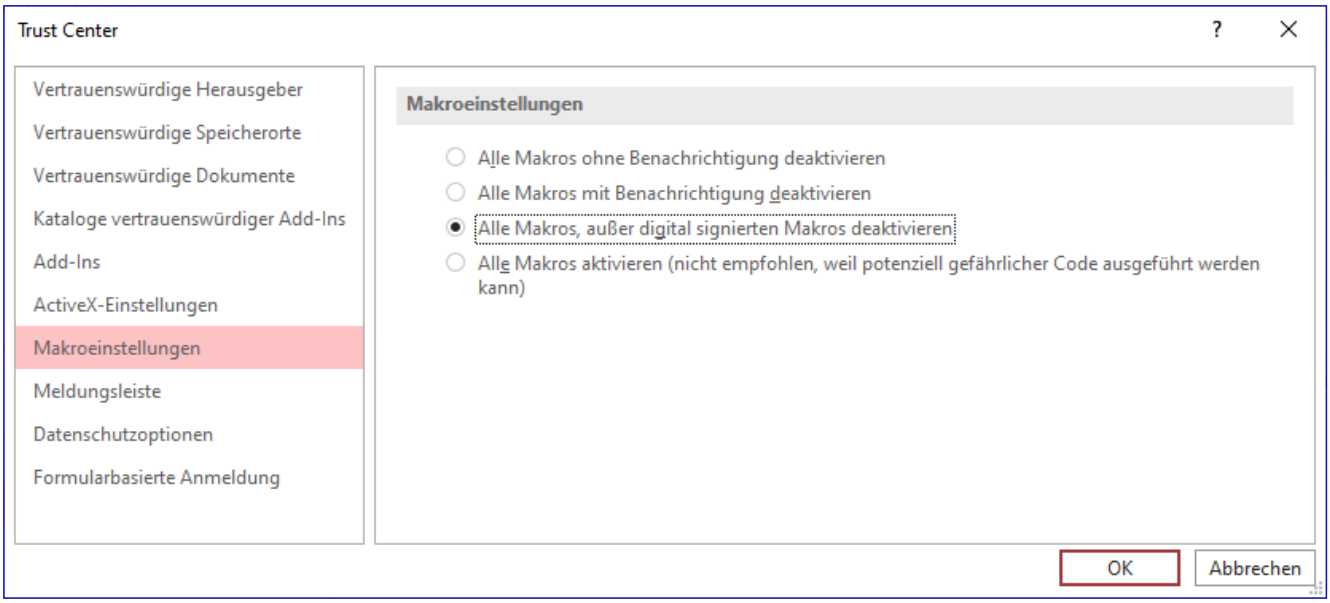


Bild 1: Trustcenter-Einstellung für VBA-Makros

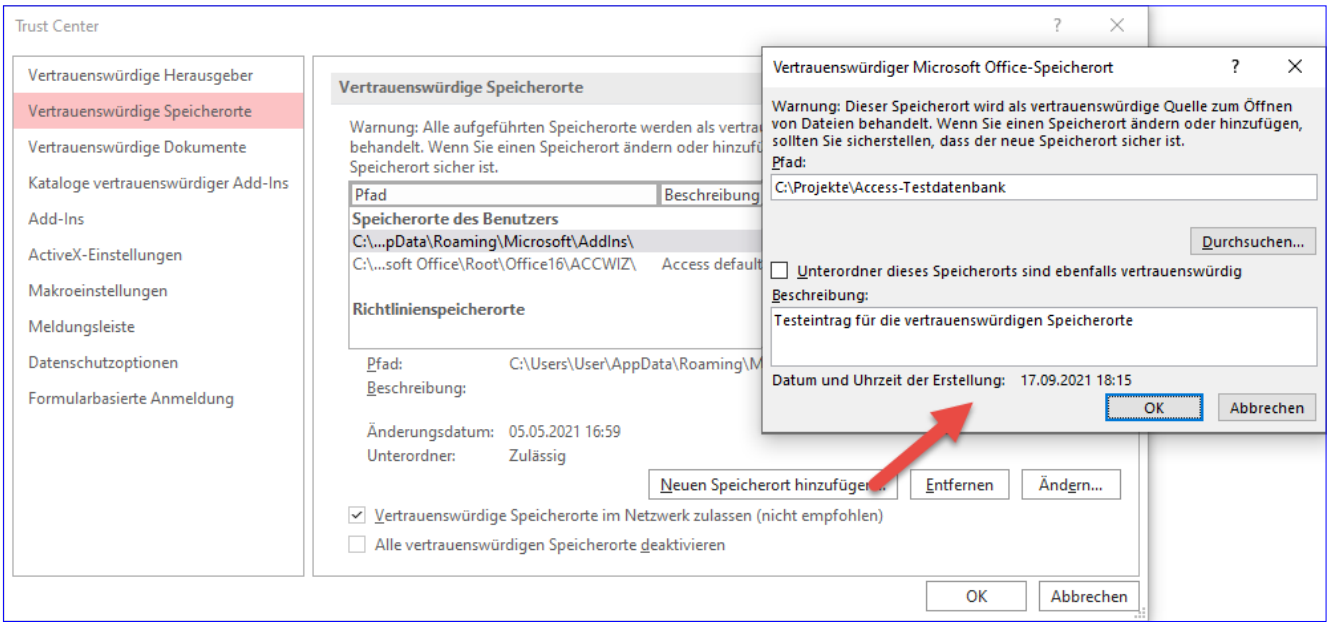


Bild 2: Eingabeformular für einen vertrauenswürdigen Speicherort

In Verbindung mit der Einstellung **Alle Makros außer digital signierten deaktivieren** (siehe Bild 1) erhält man so die Möglichkeit, die Makros aus dem eigenen Haus aktiv zu lassen, alle anderen jedoch zu blockieren.

Leider funktioniert die Verwendung einer digitalen Signatur für die VBA-Makros in Access nicht. Deshalb sind wir auf die korrekte Konfiguration der vertrauenswürdigen Speicherorte angewiesen.

Einen vertrauenswürdigen Speicherort eintragen

Schauen wir uns das zunächst bei den Einstellungen von Access an. Das ist ein wichtiger Aspekt, denn diese Einstellung kann und muss für jedes Programm und jeden Windows-Useraccount individuell vorgenommen werden.

Öffnen Sie in Access **Optionen|Trust Center|Einstellungen** für das Trustcenter. Dort finden Sie den Eintrag **Vertrauenswürdige Speicherorte**. Falls dort bei Ihnen noch nichts steht, tragen Sie ruhig mal ein Verzeichnis ein, wie in Bild 2 gezeigt. Der Haken für die Unterordner muss nicht gesetzt werden, da diese Einstellung nur unsere **.accdb**-Datei betrifft.

Unsere Eingabe wird in der Registry gespeichert, sobald wir die Dialoge mit **OK** bestätigt haben. Dies erfolgt unter dem Pfad **HKEY_CURRENT_USER\SOFTWARE\Microsoft\Office\16.0\Access\Security\Trusted Locations** (siehe Bild 3).

Die einzelnen Einträge erhalten dabei automatisch Namen wie hier im Beispiel **Location1**. Sobald ein solcher Eintrag von uns gelöscht wird, wird er auch aus der Registry entfernt.

Damit haben wir eine Möglichkeit gefunden, den vertrauenswürdigen Speicherort für unsere eigene Applikation automatisiert einzutragen. Jetzt geht es nur noch darum, das richtige Format zu finden.

Wie man oben sieht, enthält der Registry-Pfad unter anderem die interne Access-Versionsnummer. Da ich diesen Artikel mit Access 2019 vorbereitet habe, steht dort **16.0**.

Diese Versionsnummer müssen wir während unserer Setup-Ausführung ermitteln und korrekt verwenden.

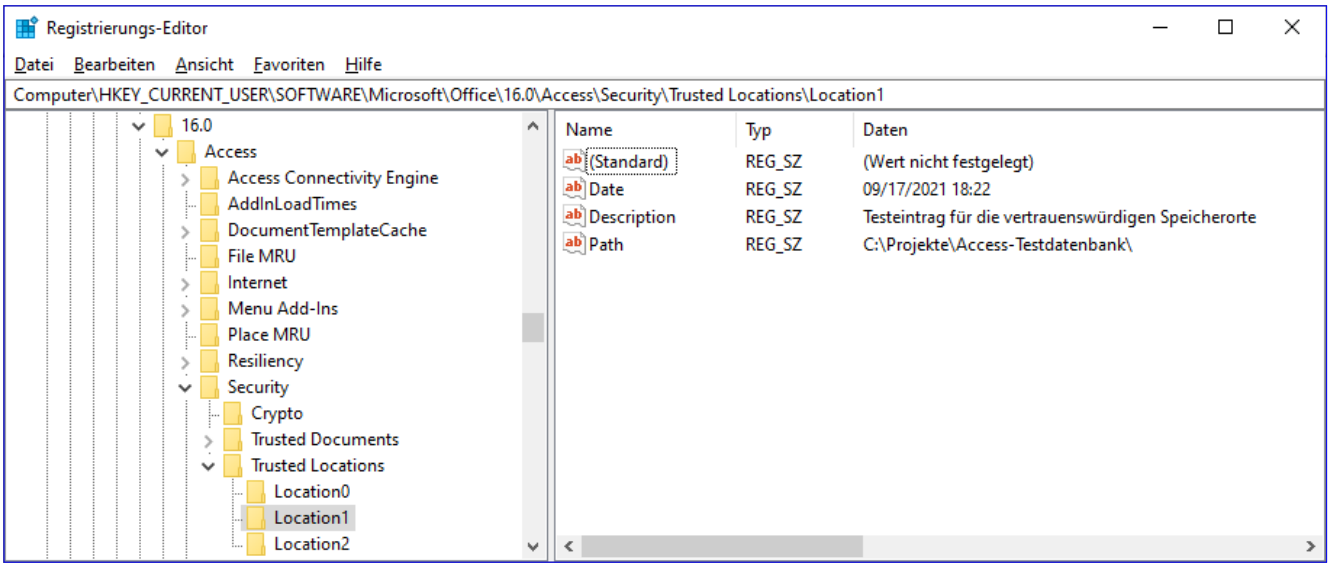


Bild 3: Registry-Eintrag eines vertrauenswürdigen Speicherorts

Die zweite Aufgabe betrifft die erkennbare Durchnummerierung der **Location**-Einträge. Doch hier gibt es eine ganz einfache Lösung: Das muss gar nicht sein. Wenn ich probeweise mal einen Eintrag umbenenne, funktioniert alles weiterhin wie gewohnt. Das heißt, dass wir zum Beispiel unsere Konstante **{#MyAppName}** dafür hernehmen können.

Pascal-Scripting in InnoSetup

Es geht bei InnoSetup schon vieles über die einfachen Einstellungen, aber irgendwann ist halt eine Grenze erreicht. In den Fällen hat InnoSetup ein sehr mächtiges Werkzeug an Bord, auf das wir nun zurückgreifen müssen: Pascal-Scripting.

Was bei Microsoft Office das VBA ist, das ist im Prinzip das Pascal-Scripting für InnoSetup. Es handelt sich im Grunde um ein Konzept, das wir von Access her auch schon kennen.

Denn hier geht es ebenfalls nicht um ein großes Programm, welches das ganze Setup steuert, sondern rein darum, auf bestimmte Ereignisse zu reagieren. Das kann zum Beispiel der Start des Setups sein oder auch der Zugriff auf bestimmte Werte für den Eintrag in die Registry.

Eingeleitet wird das Scripting durch eine Rubrikenüberschrift **[Code]**, danach wird alles anders. Ab hier gelten die Regeln einer Programmiersprache, die rein zeilenorientierte Schreibweise des bisherigen Scripts gilt nicht mehr. Aus diesem Grunde muss die Code-Rubrik auch immer die letzte im gesamten Script sein!

Welche Funktionen benötigen wir? Da ist zunächst, wie gesagt, die aktuelle Access-Version. Diese Funktion nenne ich **CurAccVer**. Des weiteren finden wir in den Daten in der Registry noch das aktuelle Datum in amerikanischer Schreibweise.

Das wäre zwar ebenfalls nicht besonders notwendig, aber es ist erstens durchaus sinnvoll, und zweitens auch nicht schwer zu ermitteln.

Diese Funktion nenne ich einfach **Now**. Schauen wir uns zunächst dieses Script an – später erkläre ich dann, wo und wie man diese Funktionen einbindet (siehe Listing 1).

Pascal und seine Besonderheiten gegenüber VB/VBA

Pascal hat bezüglich seiner Programmiersprache andere Formalien als wir es von VB/VBA her gewohnt sind, ist

Das Application-Objekt

Eines der wichtigsten Objekte bei der Programmierung von Access-Anwendungen ist das Application-Objekt. Es bietet viele Eigenschaften und Methoden, die stiefmütterlich behandelt werden. Dabei lohnt es sich, einmal einen Blick darauf zu werfen – dann weiß man im Fall der Fälle, dass es da irgendwo einen passenden Befehl geben muss. Dieser Beitrag zeigt eine Übersicht der Elemente des Application-Objekts und deren Funktion. In weiteren Beiträgen schauen wir uns die Funktion der einzelnen Eigenschaften und Methoden im Detail an.

Application-Objekt im Objektkatalog

Wenn Sie irgendwann einmal vor einer Aufgabe stehen, von der Sie wissen, dass Sie in diesem Beitrag von einer möglichen Lösung gelesen haben, können Sie schnell im Objektkatalog des VBA-Editors nachsehen – dort finden Sie zumindest eine Auflistung der Methoden und Eigenschaften des **Application**-Objekts. Den Objektkatalog öffnen Sie mit der Taste **F2**. Oben im Suchfeld geben Sie **Application** ein und klicken dann auf den gefundenen Eintrag. Die untere Liste liefert dann alle Member dieser Klasse (siehe Bild 1).

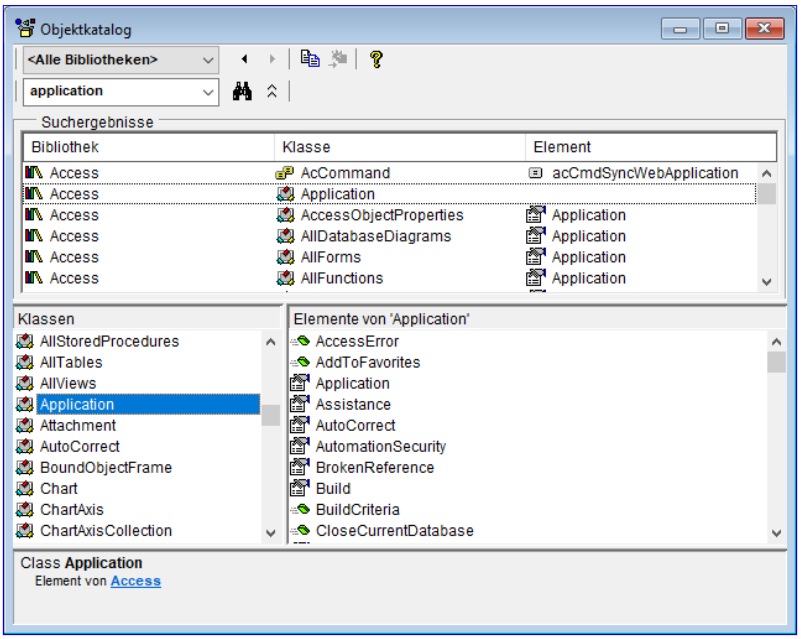


Bild 1: Elemente des **Application**-Objekts

Nachfolgend finden Sie die mehr oder weniger kurzen Beschreibungen der Elemente des **Application**-Objekts. Elemente für Webdatenbanken oder Elemente für ältere Techniken wie DDE haben wir nicht berücksichtigt. Sie können die Befehle auch ohne vorangestelltes **Application** nutzen.

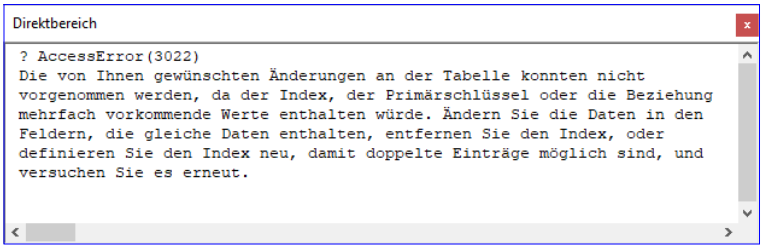


Bild 2: Ermitteln des Textes eines Access-Fehlers

AccessError

Diese Funktion erwartet eine Fehlernummer als Parameter und liefert die Beschreibung des Fehlers in der Anwendungssprache (siehe Bild 2). Praktisch, wenn ein Kunde Ihnen nur die Fehlernummer zu einem Fehler liefert und Sie schnell nachsehen wollen, welche Meldung sich dahinter verbirgt.

AddToFavorites

Hat in früheren Versionen von Access die aktuelle Datenbank zur Liste der Favoriten hinzugefügt.

Assistance

Das mit der Eigenschaft **Assistance** gelieferte Objekt bietet vier Methoden an, die Sie in Bild 3 sehen.

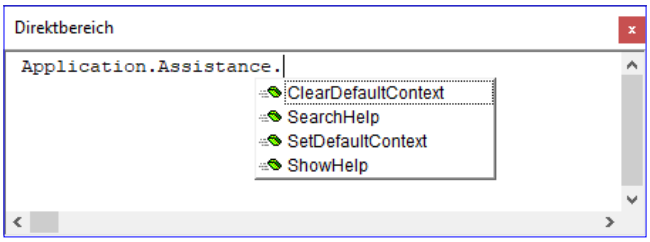


Bild 3: Methoden des **Assistance**-Objekts

SearchHelp scheint noch die nützlichste Methode zu sein, sie öffnet nach Eingabe eines Suchbegriffs als Parameter die Support-Seite von Microsoft mit den passenden Themen:

```
Application.Assistance.SearchHelp "Assistance"
```

Dies funktioniert jedoch nicht gut, denn für dieses Beispiel liefert die Seite keine Suchergebnisse.

AutoCorrect

Das mit **AutoCorrect** gelieferte Objekt bietet nur eine Eigenschaft namens **DisplayAutoCorrectOptions** an. Hiermit können Sie einstellen, ob die entsprechenden Optionen angezeigt werden sollen:

```
Application.AutoCorrect.DisplayAutoCorrectOptions = False
```

Weitere Informationen zur Autokorrektur finden Sie im Beitrag **Autokorrektur-Einträge im Griff** (**www.access-im-unternehmen.de/1279**).

AutomationSecurity

Diese Eigenschaft legt fest, in welchem Sicherheitsmodus Access andere Access-Anwendungen per VBA öffnet (auch einsetzbar von anderen Office-Anwendungen aus). Zum Öffnen wird hier **OpenCurrentDatabase** verwendet (siehe weiter unten). Es gibt drei Einstellungen:

- **msoAutomationSecurityByUI**: Verwendet die Einstellungen in den Access-Optionen (zu finden in dem Dialog, der mit **Datei|Optionen** geöffnet wird, unter **Trust Center|Einstellungen für das Trust Center|Makroeinstellungen**).
- **msoAutomationSecurityForceDisable**: Öffnet die Anwendung ohne Aktivierung von VBA, wenn die Makroeinstellung im oben genannten Dialog auf **Alle Makros ohne Benachrichtigung aktivieren** oder **Alle Makros mit Benachrichtigung aktivieren** steht. Wenn die Einstellung auf **Alle Makros aktivieren** steht, hat **msoAutomationSecurityForceDisable** keine Wirkung.
- **msoAutomationSecurityLow**: Aktiviert VBA, auch wenn die Makroeinstellung im oben genannten Dialog auf **Alle Makros ohne Benachrichtigung aktivieren** oder **Alle Makros mit Benachrichtigung aktivieren** steht.

BrokenReference

Mit dieser Eigenschaft finden Sie heraus, ob das VBA-Projekt kaputte oder nicht vorhandene Verweise enthält:

```
Public Sub Test_BrokenReference()
    Dim objReference As Access.Reference
    If Application.BrokenReference Then
        MsgBox "Es gibt kaputte Verweise."
        For Each objReference In Application.References
            If objReference.IsBroken Then
                Debug.Print objReference.Name
            End If
        Next objReference
    Else
        MsgBox "Alle Verweise in Ordnung."
    End If
End Sub
```

Damit brauchen Sie dann nicht erst die Verweise- beziehungsweise **References**-Auflistung zu durchlaufen. Das ist erst notwendig, wenn **BrokenReference** den Wert

True liefert. In diesem Fall prüfen wir mit der Eigenschaft **IsBroken** des **Reference**-Objekts, ob der aktuell durchlaufene Verweis kaputt ist und geben dann seinen Namen im Direktbereich des VBA-Editors aus.

Build

Gibt die Buildnummer der aktuellen Version von Microsoft Access zurück:

```
? Application.Build
 14228
```

BuildCriteria

Die **BuildCriteria**-Funktion erlaubt es, Vergleichsausdrücke für verschiedene Datentypen zusammenzustellen. Das ist insbesondere praktisch, wenn man Kriterien etwa für die DLookup-Funktion oder andere Domänenfunktionen zusammenstellen möchte. Die Funktion erwartet folgende Parameter:

- **Field**: Name des Feldes, nach dessen Inhalt gesucht werden soll
- **FieldType**: Felddatentyp, zum Beispiel **dbText**, **dbLong**, **dbCurrency** oder **dbDate**
- **Expression**: Vergleichsausdruck, zum Beispiel **André**, **1**, **Date()**

Hier sind einige Beispiele:

```
Debug.Print BuildCriteria("Vorname", dbText, "André")
Vorname="André"
Debug.Print BuildCriteria("Vorname", dbText, "A*")
Vorname Like "A*"
Debug.Print BuildCriteria("Menge", dbLong, "1")
Menge=1
Debug.Print BuildCriteria("Geburtsdatum", dbDate, Date)
Geburtsdatum=#10/30/2021#
Debug.Print BuildCriteria("Startzeit", dbDate, Now)
Startzeit=#10/30/2021 10:13:4#
Debug.Print BuildCriteria("Einzelpreis", dbCurrency,
"10,50")
Einzelpreis=10.5
Debug.Print BuildCriteria("Aktiv", dbBoolean, "Falsch")
Aktiv=False
```

Wir sehen hier, dass die Funktion recht flexibel ist. Beim Vergleich von Textfeldern ohne Platzhalter wie dem Sternchen (*) wird das Gleichheitszeichen als Operator verwendet, bei solchen mit Sternchen **LIKE**. Außerdem werden Zeichenketten automatisch in Anführungszeichen eingefasst. Datumsangaben werden in ein SQL-kompatibles Format umgewandelt und in Zahlen werden Kommata durch das aktuell verwendete Dezimaltrennzeichen ersetzt. Außerdem wandelt die Funktion verschiedene **Boolean**-Werte, zum Beispiel **Wahr**, direkt in eine verwertbare Form um (**True**).

Im bald erscheinenden Beitrag **Suchfunktion mit BuildCritera** (**www.access-im-unternehmen.de/1339**) zeigen wir einen Anwendungszweck für diese Funktion.

CloseCurrentDatabase

Schließt die Datenbank in der mit dem **Application**-Objekt angegebenen Access-Instanz. Sie können diesen Befehl in der aktuellen Access-Instanz wie folgt einsetzen:

```
Application.CloseCurrentDatabase
```

Wenn Sie eine Datenbank wie weiter unten unter dem Befehl **OpenCurrentDatabase** beschrieben in einer eigenen Instanz geöffnet haben, die beispielsweise **objAccess** heißt, verwenden Sie folgende Anweisung, um die Datenbank zu schließen:

```
objAccess.Application.CloseCurrentDatabase
```

CodeContextObject

CodeContextObject liefert den Namen des Objekts, in dem ein Code ausgeführt wird. Es funktioniert nur für Code in den Klassenmodulen von Formular- und Be-

richtsmodulen. Allerdings können Sie es auch in Routinen nutzen, die von Code in Formularen oder Berichten aus aufgerufen werden. So können Sie etwa in Fehlerbehandlungsroutinen abfragen, in welchem Objekt der Fehler ausgelöst wurde, ohne dass Sie einen Verweis auf das Objekt oder seinen Namen als Parameter an die Fehlerbehandlungsroutine übergeben müssen.

Dazu platzieren wir beispielsweise folgende Prozedur hinter der Ereigniseigenschaft einer Schaltfläche in einem Formular:

```
Private Sub cmdFehlerAusloesen_Click()
    On Error Resume Next
    Debug.Print 1 / 0
    If Not Err.Number = 0 Then
        Call ErrorHandler
    End If
End Sub
```

Die von dort aufgerufene Prozedur **ErrorHandler** legen wir in einem Standardmodul an:

```
Public Sub ErrorHandler()
    MsgBox "Fehler in '" _
            & Application.CodeContextObject.Name & "' (" _
           & TypeName(Application.CodeContextObject) & ")"
End Sub
```

Diese liefert dann den Namen des Formulars sowie den Typ (siehe Bild 4).

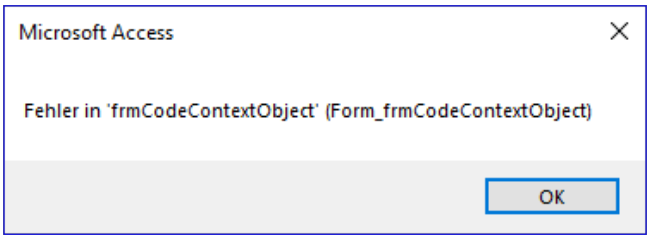


Bild 4: Meldung mit Informationen der Eigenschaft **CodeContextObject**

CodeData

Die Eigenschaft **CodeData** hat eine sehr ähnliche Funktion wie **CurrentData** (für die Beschreibung verweisen wir daher auf die Eigenschaft **CurrentData**, die Sie weiter unten finden). Die Eigenschaft ist für die Nutzung in Access-Add-Ins vorgesehen. Dort können Sie dann mit **CodeData** auf die Elemente der Add-In-Datenbank zugreifen und mit **CurrentData** auf die Elemente der aktuell geladenen Datenbank.

CodeDb

CodeDb entspricht im Prinzip der Eigenschaft **CurrentDb**, daher verweisen wir an dieser Stelle auf die Beschreibung der Eigenschaft **CurrentDb**, die wir weiter unten vorstellen. Genau wie bei **CodeData** und **CurrentData** ist **CodeDb** für den Einsatz in Access-Add-Ins vorgesehen, um auf das **Database**-Objekt der Add-In-Datenbank zuzugreifen. Mit **CurrentDb** hingegen greift man von der Add-In-Datenbank auf das **Database**-Objekt der aktuell in Access angezeigten Datenbank zu.

CodeProject

CodeProject entspricht im Prinzip der Eigenschaft **CurrentProject**, daher verweisen wir an dieser Stelle auf die Beschreibung der Eigenschaft **CurrentProject**, die wir weiter unten vorstellen. Genau wie bei **CodeData** und **CurrentData** ist **CodeProject** für den Einsatz in Access-Add-Ins vorgesehen, um auf das **CurrentProject**-Objekt der Add-In-Datenbank zuzugreifen. Mit **CurrentProject** hingegen greift man von der Add-In-Datenbank auf das **CurrentProject**-Objekt der aktuell in Access angezeigten Datenbank zu.

ColumnHistory

Mit dieser Funktion können Sie den Versionsverlauf der Daten in einem Memofeld abfragen. Dazu erstellen Sie ein Memofeld (Datentyp **Langer Text**) und stellen dessen Eigenschaft **Nur anfügen** auf **Ja** ein (siehe Bild 5). Dann können Sie mit **ColumnHistory** für einen bestimmten Datensatz den Versionsverlauf auslesen. Für unser Beispiel und den Datensatz mit dem Wert **1** im Feld **ID** sieht der Aufruf wie folgt aus:

```
Debug.Print Application.ColumnHisto-
ry("tblMemofelder", "Memofeld", "ID = 1")
```

Nachdem wir erst eine, dann eine zweite Zeile eingegeben haben, erhalten wir dieses Ergebnis:

```
[Version: 30.10.2021 11:27:22 ] Erste
Zeile.
[Version: 30.10.2021 11:27:43 ] Erste
Zeile.
Zweite Zeile.
```

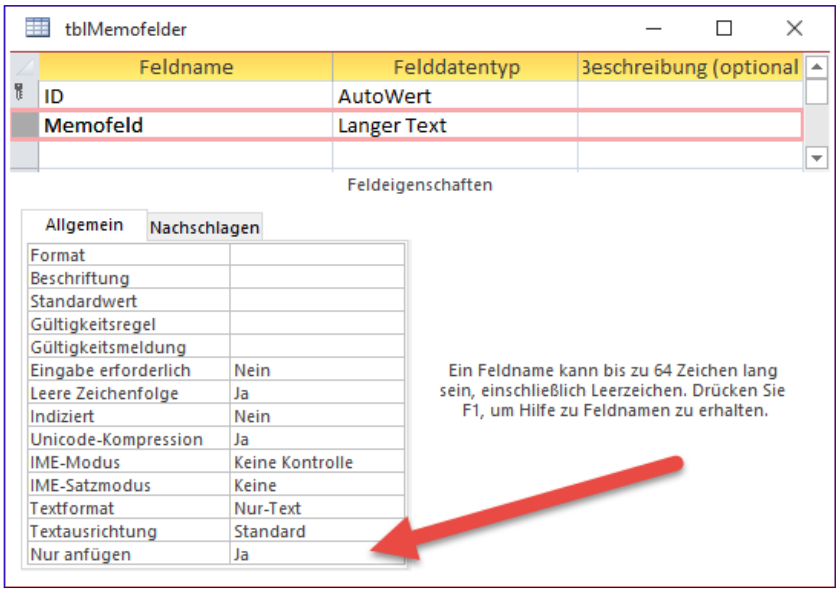


Bild 5: Memofeld zum Anfügen von Texten

COMAddIns

Die Auflistung **COMAddIns** liefert alle COM-Add-Ins, die für die aktuelle Access-Instanz vorliegen. COM-Add-Ins sind Add-Ins, die im Gegensatz zu Access-Add-Ins mit alternativen Entwicklungsumgebungen erstellt werden, zum Beispiel VB6, .NET oder twinBASIC. Die folgende **For Each**-Schleife gibt die Beschreibung und die **ProgID** aller aktuell installierten Elemente aus:

```
Public Sub Test_COMAddIns()
    Dim objCOMAddIn As COMAddIn
    For Each objCOMAddIn In Application.COMAddIns
        Debug.Print objCOMAddIn.Description, _
                                    objCOMAddIn.ProgId
    Next objCOMAddIn
End Sub
```

CommandBars

Mit der **CommandBars**-Auflistung greifen Sie auf alle **CommandBar**-Definitionen der aktuellen Instanz von Access zu. Das mag in Zeiten des Ribbons veraltet zu sein, aber wir wollen die guten, alten Kontextmenüs nicht vergessen: Auch diese lassen sich mit dieser Auflistung ermitteln.

Im folgenden Beispiel durchlaufen wir alle Elemente der **CommandBars**-Auflistung und geben all diejenigen im Direktbereich des VBA-Editors aus, deren Eigenschaft **Type** den Wert **msoBarTypePopup** aufweist:

```
Public Sub Test_CommandBars()
    Dim cbr As CommandBar
    Debug.Print "Anzahl CommandBars:" _
                          & Application.CommandBars.Count
    For Each cbr In Application.CommandBars
        If cbr.Type = msoBarTypePopup Then
            Debug.Print cbr.Name
        End If
    Next cbr
End Sub
```

CompactRepair

Diese Methode komprimiert und repariert die im ersten Parameter angegeben Datenbank und erzeugt eine komprimierte und reparierte Version unter dem mit dem zweiten Parameter angegebenen Dateinamen:

```
Application.CompactRepair _
                CurrentProject.Path & "\Beispiel.accdb", _
    CurrentProject.Path & "\Beispiel_Komprimiert.accdb", _
                                                True
```

Mit dem dritten Parameter geben Sie an, ob im Falle von bei der Reparatur entdeckten korrupten Elemente eine Log-Datei im Verzeichnis der Datenbank angelegt werden soll.

ConvertAccessProject

Diese Methode konvertiert eine ADP-Datei, also ein Access Data Project, in eine reine Access-Datenbank. Sie erwartet die folgenden Parameter:

- **SourceFilename**: Name der zu konvertierenden ADP-Datei
- **DestinationFilename**: Name der zu erstellenden Datei
- **DestinationFileFormat**: Format der zu erstellenden Datei (zum Beispiel **acFileFormatAccess2007**)

CreateAccessProject

Mit dieser Methode können Sie eine ADP-Datei, also ein Access Data Project, erzeugen. Sie ist allerdings in neueren Versionen von Access nicht mehr verfügbar und der Aufruf löst den Fehler **Das Format ADP (Access Data Project) wird ab dieser Version von Access nicht mehr unterstützt.** aus:

```
Application.CreateAccessProject _
                    CurrentProject.Path & "\Beispiel.adp"
```

Als zweiten Parameter können Sie noch die Verbindungszeichenfolge für die zu verbindende SQL Server-Datenbank angeben.

CreateAdditionalData

Diese Methode dient dazu, bei einem Export im XML-Format der Daten weitere Tabellen zu den Daten der zu exportierenden Tabelle hinzuzufügen.

Weitere Informationen zu dieser Methode finden Sie im Beitrag **XML-Export mit VBA** (**www.access-im-unternehmen.de/1046**).

CreateControl

Die Funktion **CreateControl** erstellt ein neues Steuerelement in einem Formular und liefert einen Verweis auf das neue Steuerelement als Funktionswert zurück. Diese und die Methoden **CreateForm** und **DeleteControl** erläutern wir im Artikel **Formulare per VBA erstellen** (**www.access-im-unternehmen.de/1332**).

CreateForm

Die Methode **CreateForm** erstellt ein neues Formular und liefert einen Verweis auf das neu erstellte Formular als Funktionswert zurück.

CreateGroupLevel

Diese Methode erzeugt in einem Bericht einen neuen Gruppierungs- oder Sortierungsbereich.

Diese und die Methoden **CreateReport**, **CreateReportControl** und **DeleteReportControl** erläutern wir im Artikel **Berichte per VBA erstellen** (**www.access-im-unternehmen.de/1338**).

CreateReport

Diese Funktion erstellt einen neuen Bericht und liefert einen Verweis auf den neu erstellten Bericht als Funktionswert zurück.

CreateReportControl

Diese Funktion erstellt ein Steuerelement in einem Bericht und liefert einen Verweis auf das Steuerelement als Funktionswert zurück.

CurrentData

Das mit der Eigenschaft **CurrentData** gelieferte Objekt liefert Eigenschaften mit verschiedenen Auflistungen:

- **AllQueries**: Liefert eine Auflistung aller Abfragen der aktuellen Datenbank.
- **AllTables**: Liefert eine Auflistung aller Tabellen der aktuellen Datenbank.

- **AllDatabaseDiagrams**, **AllFunctions**, **AllStoresProcedures**, **AllViews**: Liefert Auflistung der SQL Server-Objekte (nur in ADP-Dateien verfügbar, die nicht mehr unterstützt werden)

Der folgende Code gibt die Anzahl der Abfragen und Tabellen aus und anschließend in einer **For Each**-Schleife die Namen aller Einträge der Auflistung **AllTables**:

```
Public Sub Test_CurrentData()
    Dim objObject As AccessObject
    Debug.Print Application.CurrentData.AllQueries.Count
    Debug.Print Application.CurrentData.AllTables.Count
    For Each objObject In Application.CurrentData.AllTables
        Debug.Print objObject.FullName
    Next objObject
End Sub
```

Achtung: Die beiden Eigenschaften **CodeData** und **CurrentData** liefern scheinbar die gleichen Ergebnisse. Das ist nur der Fall, wenn Sie in der aktuell geladenen Datenbank verwendet werden. Wenn Sie die beiden Eigenschaften in einer Add-In-Datenbank nutzen, die ja als Add-In für die Verwendung zusätzlich zu einer anderen Access-Datenbank genutzt wird, dann haben diese einen anderen Kontext:

- **CodeData**: Liefert dann die Auflistung der Tabellen und Abfragen, die in der Add-In-Datenbank gespeichert sind.

- **CurrentData**: Liefert dann die Auflistung der Tabellen und Abfragen, die in der in Access geöffneten Datenbank gespeichert sind.

CurrentDb

Die Funktion **CurrentDb** liefert einen Verweis auf das **Database**-Objekt der aktuellen Datenbank zurück. Diese Funktion und das **Database**-Objekt beschreiben wir ausführlich im bald erscheinenden Beitrag **Die CurrentDb-Funktion und das Database-Objekt** (**www.access-im-unternehmen.de/1111**).

CurrentObjectName

Die Eigenschaft **CurrentObjectName** liefert den Namen des aktuell im Navigationsbereich markierten Elements. Wenn mehrere Elemente markiert sind, liefert die Eigenschaft den Namen des zuerst markierten Elements.

CurrentObjectType

Diese Eigenschaft liefert, wenn ein Objekt im Datenbankfenster angezeigt wird, den Objekttyp des aktuell aktiven Objekts im Datenbankfenster. Ist kein Objekt im Datenbankfenster sichtbar, liefert die Eigenschaft den Objekttyp des aktuell im Navigationsbereich markierten Elements. Wenn mehrere Elemente markiert sind, liefert die Eigenschaft den Objekttyp des zuerst markierten Elements.

Es gibt die folgenden Objekttypen:

- **0**: Tabelle (**acTable**)

- **1**: Abfrage (**acQuery**)

- **2**: Formular (**acForm**)

- **3**: Bericht (**acReport**)

- 4: Makro (**acMacro**)

- **5**: Modul (**acModule**)

CurrentProject

Mit der **CurrentProject**-Eigenschaft holen Sie einen Verweis auf das Objekt des gleichnamigen Typs. Es hält einige Methoden und Eigenschaften bereit, die wir uns im Detail im bald erscheinenden Beitrag **Die CurrentProject-Klasse** (**www.access-im-unternehmen.de/1340**) ansehen.

CurrentUser

Gibt den Namen des aktuellen Benutzers der Access-Datenbank zurück, in der Regel **Admin**. Diese Eigenschaft war in älteren Versionen wichtig, als Access noch das Sicherheitssystem mit der Benutzerverwaltung unterstützt hat.

DAvg

Diese Funktion ermittelt einen Durchschnittswert. Sie finden eine detaillierte Beschreibung im Beitrag **Domänenfunktionen** (**www.access-im-unternehmen.de/317**)

DBEngine

Die **DBEngine**-Eigenschaft liefert einen Verweis auf das gleichnamige Objekt. Dieses stellt Funktionen bereit, um Transaktionen durchzuführen, Datenbanken zu erstellen oder Fehlerauflistungen abzufragen. Der bald erscheinenden Beitrag **Das DBEngine-Objekt** (**www.access-im-unternehmen.de/1341**) beschreibt das Objekt im Detail.

DCount

Diese Funktion ermittelt die Anzahl der Datensätze mit den angegebenen Kriterien. Sie finden eine detaillierte Beschreibung im Beitrag **Domänenfunktionen** (**www.access-im-unternehmen.de/317**)

DeleteControl

Diese Methode löscht ein Steuerelement von einem Formular. Weitere Informationen weiter oben unter **CreateControl**.

DeleteReportControl

Diese Methode löscht ein Steuerelement von einem Bericht. Für weitere Informationen siehe weiter oben unter **CreateGroupLevel**.

DFirst

Diese Funktion ermittelt den Wert des angegebenen Feldes des ersten gefundenen Datensatzes mit den angegebenen Kriterien. Sie finden eine detaillierte Beschreibung im Beitrag **Domänenfunktionen** (**www.access-im-unternehmen.de/317**).

DLast

Diese Funktion ermittelt den Wert des angegebenen Feldes des letzten gefundenen Datensatzes mit den angegebenen Kriterien. Sie finden eine detaillierte Beschreibung im Beitrag **Domänenfunktionen** (**www.access-im-unternehmen.de/317**).

DLookup

Diese Funktion ermittelt den Wert des angegebenen Feldes des ersten gefundenen Datensatzes mit den angegebenen Kriterien. Sie finden eine detaillierte Beschreibung im Beitrag **Domänenfunktionen** (**www.access-im-unternehmen.de/317**).

DMax

Diese Funktion ermittelt den maximalen Wert des angegebenen Feldes der gefundenen Datensätze mit den angegebenen Kriterien. Sie finden eine detaillierte Beschreibung im Beitrag **Domänenfunktionen** (**www.access-im-unternehmen.de/317**).

DMin

Diese Funktion ermittelt den minimalen Wert des angegebenen Feldes der gefundenen Datensätze mit den angegebenen Kriterien. Sie finden eine detaillierte Beschreibung im Beitrag **Domänenfunktionen** (**www.access-im-unternehmen.de/317**).

DoCmd

Das **DoCmd**-Objekt stellt einige Methoden bereit, die üblicherweise über die Benutzeroberfläche oder als Makrobefehl bereitgestellt werden. Wir stellen diese Methoden ausführlich im bald erscheinenden Beitrag **Das DoCmd-Objekt** vor (**www.access-im-unternehmen.de/1342**).

DStDev

Diese Funktion ermittelt die Standardabweichung des angegebenen Feldes der gefundenen Datensätze mit den angegebenen Kriterien bezogen auf eine Stichprobe. Sie finden eine detaillierte Beschreibung im Beitrag **Domänenfunktionen** (**www.access-im-unternehmen.de/317**).

DStDevP

Diese Funktion ermittelt den maximalen Wert des angegebenen Feldes der gefundenen Datensätze mit den angegebenen Kriterien bezogen auf die Gesamtmenge. Sie finden eine detaillierte Beschreibung im Beitrag **Domänenfunktionen** (**www.access-im-unternehmen.de/317**).

Assistent für m:n-Beziehungen

Microsoft Access bietet eine ganze Reihe von praktischen Assistenten. Ich setze beispielsweise sehr oft den Nachschlage-Assistenten ein, der nicht nur eine Beziehung mit den gewünschten Optionen anlegt, sondern das bearbeitete Feld auch noch als Kombinationsfeld auslegt, mit dem die Daten der verknüpften Tabelle leicht ausgewählt werden können. Eine m:n-Beziehung stellen Sie her, indem Sie zwei solcher Nachschlagefelder in der sogenannten Verknüpfungstabelle anlegen. Noch praktischer wäre es, wenn Sie diese Verknüpfungstabelle gar nicht erst anlegen müssten. Stattdessen wären nur die zu verknüpfenden Tabellen auszuwählen und der Assistent erledigt den Rest – das Anlegen der Verknüpfungstabelle mit den notwendigen Feldern sowie das Einrichten der Beziehungen zu den zu verknüpfenden Tabellen. Dieser Beitrag zeigt, wie Sie einen solchen Assistenten programmieren können.

Ausgangsposition

Der Assistent soll zwei Tabellen wie die in Bild 1 miteinander verknüpfen. Dazu benötigen wir eine weitere Tabelle etwa namens **tblProdukteKategorien** mit dem Primärschlüsselfeld **ProduktKategorieID** und den beiden Fremdschlüsselfeldern **ProduktID** und **KategorieID**.

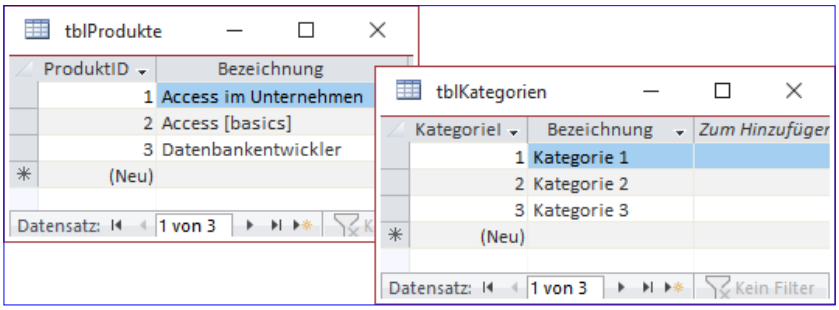


Bild 1: Zu verknüpfende Tabellen

Diese stellen jeweils die Verknüpfung zu den Tabellen **tblProdukte** und **tblKategorien** her, sodass alle Einträge der Tabelle **tblProdukte** mit den Einträgen der Tabelle **tblKategorien** verknüpft werden können. Die Fremdschlüsselfelder legen Sie am schnellsten an, indem Sie im Tabellenentwurf den Eintrag **Nachschlage-Assistent...** wählen und die Verknüpfung darüber vornehmen. Die Verknüpfungstabelle sieht danach wie in Bild 2 aus.

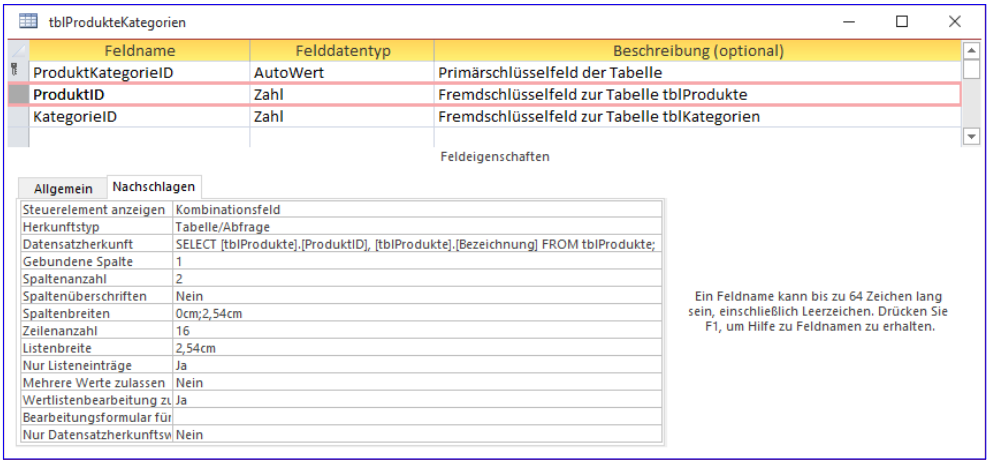


Bild 2: Die Verknüpfungstabelle

Wechseln Sie in die Datenblattansicht, erhalten Sie eine Tabelle, mit deren Fremdschlüsselfeldern Sie

die gewünschten Kombinationen aus Produkten und Kategorien leicht zusammenstellen können (siehe Bild 3).

Die Beziehungen können Sie danach wie in Bild 4 im **Beziehungen**-Fenster einsehen.

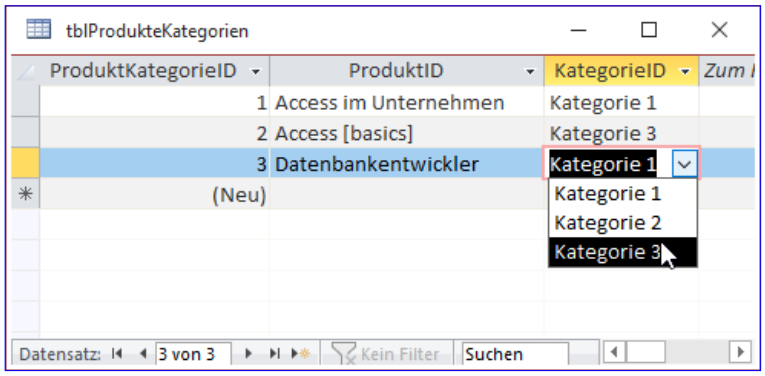


Bild 3: Datenblattansicht der Verknüpfung

Vorher

Um eine m:n-Beziehung zwischen zwei Tabellen herzustellen, sind normalerweise die folgenden Schritte nötig:

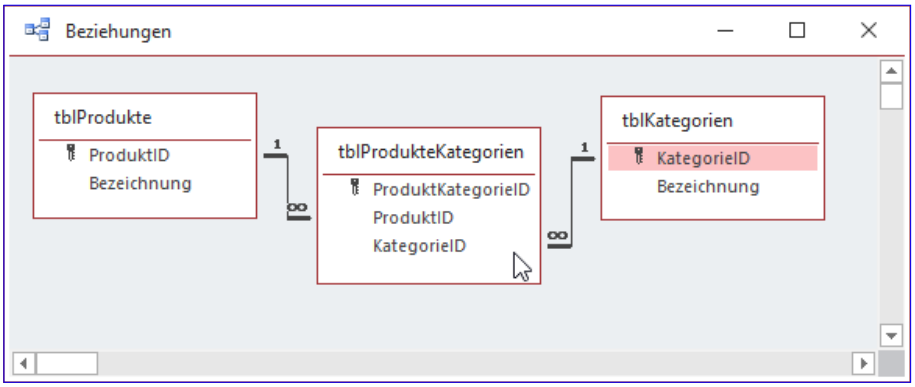


Bild 4: Die m:n-Beziehung im Beziehungen-Fenster

- Erstellen der Verknüpfungstabelle
- Wahl eines Namens für die Verknüpfungstabelle
- Hinzufügen eines Primärschlüsselfeldes
- Hinzufügen eines Fremdschlüsselfeldes zum Verknüpfen der Verknüpfungstabelle mit der m-Tabelle
- Hinzufügen eines Fremdschlüsselfeldes zum Verknüpfen der Verknüpfungstabelle mit der n-Tabelle
- Aufrufen des Nachschlage-Assistenten zum Erstellen der Beziehung zwischen dem ersten Fremdschlüsselfeld und dem Primärschlüsselfeld der m-Tabelle oder, falls kein Nachschlagefeld gewünscht ist, sondern nur die reine Beziehung, manuelles Hinzufügen der Beziehung über das Beziehungen-Fenster. Außerdem Einstellen der Beziehungseigenschaften wie referenzielle Integrität, Löschweitergabe und Aktualisierungsweitergabe.
- Aufrufen des Nachschlage-Assistenten zum Erstellen der Beziehung zwischen dem zweiten Fremdschlüsselfeld und dem Primärschlüsselfeld der n-Tabelle oder, falls kein Nachschlagefeld gewünscht ist, sondern nur die reine Beziehung, manuelles Hinzufügen der Beziehung über das **Beziehungen**-Fenster. Außerdem Einstellen der Beziehungseigenschaften wie referenzielle Integrität, Löschweitergabe und Aktualisierungsweitergabe.
- Gegebenenfalls Hinzufügen weiterer Felder zur m:n-Verknüpfungstabelle wie etwa Einzelpreis oder Menge bei einer Tabelle zum Speichern von Bestellpositionen

Anforderungen

Was genau benötigen wir an Informationen, um automatisiert eine Verknüpfungstabelle für zwei per m:n-Beziehung zu verknüpfenden Tabellen zu erstellen? Beziehungsweise welche Informationen muss der Assistent zur Erstellung einer m:n-Beziehung vom Entwickler abfragen?

- Name der ersten Tabelle

- Name des Primärschlüsselfeldes der ersten Tabelle
- Gegebenenfalls Name des anzuzeigenden Feldes der ersten Tabelle
- Name der zweiten Tabelle
- Name des Primärschlüsselfeldes der zweiten Tabelle
- Gegebenenfalls Name des anzuzeigenden Feldes der zweiten Tabelle
- Name der zu erstellenden Tabelle
- Namen für die Fremdschlüsselfelder zum Herstellen der Beziehung mit der m-Tabelle und der n-Tabelle
- Gegebenenfalls weitere Felder, die zur m:n-Tabelle hinzugefügt werden sollen

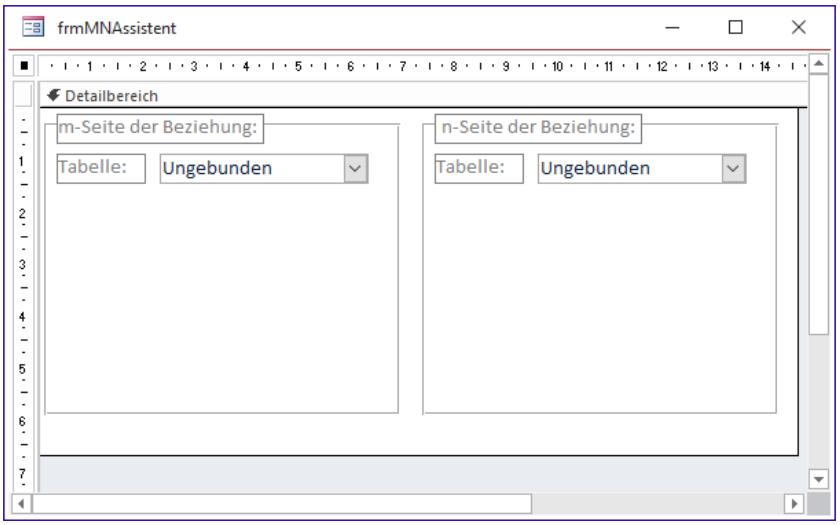


Bild 5: Assistenten-Formular

Programmieren des Add-Ins

Starten wir also direkt mit der Programmierung des Add-Ins. Dieses soll über ein Formular verfügen, mit dem wir alle Einstellungen für die Erstellung der m:n-Beziehung vornehmen können.

Auswahl der beteiligten Tabellen

Als Erstes benötigen wir zwei Kombinationsfelder, mit denen wir die beiden an der Beziehung beteiligten Tabellen ermitteln können.

Diese Kombinationsfelder erhalten nach dem Hinzufügen zu einem neuen Formular namens **frmMNAssistent** die Bezeichnungen **cboMTabelle** und **cboNTabelle**. Diese fügen wir jeweils in einen neuen Rahmen ein, sodass der Aufbau in der Entwurfsansicht zunächst wie in Bild 5 aussieht.

Damit der Benutzer die in der aktuellen Datenbank enthaltenen Tabellen auswählen kann, stellen wir die Eigenschaft **Datensatzherkunft** der beiden Kombinationsfelder auf die Abfrage aus Bild 6 ein.

Damit die hier verwendete Tabelle **MSysObjects** sichtbar ist, müssen Sie zunächst die entsprechende Option aktivieren. Diese finden Sie, wenn Sie mit der rechten Maustaste auf den Navigationsbereich klicken und dann den Eintrag **Navigationsoptionen...** auswählen. Im nun erscheinenden Dialog **Navigationsoptionen** aktivieren Sie die Option **Systemobjekte anzeigen**.

Danach können Sie der Abfrage für die Eigenschaft **Datensatzherkunft** wie in der Abbildung die Tabelle **MSysObjects** hinzufügen und die beiden Felder **Name** und **Type** zum Entwurfsraster der Abfrage hinzufügen. Außerdem stellen Sie als Kriterium für das Feld **Name** den Wert **Nicht Wie "MSys*" Und Nicht Wie "USys*" Und Nicht Wie "f_*"** und für das Feld **Type** den Wert **1** ein. Das sorgt erstens dafür, dass keine System- und temporären Tabellen erscheinen und dass nur lokale Tabellen zur Auswahl angeboten werden.

Nach dem Schließen des Abfrageentwurfs können Sie den Wert der Eigenschaft **Datensatzherkunft** des Kombinationsfeldes **cboMTabelle** in die entsprechende Eigenschaft des Kombinationsfeldes **cboNTabelle** kopieren:

```
SELECT Name, Type
FROM MSysObjects
WHERE (Name Not Like "MSys*"
And Name Not Like "USys*"
And Name Not Like "f_*")
AND (Type=1);
```

Wenn Sie anschließend in die Formularansicht wechseln, können Sie bereits die an der Beziehung beteiligten Tabellen auswählen (siehe Bild 7).

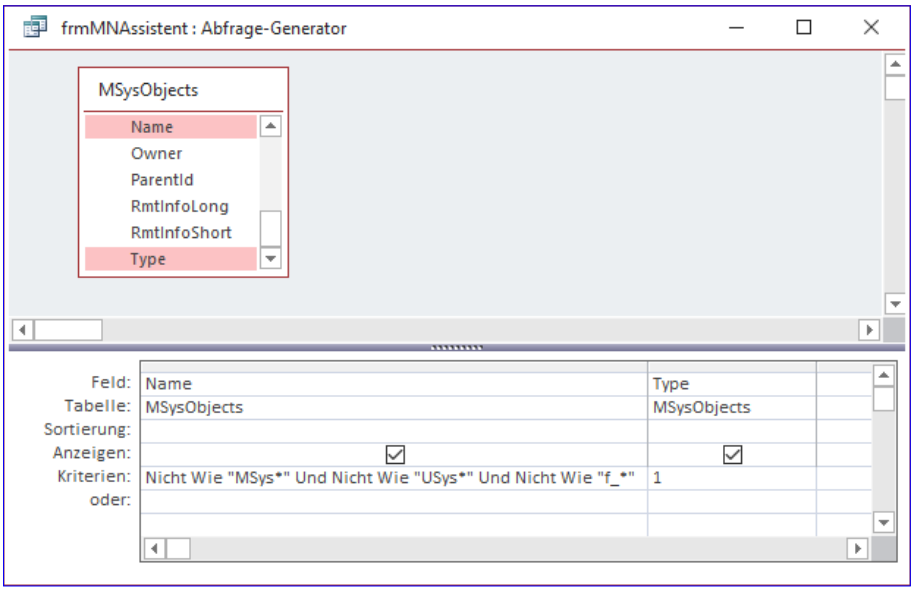


Bild 6: Datensatzherkunft der Kombinationsfelder zur Auswahl der an der Beziehung beteiligten Tabellen

Auswahl der beteiligten Primärschlüsselfelder

Unter den beiden Kombinationsfeldern platzieren wir zwei weitere Kombinationsfelder namens **cboMPrimaerschluessel** und **cboNPrimaerschluessel**.

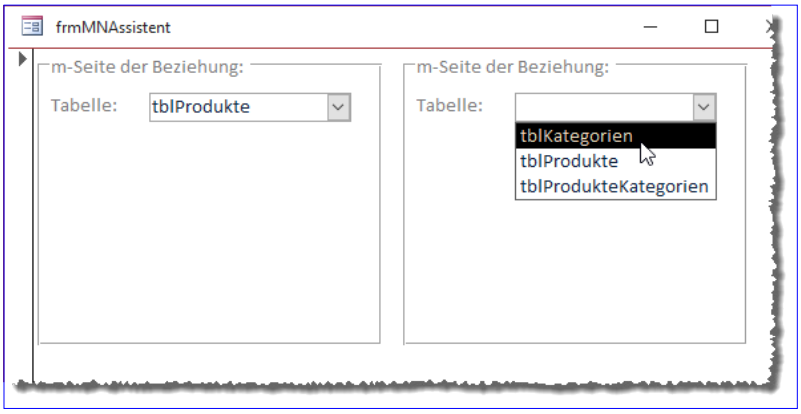


Bild 7: Auswahl der an der Beziehung beteiligten Kombinationsfelder

Diese sollen direkt nach der Auswahl einer Tabelle mit dem darüber befindlichen Kombinationsfeld alle Felder der gewählten Tabelle anzeigen und das Primärschlüsselfeld dieser Tabelle als Vorauswahl einstellen.

Dazu müssen wir zwei Aufgaben erledigen:

- eine Liste aller Felder der jeweiligen Tabelle zusammenstellen und
- das Primärschlüsselfeld aus diesen Feldern ermitteln.

Die Liste der Felder können wir auf zwei Wegen herausfinden. Der erste verwendet ausschließlich VBA und nutzt die **Fields**-Auflistung des jeweiligen **TableDef**-Objekts.

Der zweite verwendet die Herkunftsart **Feldliste** und die im ersten Kombinationsfeld ausgewählte Tabelle als **Datensatzherkunft**. Dabei müssen wir aber dennoch nach der Auswahl VBA bemühen, um die Eigenschaft **Datensatzherkunft** zu füllen. Um nur das zu erledigen, hinterlegen wir die folgenden beiden Ereignisprozeduren jeweils für Kombinationsfelder **cboMTabelle** und **cboNTabelle**:

```
Private Sub cboMTabelle_AfterUpdate()
    Me!cboMPrimaerschluessel.RowSource = Me!cboMTabelle
End Sub

Private Sub cboNTabelle_AfterUpdate()
    Me!cboNPrimaerschluessel.RowSource = Me!cboNTabelle
End Sub
```

Damit bieten die beiden Kombinationsfelder bereits die Felder der jeweiligen Tabelle zur Auswahl an (siehe Bild 8).

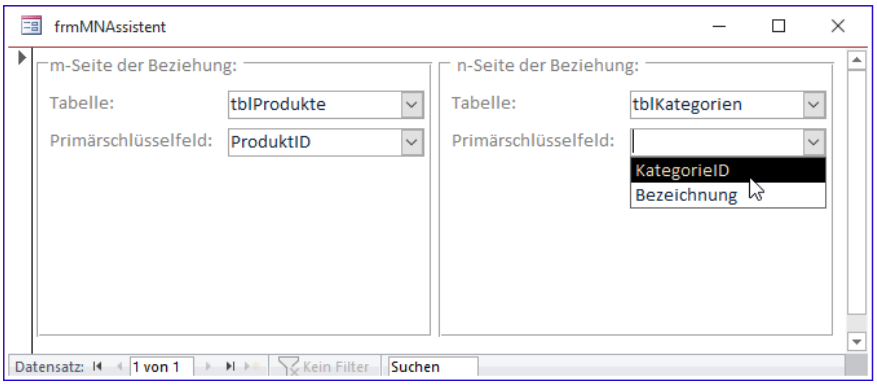


Bild 8: Auswahl des Primärschlüsselfeldes für das Erstellen der m:n-Beziehung

Primärschlüsselfeld auswählen

Nun wollen wir noch dafür sorgen, dass direkt die Primärschlüsselfelder dieser Tabellen angezeigt werden. Schließlich soll der Benutzer so wenig Aufwand wie möglich haben. Dazu fügen wir den beiden Ereignisprozeduren **cboMTabelle_AfterUpdate** und **cboNTabelle_AfterUpdate** jeweils einen Aufruf einer Funktion namens **PrimaerschluesselErmitteln** hinzu:

```
Private Sub cboMTabelle_AfterUpdate()
    Me!cboMPrimaerschluessel.RowSource = Me!cboMTabelle
    Me!cboMPrimaerschluessel = _
            PrimaerschluesselErmitteln(Me!cboMTabelle)
End Sub
```

Diese Funktion erwartet den Namen der zu untersuchenden Tabelle als Parameter. Sie füllt die Variable **db** mit einem Verweis auf das aktuelle **Database**-Objekt und **tdf** mit einem Verweis auf das **TableDef**-Objekt für die Tabelle aus **strTabelle**. Dann durchläuft sie alle Einträge der Auflistung **tdf.Indexes** und speichert das jeweils aktuelle Element in der Variablen **idx**. Für dieses prüft sie den Wert der Eigenschaft **Primary**.

Ist dieser **True**, schreibt sie den Namen des ersten Feldes dieses Indizes in den Rückgabewert der Funktion und beendet diese mit **Exit Function**. Das funktioniert für keinen, einen oder zusammengesetzte Primärschlüssel gleichermaßen sinnvoll: Wenn kein Primärschlüssel für die Tabelle vorhanden ist, liefert die Funktion eine leere Zeichenkette zurück, für einen Primärschlüssel mit einem Feld den Namen des betroffenen Feldes und für einen zusammengesetzten Primärschüssel den Namen des ersten Feldes:

```
Private Function PrimaerschluesselErmitteln(_
                        strTabelle As String) As String
    Dim db As DAO.Database
    Dim tdf As DAO.TableDef
    Dim idx As DAO.Index
    Set db = CurrentDb
    Set tdf = db.TableDefs(strTabelle)
    For Each idx In tdf.Indexes
        If idx.Primary Then
            PrimaerschluesselErmitteln = _
                                idx.Fields(0).Name
            Exit Function
        End If
    Next idx
End Function
```

Für unser Beispiel mit den Produkten und Kategorien stellen die Prozeduren zuverlässig das jeweilige Primärschüsselfeld ein.

Informationen für das Nachschlagefeld

Zusätzlich wollen wir in diesem Formular die Daten für die Nachschlagefelder abfragen, sofern der Benutzer diese anlegen möchte. Manchmal ist es sinnvoll, zum Beispiel bei Bestellpositionen, wo man direkt aus der m:n-Verknüpfungstabelle das Produkt zu einer Bestellposition auswählen möchte. An anderen Stellen sind die Werte der Verknüpfungstabelle vielleicht gar nicht sichtbar, zum Beispiel wenn diese über zwei Listenfelder angezeigt werden.

Also fügen wir die notwendigsten Felder hinzu plus einem Kontrollkästchen zum Aktivieren oder Deaktivieren dieser Einstellungen.

Die Kontrollkästchen heißen **chkMNachschlagefeld** und **chkNNachschlagefeld**, die Kombinationsfelder zur Auswahl des anzuzeigenden Feldes für das Nachschlagefeld heißen **cboMNachschlagefeld** und **cboNNachschlagefeld** (siehe Bild 9). Damit die beiden Kombinationsfelder genau wie die zur Auswahl der Primärschlüsselfelder auch die Felder der zu verknüpfenden Tabellen anzeigen, fügen wir den Prozeduren **cboMTabelle_AfterUpdate** und **cboNTabelle_AfterUpdate** noch jeweils eine Anweisung hinzu:

```
Private Sub cboMTabelle_AfterUpdate()
    ...
    Me!cboMNachschlagefeld.RowSource = Me!cboMTabelle
End Sub

Private Sub cboNTabelle_AfterUpdate()
    ...
    Me!cboNNachschlagefeld.RowSource = Me!cboNTabelle
End Sub
```

Damit die beiden Kontrollkästchen beim Aktivieren und Deaktivieren auch die beiden Kombinationsfelder **cboMNachschlagefeld** und **cboNNachschlagefeld** aktivieren beziehungsweise deaktivieren, fügen wir diese Ereignisprozeduren für das Ereignis **Nach Aktualisierung** der beiden Kontrollkästchen hinzu:

```
Private Sub chkMNachschlagefeld_AfterUpdate()
    Me!cboMNachschlagefeld.Enabled = Me!chkMNachschlagefeld
End Sub

Private Sub chkNNachschlagefeld_AfterUpdate()
    Me!cboNNachschlagefeld.Enabled = Me!chkNNachschlagefeld
End Sub
```

Name für die zu erstellende Tabelle und ihre Felder ermitteln

Die Bezeichnungen für die zu erstellende Tabelle und die enthaltenen Felder bestimmen wir automatisch auf Basis der gewählten zu verknüpfenden Tabellen und den gewählten Primärschlüsselfeldern. Dies erledigen wir in der Prozedur **VerknuepfungstabelleAktualisieren** aus Listing 1. Sie liest zuerst die gewählten Tabellen in die Variablen **strMTabelle** und **strNTabelle** ein sowie die selektierten Primärschlüsselfelder.

Dann folgt die Prüfung, ob der Benutzer eines der Präfixe **tbl** oder **tbl_** für die beiden zu verknüpfenden Tabellen verwendet. Falls ja, werden diese vorn abgeschnitten, sodass beispielsweise von **tblProdukte** oder **tbl_Produkte** nur noch **Produkte** übrig bleibt.

Daraus leitet die Prozedur den Namen der Verknüpfungstabelle ab, der aus dem Präfix **tbl**, dem Namen der m-Tabelle und dem Namen der n-Tabelle jeweils ohne Präfix besteht.

Das Ergebnis landet im Textfeld **txtVerknuepfungstabelle**.

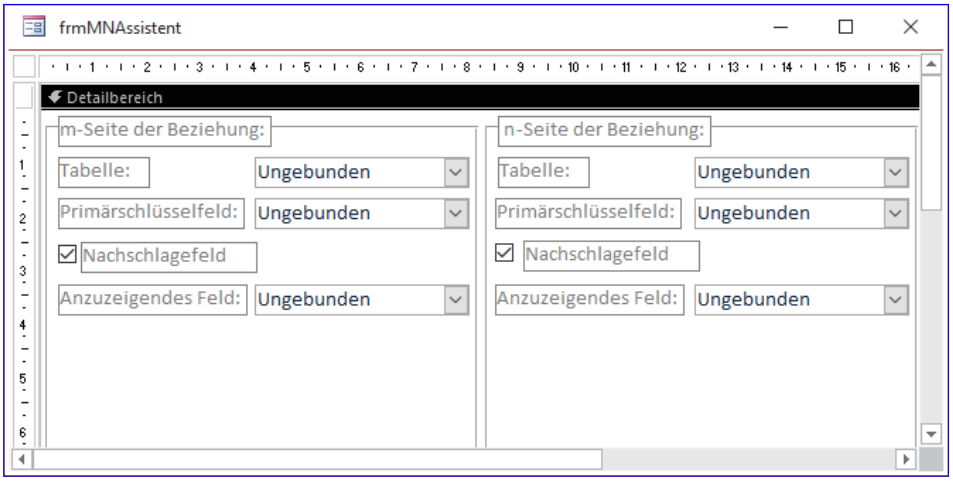


Bild 9: Informationen für die Nachschlagefelder

```
Private Sub VerknuepfungstabelleAktualisieren()
    Dim strMTabelle As String
    Dim strNTabelle As String
    Dim strMPrimaerschluessel As String
    Dim strNPrimaerschluessel As String
    strMTabelle = Nz(Me!cboMTabelle, "")
    strNTabelle = Nz(Me!cboNTabelle, "")
    strMPrimaerschluessel = Nz(Me!cboMPrimaerschluessel, "")
    strNPrimaerschluessel = Nz(Me!cboNPrimaerschluessel, "")
    If Left(strMTabelle, 4) = "tbl_" Then
        strMTabelle = Mid(strMTabelle, 5)
    End If
    If Left(strMTabelle, 3) = "tbl" Then
        strMTabelle = Mid(strMTabelle, 4)
    End If
    If Left(strNTabelle, 4) = "tbl_" Then
        strNTabelle = Mid(strNTabelle, 5)
    End If
    If Left(strNTabelle, 3) = "tbl" Then
        strNTabelle = Mid(strNTabelle, 4)
    End If
    Me!txtVerknuepfungstabelle = "tbl" & strMTabelle & strNTabelle
    If strMPrimaerschluessel = "ID" Then
        strMPrimaerschluessel = strMTabelle & "ID"
    ElseIf Right(strMPrimaerschluessel, 2) = "ID" Then
        strMPrimaerschluessel = Left(strMPrimaerschluessel, Len(strMPrimaerschluessel) - 2)
    End If
    If strNPrimaerschluessel = "ID" Then
        strNPrimaerschluessel = strNTabelle & "ID"
    ElseIf Right(strNPrimaerschluessel, 2) = "ID" Then
        strNPrimaerschluessel = Left(strNPrimaerschluessel, Len(strNPrimaerschluessel) - 2)
     End If
    Me!txtMFremdschluesselfeld = strMPrimaerschluessel & "ID"
    Me!txtNFremdschluesselfeld = strNPrimaerschluessel & "ID"
    Me!txtPrimaerschluesselfeld = strMPrimaerschluessel & strNPrimaerschluessel & "ID"
End Sub
```

Listing 1: Einstellen der Daten der Verknüpfungstabelle

Dann untersucht die Prozedur die gewählten Primärschlüsselfelder der beiden Tabellen. Diese dienen schließlich als Grundlage für die Benennung der Fremdschlüsselfelder der Verknüpfungstabelle.

Wenn der Name eines der Primärschlüsselfelder lediglich **ID** lautet, stellt die Prozedur den oben ermittelten Namen der Tabelle ohne Präfix voran. Das kann in unserem Fall auch der Plural sein, was zu Bezeichnungen wie **ProdukteID** oder **KategorienID** führen kann. Das ist aber kein Problem, denn der Benutzer kann die Bezeichnungen ja noch anpassen.

Sollte die Bezeichnung des Primärschlüsselfeldes jedoch der von uns erwarteten Konvention entsprechen, also aus dem Singular der Bezeichnung der enthaltenen

Objekte bestehen (wie **ProduktID** oder **KategorieID**), dann speichert die Prozedur den Teil vor **ID** in den Variablen **strMPrimaerschluessel** und **strNPrimaerschluessel**. Anschließend hängen wir hinten wieder **ID** an und schreiben das Ergebnis in die Textfelder **txtMFremdschluesselfeld** und **txtNFremdschluesselfeld**. Warum haben wir ID erst abgetrennt? Weil wir noch das Primärschlüsselfeld für die Verknüpfungstabelle benennen müssen, dass den Singular der Bezeichnungen der Elemente der ersten und der zweiten Tabelle und das Suffix **ID** enthalten soll, zum Beispiel **ProduktKategorieID**. Dieser Ausdruck landet schließlich im Textfeld **txtPrimaerschluesselfeld**.

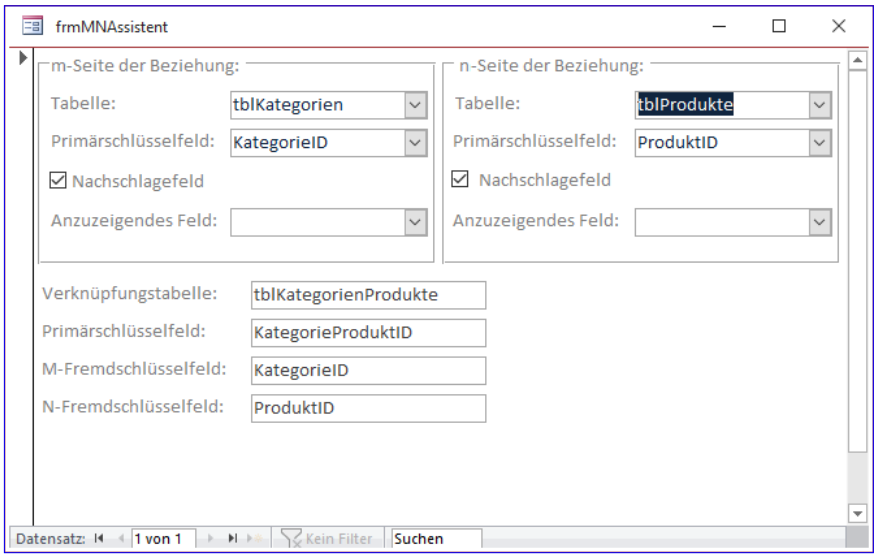


Bild 10: Das Formular **frmMNAssistent** in der Formularansicht

Das Ergebnis sieht schließlich wie in Bild 10 aus. Damit dies wie gewünscht funktioniert, fügen wir den Aufruf dieser Prozedur in die folgenden beiden Prozeduren nachträglich ein:

```
Private Sub cboMTabelle_AfterUpdate()
    ...
    VerknuepfungstabelleAktualisieren
End Sub

Private Sub cboNTabelle_AfterUpdate()
    ...
    VerknuepfungstabelleAktualisieren
End Sub
```

Für die Aktualisierung der beiden Kombinationsfelder **cboMPrimaerschluessel** und **cboNPrimaerschluessel** legen wir die folgenden Ereignisprozeduren an:

```
Private Sub cboNPrimaerschluessel_AfterUpdate()
    VerknuepfungstabelleAktualisieren
End Sub

Private Sub cboMPrimaerschluessel_AfterUpdate()
    VerknuepfungstabelleAktualisieren
End Sub
```

Prüfen, ob Verknüpfungstabelle bereits vorhanden ist

Bevor wir gleich die Verknüpfungstabelle mit den angegebenen Feldern und Indizes anlegen, prüfen wir, ob es bereits eine Tabelle mit dem angegebenen Namen gibt.

Dazu verwenden wir die folgende Funktion namens **ExistsTable**, die als Parameter den Namen der zu untersuchenden Tabelle entgegennimmt:

```
Public Function ExistsTable(strTable As String) As Boolean
    Dim db As dao.Database
    Dim tdf As dao.TableDef
    Set db = CurrentDb
    On Error Resume Next
    Set tdf = db.TableDefs(strTable)
    On Error GoTo 0
    If Not tdf Is Nothing Then
        ExistsTable = True
    End If
End Function
```

Die Funktion erstellt ein **Database**-Objekt und versucht dann, bei deaktivierter eingebauter Fehlerbehandlung auf das **TableDef**-Objekt mit dem Namen der zu untersuchenden Tabelle zuzugreifen. Gelingt das, ist **tdf** anschließend nicht leer, was dazu führt, dass der Funktionswert in der **If... Then**-Bedingung auf den Wert **True** eingestellt wird.

Vorhandene Tabelle löschen

Ist die Tabelle bereits vorhanden, fragen wir den Benutzer per Meldungsfenster, ob die vorhandene Tabelle gelöscht werden soll. Ist das der Fall, initiieren wir den Löschvorgang mit der folgenden Funktion. Diese ist auch für den Fall ausgestattet, dass die Tabelle aktuell geöffnet ist (siehe Listing 2).

```
Public Function DeleteTable(strTable As String) As Boolean
    Dim db As DAO.Database
    Dim lngError As Long
    Dim strError As String
    Set db = CurrentDb
    DoCmd.Close acTable, strTable, acSaveNo
    On Error Resume Next
    db.TableDefs.Delete strTable
    lngError = Err.Number
    strError = Err.Description
    On Error GoTo 0
    If Not lngError = 0 Then
        Select Case lngError
            Case Else
                MsgBox lngError & " " & strError
        End Select
    Else
        Application.RefreshDatabaseWindow
        DeleteTable = True
    End If
End Function
```

Listing 2: Löschen der angegebenen Tabelle

Die Funktion nimmt den Namen der Tabelle mit dem Parameter **strTable** entgegen. Als erste Aktion schließt sie die gegebenenfalls noch geöffnete Tabelle mit der Methode **DoCmd.Close**.

Für den dritten Parameter **Save** geben wir den Wert **acSaveNo** an, der dafür sorgt, dass im Falle offener Änderungen weder ein Speichern der Änderungen noch eine entsprechende Rückfrage erfolgt.

Dann deaktiviert die Funktion die Fehlerbehandlung und versucht mit der **Delete**-Methode der **TableDefs**-Auflistung, die Tabelle zu löschen. Dies kann schiefgehen, wenn zum Beispiel noch ein Formular auf dieses Element zugreift. In diesem Fall zeigt die Prozedur die entsprechende Fehlermeldung an, die wir zuvor neben der Fehlernummer in den beiden Variablen **lngError** und **strError** gespeichert haben.

Ist kein Fehler aufgetreten, wurde die Tabelle erfolgreich gelöscht. In diesem Fall sorgt ein Aufruf der Methode **Application.RefreshDatabaseWindow** dafür, dass die gelöschte Tabelle direkt aus dem Navigationsbereich verschwindet. Außerdem stellt die Funktion ihren Rückgabewert auf **True** ein.

Verknüpfungstabelle erstellen

Damit kommen wir endlich zu der Funktion, mit der wir die neue Verknüpfungstabelle erstellen. Diese heißt **AddManyToManyTable** und erwartet die folgenden Parameter:

- **strLinkTable**: Name der zu erstellenden Verknüpfungstabelle
- **strPrimaryKey**: Name des zu erstellenden Primärschlüsselfeldes
- **strForeignKeyM**: Name des Fremdschlüsselfeldes zur ersten verknüpften Tabelle

- **strForeignKeyN**: Name des Fremdschlüsselfeldes zur zweiten verknüpften Tabelle

```
Public Function AddManyToManyTable(strLinkTable As String, strPrimaryKey As String, _
        strForeignKeyM As String, strForeignKeyN As String)
    Dim db As DAO.Database
    Dim tdf As DAO.TableDef
    Dim fld As DAO.Field
    Dim idx As DAO.Index
    On Error Resume Next
    Set db = CurrentDb
    Set tdf = db.CreateTableDef(strLinkTable)
    Set fld = tdf.CreateField(strPrimaryKey, dbLong)
    fld.Attributes = fld.Attributes + dbAutoIncrField
    tdf.Fields.Append fld
    Set idx = tdf.CreateIndex("PrimaryKey")
    With idx
        .Primary = True
        .Fields.Append .CreateField(strPrimaryKey)
    End With
    tdf.Indexes.Append idx
    Set fld = tdf.CreateField(strForeignKeyM, dbLong)
    tdf.Fields.Append fld
    Set fld = tdf.CreateField(strForeignKeyN, dbLong)
    tdf.Fields.Append fld
    Set idx = tdf.CreateIndex("UniqueKey")
    With idx
        .Unique = True
        .Fields.Append .CreateField(strForeignKeyM)
        .Fields.Append .CreateField(strForeignKeyN)
    End With
    tdf.Indexes.Append idx
    db.TableDefs.Append tdf
    db.TableDefs.Refresh
    If Err.Number = 0 Then
        Application.RefreshDatabaseWindow
        AddManyToManyTable = True
    Else
        MsgBox Err.Number & vbCrLf & vbCrLf & Err.Description
    End If
End Function
```

Listing 3: Anlegen der Verknüpfungstabelle

Die Funktion **AddManyToManyTable** finden Sie in Listing 3. Sie erstellt zunächst einen Verweis auf das aktuelle **Database**-Objekt. Dann legt sie ein neues **TableDef**-Objekt mit dem Namen aus **strLinkTable** an.

Anschließend erstellt sie das Primärschlüsselfeld mit der **CreateField**-Methode des neuen **TableDef**-Objekts und gibt diesem den Namen aus **strPrimaryKey** und den Datentyp **Long**. Dann erweitert sie die Attribute über die Eigenschaft **Attributes** um das Attribut **dbAutoIncrField**, damit wir ein Autowertfeld erhalten, und hängt das Feld mit der **Append**-Methode an die Auflistung **Fields** der neuen Tabelle an.

Danach erstellt die Tabelle den Index namens **PrimaryKey** für das Primärschlüsselfeld. Um den Index als Primärindex zu definieren, stellen wir die Eigenschaft **Primary** auf den Wert **True** ein.

Schließlich hängen wir dem Index das neue Feld mit dem Namen aus **strPrimaryKey** mit der **Append**-Methode an die **Fields**-Auflistung an und fügen den Index wiederum mit Append der **Indexes**-Auflistung des **TableDef**-Objekts an.

Wenn wir schon beim Erstellen von Indizes sind, legen wir auch gleich noch den zusammengesetzten, eindeutigen Index für die beiden Fremdschlüsselfelder der Tabelle an.

```
Private Sub cmdVerknuepfungstabelleAnlegen_Click()
    Dim strMNTabelle As String
    Dim bolNeuErstellen As Boolean
    Dim bolNeuErstellt As Boolean
    strMNTabelle = Me!txtVerknuepfungstabelle
    If ExistsTable(strMNTabelle) = True Then
        If MsgBox("Die Tabelle " & strMNTabelle & " ist bereits vorhanden. Soll diese überschrieben werden?", _
                vbYesNo + vbExclamation, "Tabelle bereits vorhanden") = vbYes Then
            If DeleteTable(strMNTabelle) = True Then
                bolNeuErstellen = True
            End If
        End If
    Else
        bolNeuErstellen = True
    End If
    If bolNeuErstellen = True Then
        If AddManyToManyTable(strMNTabelle, Me!txtPrimaerschluesselfeld, Me!txtNFremdschluesselfeld, _
                Me!txtMFremdschluesselfeld) = True Then
            bolNeuErstellt = True
        End If
    End If
    If bolNeuErstellt = True Then
        MsgBox "Die Tabelle '" & strMNTabelle & "' wurde angelegt.", vbInformation + vbOKOnly, _
            "Tabelle erfolgreich angelegt"
    Else
        MsgBox "Die Tabelle '" & strMNTabelle & "' wurde nicht angelegt.", vbExclamation + vbOKOnly, _
            "Tabelle nicht angelegt"
    End If
End Sub
```

Listing 4: Aufruf der Prozedur zum Anlegen der Verknüpfungstabelle

Diesen erstellen wir mit **CreateIndex** unter dem Namen **UniqueKey**, stellen die Eigenschaft **Unique** auf **Ja** ein und fügen die beiden Felder mit den Namen aus den Parametern **strForeignKeyM** und **strForeignKeyN** an.

Danach fügen wir auch diesen Index mit der **Append**-Methode an die Auflistung **Indexes** der Tabelle an.

Anschließend folgen die beiden Fremdschlüsselfelder, die wir mit **CreateField** erstellen und dabei die Namen aus **strForeignKeyM** und **strForeignKeyN** sowie den Datentyp **dbLong** übergeben. Nun hängen wir das neue **TableDef**-Objekt an die **TableDefs**-Auflistung an und aktualisieren diese mit der **Refresh**-Methode.

Ist während all dieser Anweisungen kein Fehler aufgetreten, aktualisiert die Funktion die Anzeige im Navigationsbereich und gibt den Wert **True** als Funktionswert zurück. Ist doch ein Fehler aufgetreten, gibt die Funktion eine entsprechende Fehlermeldung aus.

Verknüpfungstabelle per Mausklick

Nun fehlt noch die Ereignisprozedur für die Schaltfläche **cmdVerknuepfungstabelleAnlegen**, welche die drei zuvor definierten Funktionen zusammenbringt (siehe Listing 4).

Diese schreibt zunächst den Namen der zu erstellenden Verknüpfungstabelle in die Variable **strMNTabelle**. Dann