HAUFE.
Ausgabe 05/2021

ACCESS

IM UNTERNEHMEN

ACCESS-OPTIONEN IM RIBBON

Ändern Sie wichtige Optionen schnell im Ribbon statt umständlich den Optionen-Dialog aufzurufen (ab S. 60)

In diesem Heft:

ACCESS-ANWENDUNGEN WEITERGEBEN, TEIL 2

Nutzen Sie InnoSetup, um praktische Setups zum Weitergeben Ihrer Anwendungen zu erstellen.

SEITE 51

OPTIONSGRUPPEN LEEREN

Fügen Sie Optionsgruppen die Funktion zum Entfernen der aktuell gewählten Option hinzu.

SEITE 2

REGISTRY PER VBA UND API

Greifen Sie per VBA auf die Registry von Windows hinzu – unter 32-Bit und 64-Bit!

SEITE 29

Optionen per Ribbon

Nervt Sie das auch, wenn Sie immer wieder umständlich den Optionen-Dialog von Access öffnen müssen, um oft verwendete Optionen einzustellen? Datei-Menü aufrufen, auf Optionen klicken, dann überlegen, in welchem Bereich sich die Option überhaupt befindet und so weiter. Wie schön wäre es doch, wenn oft verwendete Optionen einfach im Ribbon angezeigt würden, wo man sie direkt per Mausklick einstellen kann. Genau dafür finden Sie eine Lösung in der aktuellen Ausgabe: Ein COM-Add-In, welches das Ribbon um genau diese Funktion erweitert!

Im Beitrag **Access-Optionen per Ribbon ändern** finden Sie ab Seite 60 eine genaue Beschreibung, wie Sie mit dem neuen twinBASIC ein COM-Add-In programmieren, das einige wichtige Access-Optionen direkt in einem eigenen Reiter im Ribbon anzeigt. Das heißt, Sie bekommen nicht nur eine sofort einsetzbare Lösung, sondern auch noch die Anleitung, wie Sie diese Lösung programmieren und an Ihre Bedürfnisse anpassen können! Immerhin wird nicht jeder von uns die gleichen Optionen regelmäßig nutzen.

Im Rahmen dieser Lösung haben wir uns ein wenig Arbeit gemacht und einmal geschaut, wie Sie die im Optionen-Dialog von Access enthaltenen Optionen überhaupt per VBA ändern können. Der Beitrag **Optionen per VBA für Access 2019** zeigt ab Seite 8, wie Sie die einzelnen Optionen anpassen können. Dabei gibt es grundsätzlich zwei Möglichkeiten: für Optionen die aktuelle Datenbank betreffend über Properties der Datenbank und für allgemeine Access-Optionen die Registry. Anhand dieses Beitrags können Sie für alle Optionen entnehmen, wie Sie diese per VBA ändern können.

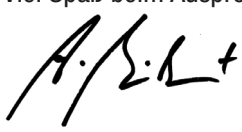
Einige der allgemeinen Access-Optionen befinden sich in einem speziellen Bereich der Registry, den Sie mit speziellen VBA-Befehlen lesen und beschreiben können. Andere Optionen, etwa diejenigen, die sich auf alle Office-Anwendungen beziehen, finden Sie an anderen Stellen in der Registry. Diese können Sie nur über API-Funktionen manipulieren. Deshalb haben wir uns auch angesehen, wie dies gelingt – das Ergebnis lesen Sie ab Seite 29 unter dem Titel **Registry per VBA, 32- und 64-Bit**.

Da wir in unserer Lösung Ribbon-Steuerelemente zum Einstellen von Optionen hinzufügen, nutzen wir auch einige Callbackfunktionen. Diese lauten für COM-Add-Ins allerdings anders als für VBA. Deshalb haben wir den Aufbau dieser Funktionen samt Parametern angeschaut. Das Resultat liefert der Beitrag **Ribbon: Callback-Signaturen für VBA und VB6** ab Seite 40.

Das in der vorherigen Ausgabe begonnene Thema Weitergabe von Access-Anwendungen führen wir ab Seite 51 mit dem Beitrag **Setup für Access: Umsetzung mit InnoSetup** fort.Das Steuerelement **Optionsgruppe** bietet eine einfache Möglichkeit zur Auswahl eines Wertes aus mehreren Optionen. Allerdings können Sie diese nach einmal erfolgter Auswahl nicht mehr leeren. Dafür haben wir eine Lösung entwickelt, die Sie unter **Optionsgruppe leeren mit Klasse** ab Seite 2 lesen.

Schließlich schauen wir uns noch an, wie der Datei speichern-Dialog 64-Bit-kompatibel realisiert wird (**API-Funktion GetSaveFileDialog (32-Bit und 64-Bit)**, ab Seite 23), was eigentlich mit 32-Bit- und 64-Bit-Versionen gemeint ist (**32-Bit, 64-Bit, VBA-Version und Co.** ab Seite 28) und Sie **Benutzerdefinierte Bilder in twinBASIC anzeigen** (ab Seite 48).

Viel Spaß beim Ausprobieren!



Ihr André Minhorst

Optionsgruppe leeren mit Klasse

Optionsgruppen sind praktische Steuerelemente für die Auswahl einiger weniger, vorab fest definierter Optionen. Leider bietet dieses Steuerelement nach einmaliger Auswahl nicht mehr die Möglichkeit, dieses wieder zu leeren. Im vorliegenden Beitrag schauen wir uns an, wie das grundsätzlich zu erledigen ist. Außerdem erstellen wir eine Klasse, mit der Sie den dazu benötigten Code für die Nutzung in mehreren Optionsgruppen wiederverwenden können, statt ihn jedes Mal zu reproduzieren.

Vorbereitung

Für die Beispiele in diesem Beitrag legen wir ein Formular mit einer Optiongruppe namens **ogrOptionen** an. Dieser fügen wir drei Optionen hinzu, deren Wert für die Eigenschaft **Optionswert** wir auf **1**, **2** und **3** festlegen und die wir **opt1**, **opt2** und **opt3** benennen. Der Entwurf dieses Formulars sieht wie in Bild 1 aus.

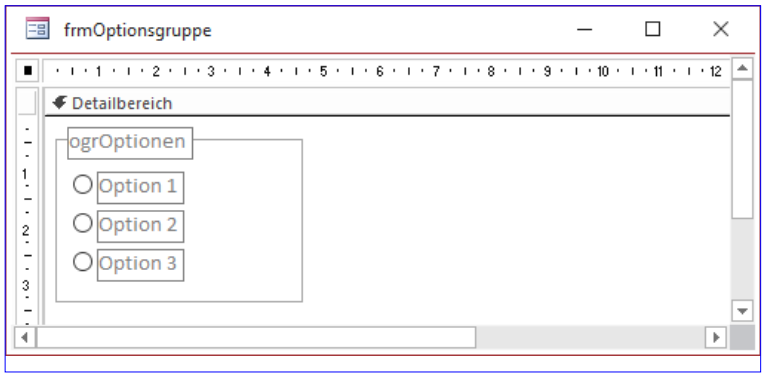


Bild 1: Entwurf des Formulars mit der Optionsgruppe

Dieses Formular kopieren wir direkt einmal, und zwar unter dem Namen **frmOptionenMitKlasse**. Wir schauen uns also zuerst an, wie wir das Leeren der Optionsgruppe direkt im Klassenmodul des Formulars erledigen.

Danach programmieren wir Klassen, mit denen wir das Verhalten abbilden und wiederverwendbar machen, und wenden diese in der Kopie des Formulars an.

Gewünschtes Verhalten

Wir wollen erreichen, dass der Benutzer sowohl mit der Maus als auch mit der Tastatur eine einmal getätigte Auswahl in der Optionsgruppe wieder rückgängig machen kann.

Dazu untersuchen wir zwei Ereignisse: das Loslassen der linken Maustaste sowie das Betätigen der Leertaste. Hier prüfen wir jeweils, ob die aktuelle Option gerade aktiviert ist und nur in diesem Fall soll die Optionsgruppe geleert werden.

Ereignisse zum Abbilden des Verhaltens

Dazu nutzen wir zwei Ereignisse der jeweiligen Optionsfelder, nämlich **Bei Maustaste auf** und **Bei Taste auf**. Diese legen wir für alle vorhandenen Optionsfelder an. Für das erste Optionsfeld sieht das im ersten Schritt dann wie folgt aus:

```
Private Sub opt1_MouseUp(Button As Integer, _
                Shift As Integer, X As Single, Y As Single)

End Sub

Private Sub opt1_KeyUp(KeyCode As Integer, _
                                    Shift As Integer)

End Sub
```

Hier brauchen wir nun Action. Für das Loslassen des Mauszeigers sieht das wie folgt aus:

- Prüfen, ob die linke Maustaste gedrückt wurde
- Prüfen, ob die Maus vor dem Loslassen die aktuell selektierte Option angeklickt hat
- Dann die Optionsgruppe leeren

Für das Ereignis **Bei Taste auf** stehen folgende Aufgaben an:

- Prüfen, ob die Leertaste gedrückt wurde
- Prüfen, ob beim Betätigen der Leertaste der Fokus auf der aktuell selektierten Option liegt
- Dann die Optionsgruppe leeren

Individueller Code für diese Optionsgruppe

Die Lösung für die eingangs beschriebene Optionsgruppe finden Sie in Listing 1. Wir kümmern uns zunächst um die Mausereignisse. Dazu schauen wir uns den Code für das Ereignis **opt1_MouseUp** an. Die Prozedur vergleicht zunächst den Wert des Parameters **Button** mit dem Wert der Konstanten

```
Private Sub opt1_MouseUp(Button As Integer, Shift As Integer, X As Single, Y As Single)
    If Button = acLeftButton Then
        UpdateOptiongroup Me!ogrOptionen, Me!opt1
    End If
End Sub

Private Sub opt2_MouseUp(Button As Integer, Shift As Integer, X As Single, Y As Single)
    If Button = acLeftButton Then
        UpdateOptiongroup Me!ogrOptionen, Me!opt2
    End If
End Sub

Private Sub opt3_MouseUp(Button As Integer, Shift As Integer, X As Single, Y As Single)
    If Button = acLeftButton Then
        UpdateOptiongroup Me!ogrOptionen, Me!opt3
    End If
End Sub

Private Sub opt1_KeyUp(KeyCode As Integer, Shift As Integer)
     Select Case KeyCode
         Case vbKeySpace
             UpdateOptiongroup Me!ogrOptionen, Me!opt1
     End Select
End Sub

Private Sub opt2_KeyUp(KeyCode As Integer, Shift As Integer)
     Select Case KeyCode
         Case vbKeySpace
             UpdateOptiongroup Me!ogrOptionen, Me!opt2
     End Select
End Sub

Private Sub opt3_KeyUp(KeyCode As Integer, Shift As Integer)
     Select Case KeyCode
         Case vbKeySpace
             UpdateOptiongroup Me!ogrOptionen, Me!opt2
     End Select
End Sub

Private Sub UpdateOptiongroup(ogr As OptionGroup, opt As OptionButton)
     If ogr = opt.OptionValue Then
         ogr = Null
     End If
End Sub
```

Listing 1: Prozeduren, um die aktuelle Option wieder abzuwählen

Optionen per VBA für Access 2019

In einem früheren Beitrag namens »Access-Optionen gestern und heute« haben wir uns einmal angesehen, welche Access-Optionen es gibt und wie Sie diese per VBA einstellen können – unter anderem mit den Methoden SetOption oder über die Eigenschaften des Database-Objekts der aktuell geöffneten Datenbank. Damals ging es noch um den Optionen-Dialog von Access 2003, der sich mittlerweile stark verändert hat. Um diese aktualisierte Version soll es in diesem Beitrag gehen. Grundlage ist dabei die Access 365-Version von Mitte 2021.

Wo findet man die Optionen per VBA?

Bevor wir in die Referenz der Optionen und ihrer VBA-Pendants einsteigen, schauen wir uns an, wie Sie überhaupt Optionen per VBA lesen und schreiben können. Dazu brauchen wir zunächst eine wichtige Unterscheidung: Es gibt Optionen für die Access-Anwendung und für die jeweils geöffnete Datenbank.

Optionen in der Registry

Die Optionen der Access-Anwendung werden in der Registry im Bereich **HKEY_CURRENT_USER** gespeichert und wirken sich somit nach der Änderung ausschließlich auf die vom aktuellen Benutzer geöffneten Access-Instanzen aus. Diese können wir mit den beiden VBA-Befehlen **GetOption** und **SetOption** lesen und schreiben.

Genaugenommen finden Sie die Optionen in der Registry unter **HKEY_CURRENT_USER\SOFTWARE\Microsoft\Office\16.0\Access\Settings**. Die meisten dieser Optionen ändern Sie in der Benutzeroberfläche über die verschiedenen Elemente des Dialogs **Access-Optionen**, ein paar auch über den Dialog **Navigationsoptionen**.

Optionen in den Eigenschaften des Database-Objekts

Die Optionen der Access-Datenbank wiederum beziehen sich nur auf die aktuell geöffnete Access-Datenbank. Die meisten davon befinden sich im Dialog **Access-Optionen** im Bereich **Aktuelle Datenbank**. Tatsächlich werden die Werte für diese Optionen in der **Properties**-Auflistung des **Database**-Objekts der aktuellen Datenbank gespeichert. Sie können diese mit folgender Anweisung einstellen:

```
CurrentDb.Properties("<Eigenschaftsname>") = <Eigenschaftswert>
```

Eine nicht vorhandene Eigenschaft müssen Sie zuvor noch anlegen. Mit folgender Anweisung fragen Sie den Wert im Direktbereich von Access ab:

```
Debug.Print CurrentDb.Properties("<Eigenschaftsname>")
```

Optionen in weiteren Bereichen der Registry

Einige Optionen, etwa die für alle Office-Anwendungen gültigen Optionen, befinden sich in anderen Bereichen der Registry. Für diese gibt es keine einfache VBA-Anweisung wie **SetOption** oder **GetOption**. Um diese zu lesen oder zu schreiben, benötigen Sie spezielle Befehle, die wir in einem weiteren Beitrag namens **Registry per VBA, 32- und 64-Bit** (**www.access-im-unternehmen.de/1323**) untersuchen.

Access-Optionen und ihre VBA-Pendants

Damit steigen wir gleich in die Optionen ein, die Sie über die Benutzeroberfläche anpassen können, und schauen uns an, wie Sie diese unter VBA einstellen können.

Wir haben die Eigenschaften jeweils in den Screenshots der Dialoge nummeriert. Sie finden zu den Nummern die deutsche Bezeichnung (in Klammern), die englische

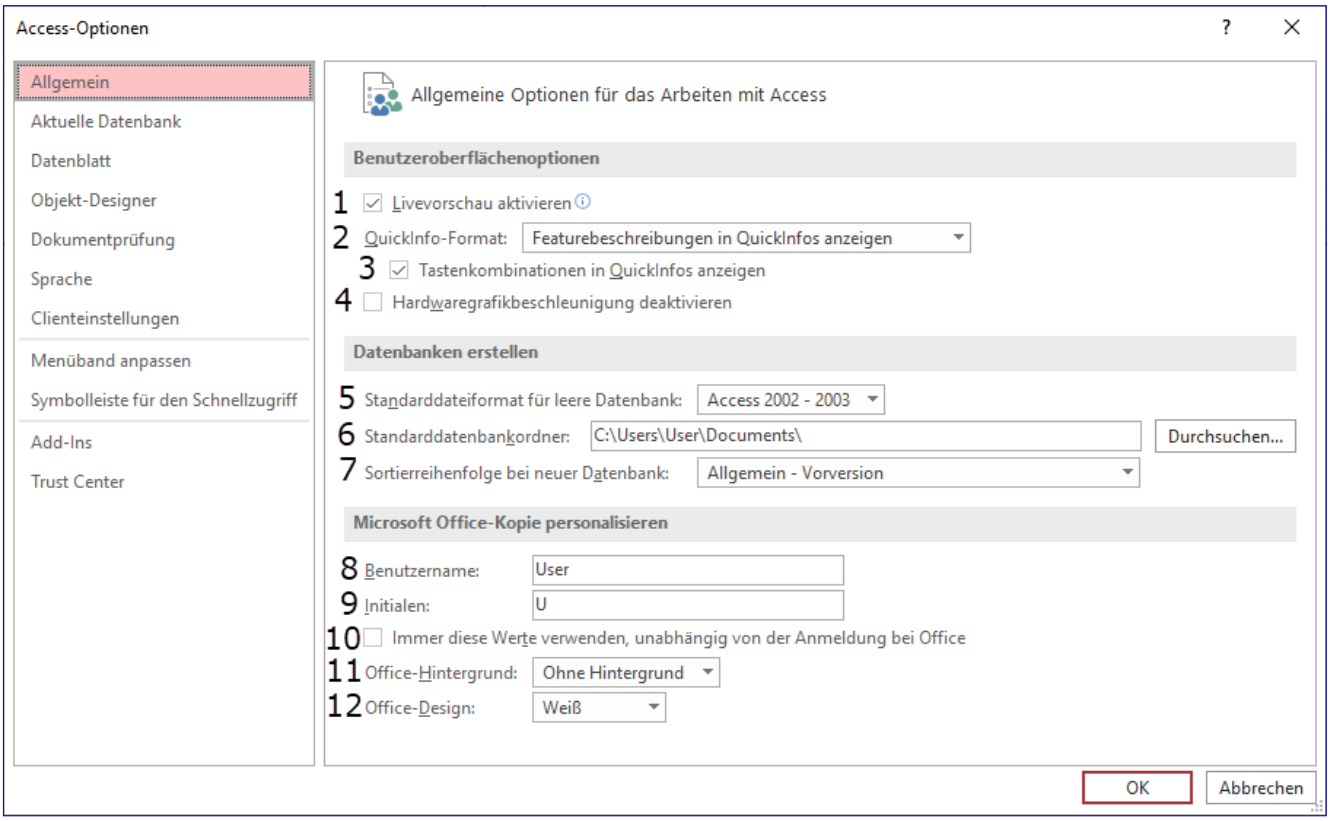


Bild 1: Optionen des Bereichs **Allgemein**

Bezeichnung, den Speicherort der Eigenschaft (Registry/Database oder im Falle von anderen Speicherorten in der Registry denentsprechenden Zweig) und die möglichen Werte.

Optionen des Bereichs Allgemein

Gleich im ersten Bereich des Optionen-Dialogs von Access finden wir einige Optionen, die sich tatsächlich nicht von Access aus per VBA steuern lassen (siehe Bild 1).

- **1** (**Live-Vorschau aktivieren**): **LivePreview**, Registry, **0**: **Nein**, **1**: **Ja**, wird bei erster Änderung angelegt
- **2** (**QuickInfo-Format**): nicht gefunden
- **3** (**Tastenkombinationen in QuickInfos anzeigen**): nicht gefunden
- **4** (**Hardwaregrafikbeschleunigung deaktivieren**):
- **5** (**Standarddateiformat für leere Datenbank**): **Default File Format**, Registry, **Access 2000**: **9**, **Access 2002 - 2003**: **10**, **Access 2007 - 2019, Access 365**: **12**
- **6** (**Standarddatenbankordner**): **Default Database Directory**, Registry, Verzeichnis im String-Format
- **7** (**Sortierreihenfolge bei neuer Datenbank**): **New Database Sort Order**, Registry, verschiedene Werte, zum Beispiel **2052** (**Allgemein - Vorversion**) oder **1033** (**Deutsches Telefonbuch**)
- **8** (**Benutzername**): Office-Einstellung
- **9** (**Initialen**): Office-Einstellung
- **10** (**Immer diese Werte verwenden, unabhängig von der Anmeldung bei Office**): Office-Einstellung

- **11** (**Office-Hintergrund**): Office-Einstellung
- **12** (**Office-Design**): Office-Einstellung

Optionen des Bereichs Aktuelle Datenbank

Die Elemente im Bereich **Aktuelle Datenbank** des Optionen-Dialogs von Access sehen Sie in Bild 2.

Hier sind die Schrauben, an denen Sie mit VBA drehen können:

- **1** (**Anwendungstitel**): **AppTitle**, Database, Anwendungstitel im String-Format, wird bei erster Änderung erstellt
- **2** (**Anwendungssymbol**): **AppIcon**, Database, Pfad zum Icon im String-Format, wird bei erster Änderung erstellt
- **3** (**Als Formular- und Berichtssymbol verwenden**): **UseAppIconForFrmRpt**, Database, Boolean, Neustart erforderlich
- **4** (**Formular anzeigen**): **StartupForm**, Database, Formularname im **String**-Format, wird bei erster Änderung erstellt, Neustart erforderlich
- **5** (**Webanzeigeformular**): im Desktop-Modus nicht verfügbar
- **6** (**Statusleiste anzeigen**): **StartUpShowStatusBar**, Database, Boolean, Neustart erforderlich
- **7** (**Dokumentfensteroptionen**): **UseMdiMode**, Database, **0**: **Dokumente im Registerkartenformat**, **1**: **Überlappende Fenster**, Neustart erforderlich

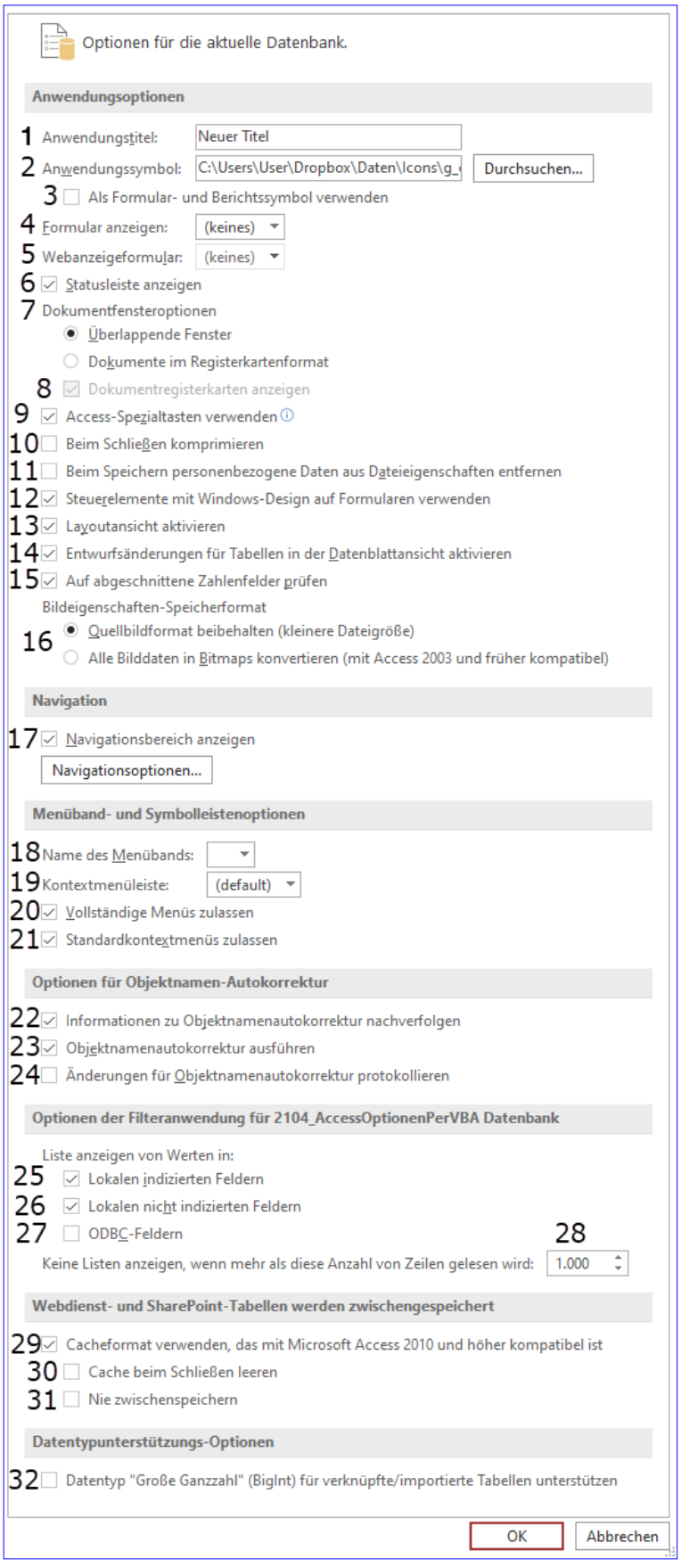


Bild 2: Optionen des Bereichs **Aktuelle Datenbank**

- **8** (**Dokumentregisterkarten anzeigen**): **ShowDocumentTabs**, Database, Boolean-Format, Neustart erforderlich
- **9** (**Access-Spezialtasten verwenden**): **AllowSpecialKeys**, Database, Boolean, Neustart erforderlich
- **10** (**Beim Schließen komprimieren**): **Auto Compact**, Database, **1**: Ja, **0**: Nein, Neustart erforderlich
- **11** (**Beim Speichern personenbezogene Daten aus Dateieigenschaften entfernen**): **Remove Personal Information**, Database, **1**: Ja, **0**: Nein, wird bei erster Änderung erstellt, Neustart erforderlich
- **12** (**Steuerelemente mit Windows-Design auf Formularen verwenden**): **Themed Form Controls**, Database, **1**: **Ja**, **2**: **Nein**
- **13** (**Layoutansicht aktivieren**): **DesignWithData**, Database, Boolean
- **14** (**Entwurfsänderungen für Tabellen in der Datenblattansicht aktivieren**): **AllowDatasheetSchema**, Database, Boolean
- **15** (**Auf abgeschnittene Zahlenfelder prüfen**): **CheckTruncatedNumFields**, Database, Boolean
- **16** (**Bildeigenschaften-Speicherformat**): **Picture Property Storage Format**, Database, **0**: **Quellbildformat beibehalten (kleinere Dateigröße)**, **1**: **Alle Bilddaten in Bitmaps konvertieren (mit Access 2003 und früher kompatibel)**
- **17** (**Navigationsbereich anzeigen**): **StartUpShowDBWindow**, Database, Boolean
- **18** (**Name des Menübands**): **CustomRibbonID**, Database, Name des Ribbons als String, wird bei erster Änderung erstellt
- **19** (**Kontextmenüleiste**): **StartupShortcutMenuBar**, Database, Name der Kontextmenüleiste, wird bei erster Änderung erstellt
- **20** (**Vollständige Menüs zulassen**): **AllowFullMenus**, Database, Boolean-Wert
- **21** (**Standardkontextmenüs zulassen**): **AllowShortcutMenus**, Database, Boolean-Wert
- **22** (**Informationen zu Objektnamenautokorrektur nachverfolgen**): **Track Name AutoCorrect Info**, Database, **0**: **Nein**, **1**: **Ja**, wird bei erster Änderung erstellt
- **23** (**Objektnamenautokorrektur ausführen**): **Perform Name AutoCorrect**, Database, **0**: **Nein**, **1**: **Ja**
- **24** (**Änderungen für Objektnamenautokorrektur protokollieren**): **Log Name AutoCorrect Changes**, Database, **0**: **Nein**, **1**: **Ja**, wird bei erster Änderung erstellt
- **25** (**Liste anzeigen von Werten in ... Lokalen indizierten Feldern**): **Show Values in Indexed**, Database, **0**: **Nein**, **1**: **Ja**
- **26** (**Liste anzeigen von Werten in ... Lokalen nicht indizierten Feldern**): **Show Values in Non-Indexed**, Database, **0**: **Nein**, **1**: **Ja**
- **27** (**Liste anzeigen von Werten in ... ODBC-Feldern**): **Show Values in Remote**, Database , **0**: **Nein**, **1**: **Ja**
- **28** (**Keine Listen anzeigen, wenn mehr als diese Anzahl von Zeilen gelesen wird**): **Show Values Limit**, Database, Zahl
- **29** (**Cacheformat verwenden, das mit Microsoft Access 2010 und höher kompatibel ist**): **Use Microsoft 2007 compatible cache**, Database, **0**: **Nein**, **1**: **Ja**

- **30** (**Cache beim Schließen leeren**): **Clear Cache on Close**, Database, **0**: **Nein**, **1**: **Ja**
- **31** (**Nie zwischenspeichern**): **Never Cache**, Database, **0**: **Nein**, **1**: **Ja**
- **32** (**Datentyp "Große Ganzzahl" (BigInt) für verknüpfte/importierte Tabellen unterstützen**): **Use BigInt for linking and importing data**, Database,, **0**: **Nein**, **1**: **Ja**

Optionen des Dialogs Navigationsoptionen

Diesen Dialog öffnen Sie entweder über die Schaltfläche **Navigationsoptionen...** im Bereich **Aktuelle Datenbank** der Access-Optionen oder über das Kontextmenü des Titels des Navigationsbereichs (siehe Bild 3).

Hier finden Sie die folgenden Optionen:

- **1** (**Ausblendete Objekte anzeigen**): **Show Hidden Objects**: Registry, **0**: **Nein**, **1**: **Ja**
- **2** (**Systemobjekte anzeigen**): **Show System Objects**: Registry, **0**: **Nein**, **1**: **Ja**

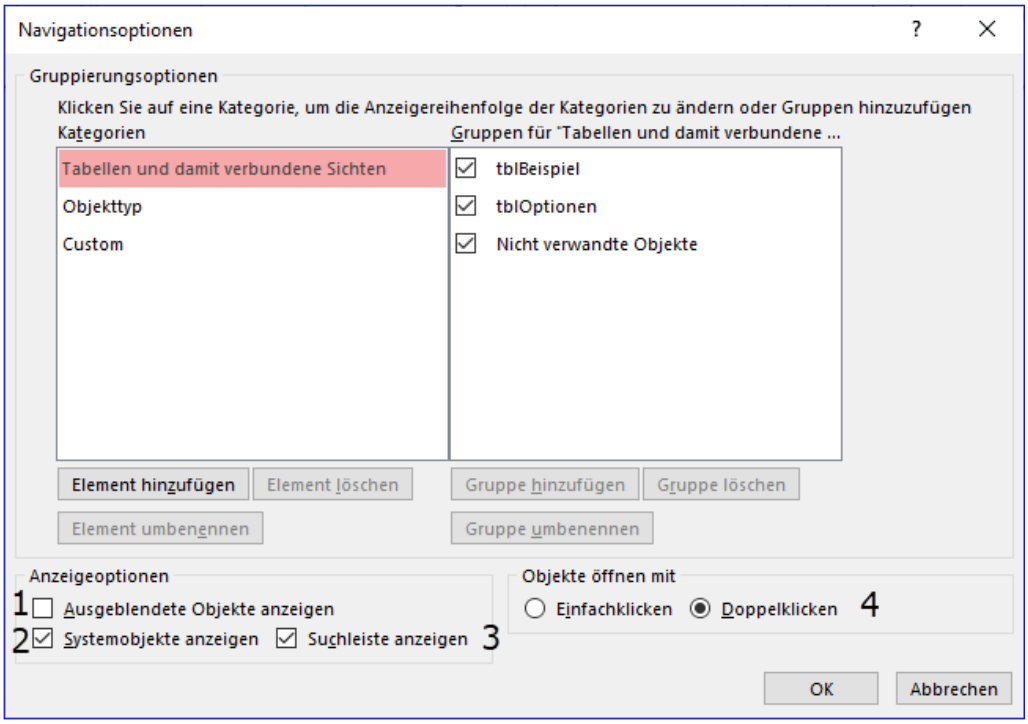


Bild 3: Optionen des Bereichs **Navigationsoptionen**

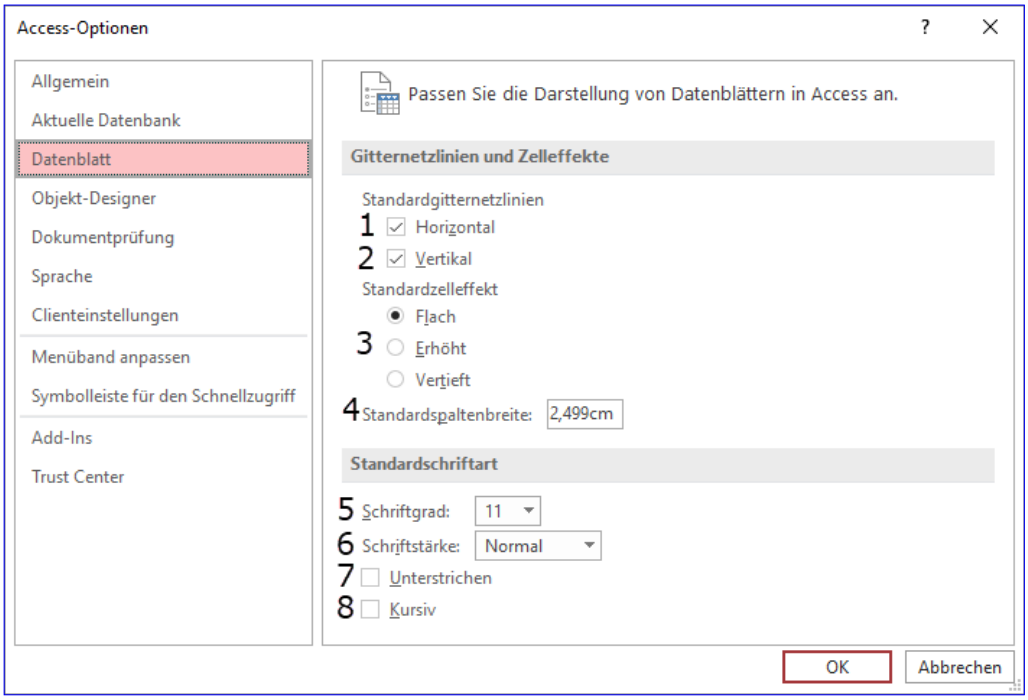


Bild 4: Optionen des Bereichs **Datenblatt**

- **3** (**Suchleiste anzeigen**): **Show Navigation Pane Search Bar**, Database, Boolean

- **4** (**Objekte öffnen mit**): **Database Explorer Click Behavior**, Registry, **0**: **Nein**, **1**: **Ja**, wird bei erster Änderung erstellt

Optionen des Bereichs Datenblatt

Der Bereich **Datenblatt** hält die Optionen aus Bild 4 bereit. Diese bieten die folgenden Möglichkeiten zur Anpassung per VBA:

- **1** (**Standardgitternetzlinien – Horizontal**): **Default Gridlines Horizontal**, Registry, **0**: **Nein**, **1**: **Ja**, wird bei erster Änderung erstellt

- **2** (**Standardgitternetzlinien – Vertikal**): **Default Gridlines Vertical**, Registry, **0**: **Nein**, **1**: **Ja**, wird bei erster Änderung erstellt

- **3** (**Standardzelleffekt**): **Default Cell Effect**, Registry, **0**: **Flach**, **1**: **Erhöht**, **2**: **Vertieft**, wird bei erster Änderung erstellt

- **4** (**Standardspaltenbreite**): **Default Column Width**, Registry, Breite in Twips, wird bei erster Änderung erstellt

- **5** (**Schriftgrad**): **Default Font Size**, Registry, Zahl, wird bei erster Änderung erstellt

- **6** (**Schriftstärke**): **Default Font Weight**, Registry, **0**: **Extra dünn**, **1**: **Sehr Dünn**, **2**: **Dünn**, 3: Normal, **4**: **Mittel**, **5**: **Halb Fett**, **6**: **Fett**, **7**: **Sehr Fett**, **8**: **Extra Fett** wird bei erster Verwendung erstellt

- **7** (**Unterstrichen**): **Default Font Underline**, Registry, **0**: **Nein**, **1**: **Ja**, wird bei erster Verwendung erstellt

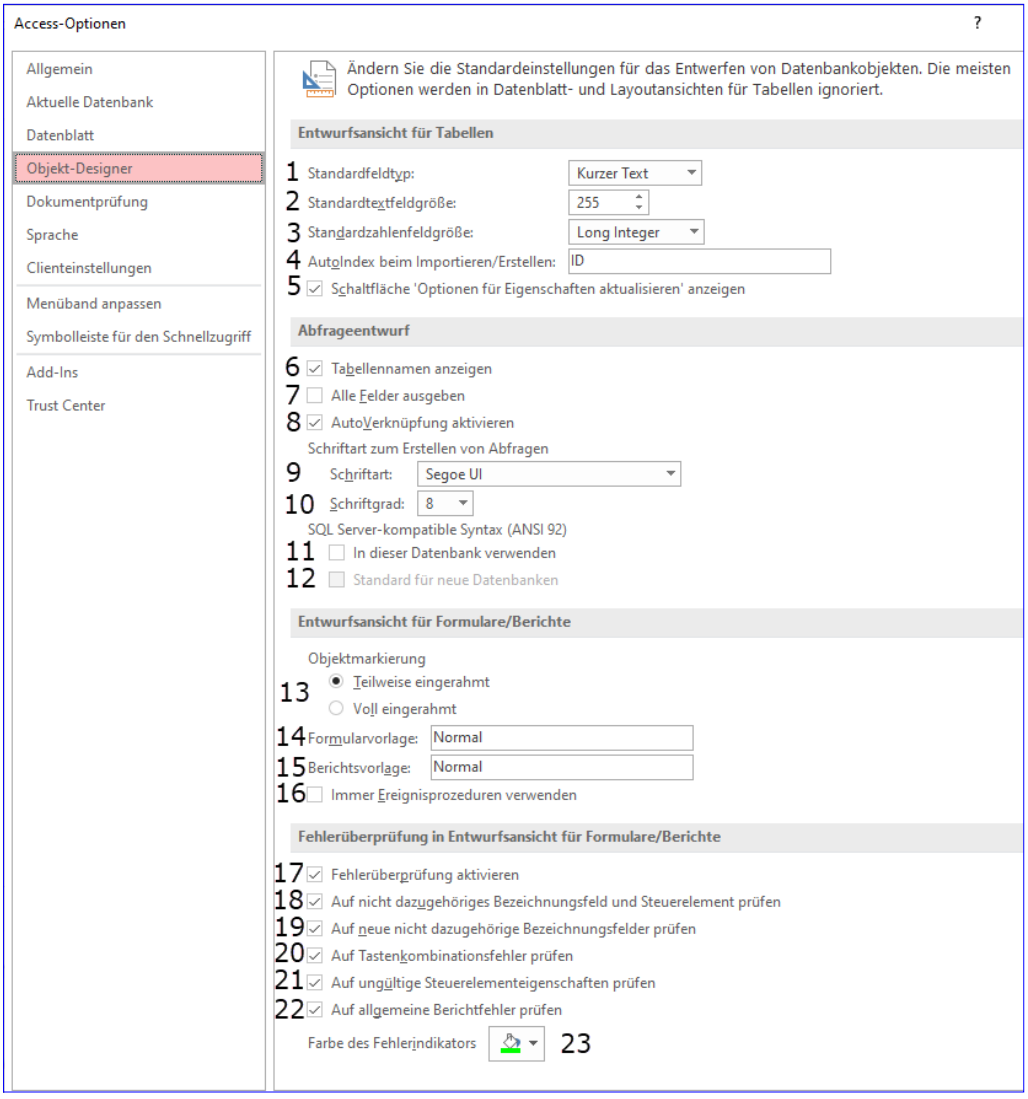


Bild 5: Optionen des Bereichs **Objekt-Designer**

- **8** (**Kursiv**): **Default Font Italic**, Registry, **0**: **Nein**, **1**: **Ja**, wird bei erster Änderung erstellt

Optionen des Bereichs Objekt-Designer

Die Optionen für die Objekt-Designer finden Sie in Bild 5. Und hier sind die enthaltenen Optionen:

- **1** (**Standardfeldtyp**): **Default Field Type**, Registry, **0**: **Kurzer Text**, **1**: **Langer Text**, **2**: **Zahl**, **3**: **Datum/Uhrzeit**, 4: **Währung**, **5**: **Ja/Nein**, **6**: **Link**
- **2** (**Standardtextfeldgröße**): **Default Text Field Size**, Registry, Zahl
- **3** (**Standardzahlenfeldgröße**): **Default Number Field Size**, Registry, **0**: **Byte**, **1**: **Integer**, **2**: **Long Integer**, **3**: **Single**, **4**: **Double**, **5**: **Decimal**, **6**: **Replikations-ID**
- **4** (**Autoindex beim Importieren/Bestellen**): **Autoindex on Import/Create**, Registry, Kommaseparierte Liste der Felder, für die ein Index angelegt werden soll
- **5** (**Schaltfläche 'Optionen für Eigenschaften aktualisieren' anzeigen**): **Show Property Update Option Buttons**, Registry, **0**: **Nein**, **1**: **Ja**
- **6** (**Tabellennamen anzeigen**): **Show Table Names**, Registry, **0**: **Nein**, **1**: **Ja**
- **7** (**Alle Felder ausgeben**): **Output all Fields**, Registry, **0**: **Nein**, **1**: **Ja**
- **8** (**AutoVerknüpfung aktivieren**): **Enable AutoJoin**, Registry, **0**: **Nein**, **1**: **Ja**, wird bei erster Änderung angelegt
- **9** (**Schriftart zum Erstellen von Abfragen – Schriftart**): **Query Design Font Name**, Registry, Name als String
- **10** (**Schriftart zum Erstellen von Abfragen – Schriftgrad**): **Query Design Font Size**, Registry, Zahlenwert
- **11** (**SQL Server-kompatible Syntax (ANSI 92) – In dieser Datenbank verwenden**):
- **12** (**SQL Server-kompatible Syntax (ANSI 92) – Standard für neue Datenbanken**):
- **13** (**Objektmarkierung**): **Selection Behavior**, Registry, **0**: **Teilweise eingerahmt**, **1**: **Voll eingerahmt**
- **14** (**Formularvorlage**): **Form Template**, Registry, Name als String, wird bei erster Änderung angelegt
- **15** (**Berichtsvorlage**): **Report Template**, Registry, Name als String, wird bei erster Änderung angelegt
- **16** (**Immer Ereignisprozeduren verwenden**): **Always use Event Procedure**, **0**: **Nein**, **1**: **Ja**, wird bei erster Änderung angelegt
- **17** (**Fehlerüberprüfung aktivieren**): **Enable Error Checking**, Registry, **0**: **Nein**, **1**: **Ja**, wird bei erster Änderung angelegt
- **18** (**Auf nicht dazugehöriges Bezeichnungsfeld und Steuerelement prüfen**): **Unassociated Label and Control Error Checking**, Registry, **0**: **Nein**, **1**: **Ja**, wird bei erster Änderung angelegt
- **19** (**Auf neue nicht dazugehörige Bezeichnungsfelder prüfen**): **New Unassociated Label Error Checking**, Registry, **0**: **Nein**, **1**: **Ja**, wird bei erster Änderung angelegt
- **20** (**Auf Tastenkombinationsfehler prüfen**): **Keyboard Shortcut Errors Error Checking**, Registry, **0**: **Nein**, **1**: **Ja**, wird bei erster Änderung angelegt
- **21** (**Auf ungültige Steuerelementeigenschaften prüfen**): **Invalid Control Source Error Checking**, Registry, **0**: **Nein**, **1**: **Ja**, wird bei erster Änderung angelegt

API-Funktion GetSaveFileDialog (32-Bit und 64-Bit)

Das Öffnen eines Dialogs zum Auswählen des Namens einer zu speichernden Datei erledigen Sie beispielsweise mit der API-Funktion »GetSaveFileDialog«. Diese stellen wir im vorliegenden Beitrag für 32-Bit- und 64-Bit-Office vor. Dabei stellen wir auch die Änderungen heraus, die für das Update einer eventuell bereits bestehenden 32-Bit-Version auf die 64-Bit-Version notwendig sind.

Die API-Funktion **GetSaveFileDialog** ist eine Funktion der Bibliothek **comdlg32.dll**. Sie erwartet alle Parameter, die das Aussehen des Speichern-Dialogs beeinflussen, in Form eines Typs namens **OPENFILENAME**. Er heißt **OPENFILENAME**, weil er in gleicher Form auch für die API-Funktion zum Anzeigen eines **Datei öffnen**-Dialogs zum Einsatz kommt. Sie liefert den Pfad der zu speichernden Datei zurück oder eine leere Zeichenkette, falls kein Name ausgewählt wurde – zum Beispiel, weil der Benutzer den Dialog mit der **Abbrechen**-Schaltfläche geschlossen hat.

Die Wrapper-Funktion GetSaveFile

Vor dem Aufruf der API-Funktion **GetSaveFileDialog** füllt man also den Typ **OPENFILENAME** mit den gewünschten Einstellungen.

Da Sie das nicht jedes Mal erledigen sollen, wenn Sie diese Funktion zu einer Ihrer Anwendungen hinzufügen wollen, liefern wir eine Wrapperfunktion namens **GetSaveFile**. Diese sieht wie in Listing 1 aus.

```
Public Function GetSaveFile(Optional strStartDir As String, _
        Optional strDefFileName As String, _
        Optional strFilter As String = "Alle Dateien (*.*)", _
        Optional strTitle As String) As String
    Dim udtOpenFileName As OPENFILENAME
    Dim strExt As String
    On Error GoTo Fehler
    If Len(strStartDir) = 0 Then
        strStartDir = CurrentProject.Path
    End If
    With udtOpenFileName
        .nStructSize = LenB(udtOpenFileName)
        .hwndOwner = Application.hWndAccessApp
        strFilter = strFilter & vbNullChar & vbNullChar
        .sFilter = strFilter
        .nFilterIndex = 1
        .sInitDir = strStartDir & vbNullChar
        .sDlgTitle = strTitle
        .sFile = Space$(256) & vbNullChar
        .nFileSize = Len(.sFile)
        If Len(strDefFileName) <> 0 Then
            Mid(.sFile, 1) = strDefFileName
        End If
        .sFileTitle = Space$(256) & vbNullChar
        .nTitleSize = Len(.sFileTitle)
        If GetSaveFileName(udtOpenFileName) Then
            GetSaveFile = Left(udtOpenFileName.sFile, InStr(.sFile, vbNullChar) - 1)
        Else
            GetSaveFile = ""
        End If
    End With
Ende:
    Exit Function
Fehler:
    MsgBox Err.Description, vbCritical, "GetSaveFileName"
    Resume Ende
End Function
```

Listing 1: Wrapperfunktion für die API-Funktion **GetSaveFileName**

Dies sind die Parameter:

- **strStartDir**: Hier geben Sie das Verzeichnis an, das beim Öffnen des Dialogs vorausgewählt werden soll.
- **strDefFileName**: Vordefinierter Dateiname, der beim Öffnen des Dialogs im Feld **Dateiname** eingetragen werden soll.
- **strFilter**: Ein Ausdruck, der angibt, welche Dateitypen als Speichername verwendet werden dürfen.
- **strTitle**: Titel des Dialogs

Ablauf von GetSaveFile

Der Wrapperfunktion für die API-Funktion **GetSaveFileName** haben wir einen etwas verkürzten Namen gegeben – eben **GetSaveFile**. Sie definiert als Erstes eine Variable namens **udtOpenFileName** des Typs **OPENFILENAME**. Wie dieser genau aussieht und deklariert wird, zeigen wir weiter unten.

Dieser Typ arbeitet wie eine Klasse und hat verschiedene Eigenschaften, die wir mit der Funktion **GetSaveFile** füllen. **nStructSize** nimmt beispielsweise die Größe des Typs **udtOpenFileName** entgegen, die wir mit der Funktion **LenB** ermitteln. Das ist übrigens auch eine der Änderungen der 64-Bit-Version gegenüber der 32-Bit-Version – früher wurde hier die Funktion **Len** verwendet. **Len** ermittelt die Anzahl der Zeichen einer Zeichenkette, **LenB** die Anzahl der Bytes dieser Zeichenkette. **hwndOwnder** füllen wir mit dem Handle des aktuellen Fensters. Den Filter aus dem Parameter **strFilter** erweitern wir noch um zwei **vbNullChar**, bevor wir ihn der Eigenschaft **sFilter** zuweisen. Gültige Werte dazu schauen wir uns gleich im Anschluss an. Welcher der Filter voreingestellt wird, legt die Funktion für die Eigenschaft **nFilterIndex** fest, in diesem Fall mit dem Wert **1**.

Das mit dem Parameter **strStartDir** übergebene Startverzeichnis ergänzt die Funktion ebenfalls um **vbNullChar** und schreibt sie dann in die Eigenschaft **sInitDir**. Der Titel für den Dialog landet unbehandelt in **sDlgTitle**.

In **sFile** trägt die Funktion eine leere Zeichenkette mit einer Länge von 256 Zeichen und einem abschließenden **vbNullChar** ein. **nFileSize** erhält die Länge von **sFile**.

Wenn der Aufruf einen Wert für den Parameter **strDefFileName** enthält, also den voreinzustellenden Speichernamen, wird dieser vorn in der Eigenschaft **sFile** eingesetzt. Damit die Länge der darin enthaltenen Zeichenkette nicht verändert wird, verwenden wir dazu die **Mid**-Funktion in einer eher unbekannten Art. Der Aufruf **Mid(.sFile, 1) = strDefFileName** sorgt dafür, dass der Inhalt von **strDefFileName** die entsprechenden Zeichen von **.sFile** ab der ersten Position überschreibt.

sFileTitle soll den ermittelten Namen ohne Verzeichnis aufnehmen und wird mit 256 Leerzeichen plus abschließendem **vbNullChar** gefüllt. Auch die Länge diese Zeichenkette landet in **udtOpenFileName**, und zwar in der Eigenschaft **nTitleSize**.

Nachdem alle relevanten Eigenschaften von **udtOpenFileName** gefüllt sind, ruft die Funktion die API-Funktion **GetSaveFileName** mit **udtOpenFileName** als Parameter auf. Liefert diese Funktion den Wert **True** zurück, was der Fall ist, wenn der Benutzer die Schaltfläche **OK** betätigt, schreibt die Funktion den Inhalt von **sFile** aus **udtOpenFileName** in den Rückgabewert der Funktion. Anderenfalls liefert die Funktion eine leere Zeichenkette zurück.

Deklaration der API-Funktion GetSaveFileName

Wichtig für die Einsatzbereitschaft der Funktion **GetSaveFileName** unter 32-Bit- und 64-Bit-Office ist, dass Sie für beide Varianten die richtige Deklaration bereitstellen. Genau genommen ist es sogar so, dass Sie nur prüfen müssen, ob Sie VBA 6.x oder VBA 7.x verwenden. VBA 6.0 kam mit älteren Office-Versionen bis 2007. VBA 7 wurde eingeführt, um die Kompatibilität mit 64-Bit zu sichern. Grundsätzlich müssen wir also prüfen, welche Access-

32-Bit, 64-Bit, VBA-Version und Co.

Mitunter kommt es zu Missverständnissen, wenn es darum geht, die Kompatibilität von VBA-Code für verschiedene Zielversionen sicherzustellen. Dieser Beitrag erläutert in Kürze die wichtigsten Grundlagen.

Von VBA 6.x zu VBA 7.x

Mit Office 2010 wurde eine neue VBA-Version namens VBA 7.0 eingeführt (mittlerweile VBA 7.1). Diese Version stellt unter anderem die Kompatibilität mit den 64-Bit-Versionen von Office und Windows sicher.

Wir hören und lesen oft, dass Benutzer ihre Anwendung nicht mehr nutzen können, weil diese nicht mit der 64-Bit-Version von Office kompatibel ist. Das liegt daran, dass die Deklarationen von API-Funktionen und den davon verwendeten Typen nicht mit der 64-Bit-Version von Office kompatibel sind.

In vielen Beispielen findet man dann eine Unterscheidung, die durch das Prüfen einer Kompilerkonstanten erfolgt. Diese sieht so aus:

```
#If VBA7 Then
    'mit 64-Bit kompatibler Code
#Else
    'nicht mit 64-Bit kompatibler Code
#End If
```

Im ersten Teil finden wir dann schon oft besprochene Änderungen für die Kompatibilität von APIs für 64-Bit wie das Schlüsselwort **PtrSafe** oder den Datentyp **LongPtr**.

Die Unterscheidung könnte zu dem Missverständnis führen, dass **VBA7** synonym mit 64-Bit ist. Das ist aber nicht der Fall. Genau genommen macht diese Unterscheidung Folgendes: Sie hat im ersten Teil 64-Bit-kompatiblen Code, der aber nicht nur unter 64-Bit läuft, sondern auch unter 32-Bit. Wichtig ist, dass es VBA 7.x ist.

Der **Else**-Teil des Codes enthält Code, der auf jeden Fall mit VBA 6.x kompatibel ist, was sich vor allem dadurch darstellt, dass eben noch nicht das Schlüsselwort **PtrSave** oder der Datentyp **LongPtr** in APIs verwendet werden.

Sie brauchen diese umständliche Schreibweise mit der Kompilerkonstanten **VBA7** also nur, wenn Sie noch Code erzeugen, der auch in Anwendungen verwendet werden soll, die in Access 2007 und älter laufen.

Nur eine Version ab Access 2010 nötig

Wenn Sie nur noch Code für Anwendungen schreiben, die unter Access 2010 und neuer laufen, dann brauchen Sie nur den Teil des Codes aus dem ersten Teil der Bedingung **#If VBA 7 Then** abzubilden und können die Bedingung weglassen.

Denn: Die für die Nutzung mit 64-Bit eingeführten Elemente sind unter VBA 7.x auch mit der 32-Bit-Version kompatibel.

Wozu die Kompilerkonstante Win64?

Hier unterscheiden wir zwischen Office- beziehungsweise VBA-Version und der Windows-Version. Windows in der 64-Bit-Version unterstützt die von der 32-Bit-Version bekannten APIs weiter, aber es wurden auch ein paar neue Funktionen eingeführt. Wenn Sie für die 32-Bit- und die 64-Bit-Version von Windows programmieren und unter Windows 64-Bit die neuen Funktionen nutzen wollen, müssen Sie eine entsprechende Unterscheidung mithilfe der Kompilerkonstanten **Win64** treffen:

```
#If Win64 Then
    'neue, nur unter Win64 vorhandene Funktionen
#Else
    'Pendant der Funktion unter Win32
#End If
```

Registry per VBA, 32- und 64-Bit

Die Registry von Windows ist für den einen oder anderen ein Buch mit sieben Siegeln. Tatsache ist: Dort landen manche wichtigen Informationen, die Sie gegebenenfalls einmal mit VBA auslesen wollen, oder Sie wollen dafür sorgen, dass per VBA bestimmte Elemente in der Registry angelegt werden. Wir stellen einige Routinen vor, die Ihnen die Arbeit mit der Registry erleichtern. Gleichzeitig liefern wir den Code in 64-Bit-kompatibler Form.

Per VBA mit der Registry arbeiten

Wann und wo Sie auf eine Anforderung stoßen, die mit dem Lesen oder Schreiben von Registry-Werten per VBA zu tun hat, lassen wir einmal dahingestellt. Wichtig ist allein, dass Sie nach der Lektüre dieses Beitrags auf den Ernstfall vorbereitet sind!

Daher schauen wir uns in diesem Beitrag nicht nur einen Satz von Funktionen an, welche wiederum die API von Windows für den lesenden und schreibenden Zugriff auf die Registry nutzen, sondern liefern auch noch eine Reihe von Beispielen für die Anwendung dieser Funktionen.

Unser konkreter Anlass, uns mit dem Thema VBA-Zugriff auf die Registry auseinanderzusetzen, war der Beitrag **Optionen per VBA für Access 2019** (**www.access-im-unternehmen.de/1320**). Hier haben wir untersucht, welche Einstellungen des Dialogs Access-Optionen in der Registry landen. Um schnell prüfen zu können, ob sich nach dem Ändern einer Option unter Access einer der Ein-

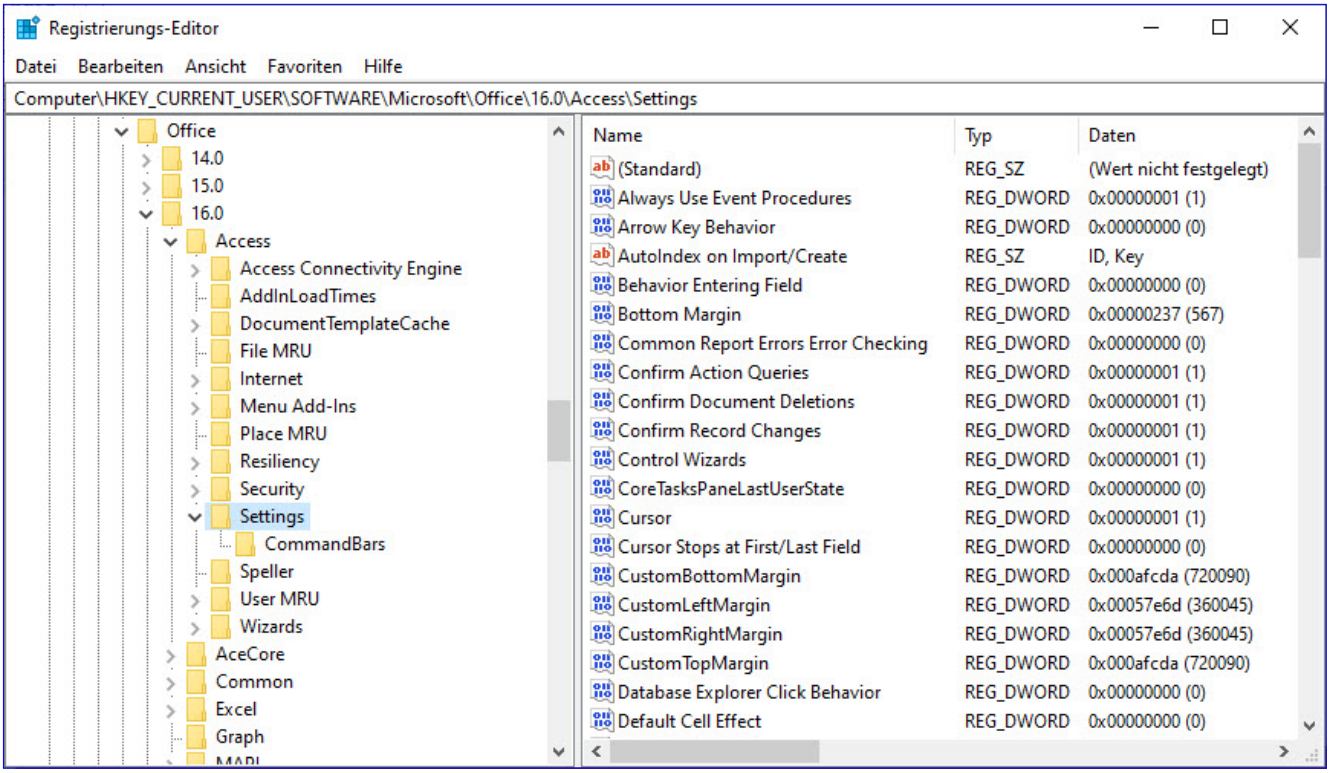


Bild 1: Dieser Bereich der Registry speichert die meisten der Access-Optionen.

träge in der Registry geändert hat, haben wir eine Tabelle erstellt, die alle bereits in der Registry vorhandenen Werte enthält. Diese Werte haben wir initial einmal aus dem entsprechenden Bereich der Registry ausgelesen.

Anschließend haben wir nach jeder Änderung eine Prozedur durchlaufen lassen, welche alle Werte der Registry mit den bestehenden Werten in der Tabelle abgeglichen und eventuelle Unterschiede oder gar neue Werte im Direktbereich ausgegeben hat.

Access-Einstellungen in der Registry

Den Bereich der Registry, der die Access-Optionen enthält, findet sich unter **HKEY_CURRENT_USER** im Ordner **SOFTWARE\Microsoft\Office\16.0\Access\Settings**. Der Screenshot aus Bild 1 zeigt einige der Einstellungen.

Auf diesen Ordner wollen wir uns in diesem Beitrag konzentrieren – wie lesen die Informationen aus, legen neue Werte an, erstellen Unterordner und so weiter. Vorher schauen wir uns allerdings noch die dazu notwendigen Funktionen im Modul **mdlRegistry** an.

Das Modul mdlRegistry und die Registry-Funktionen

Da das Modul die Deklaration von API-Funktion enthält sowie dafür vorgesehene Wrapperfunktionen, benötigen wir auch einige **Enum**- und **Const**-Elemente. Diese sehen wie folgt aus:

```
Public Enum Key
    HKEY_CLASSES_ROOT = &H80000000
    HKEY_CURRENT_USER = &H80000001
    HKEY_LOCAL_MACHINE = &H80000002
    HKEY_USERS = &H80000003
End Enum

Public Enum DataType
    dtString = 1
    dtNumber = 4
End Enum

Global Const ERROR_NONE = 0
Global Const ERROR_BADDB = 1
Global Const ERROR_BADKEY = 2
Global Const ERROR_CANTOPEN = 3
Global Const ERROR_CANTREAD = 4
Global Const ERROR_CANTWRITE = 5
Global Const ERROR_OUTOFMEMORY = 6
Global Const ERROR_INVALID_PARAMETER = 7
Global Const ERROR_ACCESS_DENIED = 8
Global Const ERROR_INVALID_PARAMETERS = 87
Global Const ERROR_NO_MORE_ITEMS = 259
Global Const KEY_ALL_ACCESS = &H3F
Global Const REG_OPTION_NON_VOLATILE = 0
```

Neben diesen Elementen benötigen wir die Deklarationen der eigentlichen API-Funktionen. Dabei handelt es sich um 13 Funktionen, deren Deklaration Sie in Listing 1 finden.

Wrapperfunktion zum Auslesen von Werten

Die erste Wrapperfunktion soll das Auslesen eines Wertes für einen Eintrag eines Schlüssels/Unterschlüssels ermöglich. Diese Funktion erwartet drei Parameter:

- **lngKey**: Einen der Werte der **Enum**-Auflistung **Key**, zum Beispiel **HKEY_CURRENT_USER**
- **strKeyName**: Den Unterschlüssel, zum Beispiel **SOFTWARE\Microsoft\Office\16.0\Access\Settings**
- **strValueName**: Den Namen des zu ermittelnden Elements, zum Beispiel **Form Template**

Die Funktion nutzt die API-Funktion **RegOpenKeyEx**, um den angegebenen Schlüssel zu öffnen. Dann liest sie mit **RegQueryValueEx** das gewünschte Element aus. Der Parameter **varValue** nimmt den gesuchten Wert auf.

Dieser wird in den folgenden Schritten noch untersucht und von nicht erwünschten Zeichen befreit. Schließlich wird dieser Wert zurückgegeben und der Registry-Schlüs-

```
Declare PtrSafe Function RegCloseKey Lib "advapi32.dll" (ByVal hKey As Long) As Long
Declare PtrSafe Function RegCreateKeyEx Lib "advapi32.dll" Alias "RegCreateKeyExA" (ByVal hKey As Long, _
    ByVal lpSubKey As String, ByVal Reserved As Long, ByVal lpClass As String, ByVal dwOptions As Long, _
    ByVal samDesired As Long, ByVal lpSecurityAttributes As Long, phkResult As Long, lpdwDisposition As Long) As Long
Declare PtrSafe Function RegOpenKeyEx Lib "advapi32.dll" Alias "RegOpenKeyExA" (ByVal hKey As Long, _
    ByVal lpSubKey As String, ByVal ulOptions As Long, ByVal samDesired As Long, phkResult As Long) As Long
Declare PtrSafe Function RegQueryValueExString Lib "advapi32.dll" Alias "RegQueryValueExA" (ByVal hKey As Long, _
    ByVal lpValueName As String, ByVal lpReserved As Long, lpType As Long, ByVal lpData As String, lpcbData As Long) As Long
Declare PtrSafe Function RegQueryValueExLong Lib "advapi32.dll" Alias "RegQueryValueExA" (ByVal hKey As Long, _
    ByVal lpValueName As String, ByVal lpReserved As Long, lpType As Long, lpData As Long, lpcbData As Long) As Long
Declare PtrSafe Function RegQueryValueExNULL Lib "advapi32.dll" Alias "RegQueryValueExA" (ByVal hKey As Long, _
    ByVal lpValueName As String, ByVal lpReserved As Long, lpType As Long, ByVal lpData As Long, lpcbData As Long) As Long
Declare PtrSafe Function RegSetValueExString Lib "advapi32.dll" Alias "RegSetValueExA" (ByVal hKey As Long, _
    ByVal lpValueName As String, ByVal Reserved As Long, ByVal dwType As Long, ByVal lpValue As String, _
    ByVal cbData As Long) As Long
Declare PtrSafe Function RegSetValueExLong Lib "advapi32.dll" Alias "RegSetValueExA" (ByVal hKey As Long, _
    ByVal lpValueName As String, ByVal Reserved As Long, ByVal dwType As Long, lpValue As Long, ByVal cbData As Long) _
    As Long
Declare PtrSafe Function RegDeleteKey& Lib "advapi32.dll" Alias "RegDeleteKeyA" (ByVal hKey As Long, _
    ByVal lpSubKey As String)
Declare PtrSafe Function RegDeleteValue& Lib "advapi32.dll" Alias "RegDeleteValueA" (ByVal hKey As Long, _
    ByVal lpValueName As String)
Declare PtrSafe Function RegOpenKey Lib "advapi32.dll" Alias "RegOpenKeyA" (ByVal hKey As Long, _
    ByVal lpSubKey As String, phkResult As Long) As Long
Declare PtrSafe Function RegEnumKeyEx Lib "advapi32.dll" Alias "RegEnumKeyExA" (ByVal hKey As Long, _
    ByVal dwIndex As Long, ByVal lpName As String, lpcbName As Long, ByVal lpReserved As Long, _
    ByVal lpClass As String, lpcbClass As Long, lpftLastWriteTime As Any) As Long
Declare PtrSafe Function RegEnumValue Lib "advapi32.dll" Alias "RegEnumValueA" (ByVal hKey As Long, _
    ByVal dwIndex As Long, ByVal lpValueName As String, lpcbValueName As Long, ByVal lpReserved As Long, _
    lpType As Long, lpData As Byte, lpcbData As Long) As Long
```

Listing 1: Die benötigten API-Funktionen

sel mit der API-Funktion **RegCloseKey** wieder geschlossen (siehe Listing 2)

Wrapperfunktion zum Auslesen aller Einträge eines Schlüssels

Die Wrapperfunktion **EnumKeyValues** ermittelt alle Einträge des mit den Parametern übergebenen Schlüssel/Unterschlüssel-Kombination.

Die Funktion erwartet diese beiden Parameter:

- **lngKey**: Einen der Werte der **Enum**-Auflistung **Key**, zum Beispiel **HKEY_CURRENT_USER**
- **strKeyName**: Den Unterschlüssel, zum Beispiel **SOFTWARE\Microsoft\Office\16.0\Access\Settings**

Die Funktion öffnet wieder mit **RegOpenKey** den Schlüssel. Dann startet sie eine **Do...Loop**-Schleife, die dann endet, wenn die Funktion **RegEnumValue** den Wert **0** zurückgibt. Diese erhält in der Schleife wiederum einen Index als zweiten Parameter, der mit jedem Schleifendurchlauf erhöht wird, sowie als dritten Parameter eine mit 255 Leerzeichen vorbelegte Zeichenkette. Diese wird, sofern der aktuelle Index aus lngIndex ein gültiges Ergebnis hergibt, mit dem Namen des Eintrags gefüllt. Der Inhalt von **strSave** wird dann mit der Funktion **StripTerminator**

```
Public Function QueryValue(lngKey As Key, strKeyName As String, strValueName As String)
    Dim lngRetVal As Long
    Dim hKey As Long
    Dim varValue As Variant
    lngRetVal = RegOpenKeyEx(lngKey, strKeyName, 0, KEY_ALL_ACCESS, hKey)
    lngRetVal = QueryValueEx(hKey, strValueName, varValue)
    varValue = Trim(varValue)
    If varValue <> "" Then
        If Asc(Right(varValue, 1)) > 127 Then
            varValue = Left(varValue, Len(varValue) - 1)
        End If
    End If
    If varValue <> "" Then
        If Asc(Right(varValue, 1)) < 21 Then
            varValue = Left(varValue, Len(varValue) - 1)
        End If
    End If
    QueryValue = varValue
    RegCloseKey hKey
End Function
```

Listing 2: Die benötigten API-Funktionen

von **vbNullchar**-Zeichen befreit und an das Array **strKeys** angehängt. Dieses gibt die Funktion schließlich als Ergebnis zurück:

```
Public Function EnumKeyValues(lngKey As Key, _
        strSubkey As String) As String()
    Dim hKey As Long
    Dim lngIndex As Long
    Dim strSave As String
    Dim strKeys() As String
    RegOpenKey lngKey, strSubkey, hKey
    Do
        strSave = String(255, 0)
        If RegEnumValue(hKey, lngIndex, strSave, 255, _
                0, ByVal 0&, ByVal 0&, ByVal 0&) <> 0 Then
            Exit Do
        Else
            ReDim Preserve strKeys(lngIndex)
            strKeys(lngIndex) = StripTerminator(strSave)
            lngIndex = lngIndex + 1
        End If
    Loop
    RegCloseKey hKey
    EnumKeyValues = strKeys
End Function
```

Die Hilfsfunktion StripTerminator

Die in der vorher vorgestellten Funktion verwendete Hilfsfunktion **StripTerminator** befreit die übergebene Zeichenkette von Vorkommen des Zeichens **vbNullChar**. Die API-Funktion **RegEnumValue** ersetzt in der **String**-Variablen **strSave** die 255 Leerzeichen durch das Ergebnis und füllt die übrigen Zeichen mit dem Zeichen **vbNullChar** auf. Diese wollen wir entfernen und verwenden dazu die Funktion **StripTerminator**. Die Funktion erwartet die zu untersuchende Zeichenkette als Parameter und gibt die überarbeitete Zeichenkette zurück. Sie sucht im ersten Schritt nach dem ersten Vorkommen von **vbNullChar** und geht davon aus, dass hier das eigentliche Ergebnis beendet ist. Die Suche erledigt sie mit der Funktion **InStr**, welche die Position des ersten Vorkommens zurückliefert. Ist das Ergebnis größer als **0**, wurde ein **vbNullChar**-

```
Public Function EnumRegKeys(lngKey As Key, strKeyName As String) As String()
    Dim hKey As Long
    Dim lngIndex As Long
    Dim strSave As String
    Dim strKeys() As String
    RegOpenKey lngKey, strKeyName, hKey
    Do
        strSave = String(255, 0)
        If RegEnumKeyEx(hKey, lngIndex, strSave, 255, 0, vbNullString, ByVal 0&, ByVal 0&) <> 0 Then
            ReDim Preserve strKeys(lngIndex)
            strKeys(lngIndex) = StripTerminator(strSave)
            lngIndex = lngIndex + 1
        Else
            Exit Do
        End If
    Loop
    RegCloseKey hKey
    EnumRegKeys = strKeys
End Function
```

Listing 3: Die Wrapperfunktion zum Auslesen aller untergeordneten Schlüssel

Zeichen gefunden. Dann schneidet die Funktion alle Zeichen von diesem Zeichen an ab und trägt nur die davor liegenden Zeichen in den Rückgabewert der Funktion ein. Andernfalls erhält dieser einfach die übergebene Zeichenkette als Ergebnis:

```
Private Function StripTerminator(strInput As String) _
        As String
    Dim intPosNull As Integer
    intPosNull = InStr(1, strInput, vbNullChar)
    If intPosNull > 0 Then
        StripTerminator = Left$(strInput, intPosNull - 1)
    Else
        StripTerminator = strInput
    End If
End Function
```

Wrapperfunktion zum Auslesen aller Unterschlüssel eines Schlüssels

Wenn Sie alle Schlüssel unterhalb eines anderen Schlüssels auslesen möchten, können Sie dazu die Wrapperfunktion **EnumRegKeys** nutzen (siehe Listing 3). Diese Funktion erwartet die gleichen Parameter wie die Funktion **EnumKeyValues**.

Sie öffnet mit der API-Funktion **RegOpenKey** den zu untersuchenden Schlüssel. Dann durchläuft sie eine **Do Loop**-Schleife solange, bis kein weiterer untergeordneter Schlüssel mehr gefunden werden kann.

Dabei füllt sie die Variable **strSave**, die mit dem jeweiligen Namen des Unterschlüssels gefüllt werden soll, mit 255 Leerzeichen. Dann prüft sie in einer **If...Then**-Bedingung das Ergebnis der API-Funktion **RegEnumKeyEx**.

Ist dieses **0**, wurde für den Index mit dem Wert aus **lngIndex** ein Unterschlüssel gefunden.

Dann vergrößert die Funktion im **If**-Teil der Bedingung das Array **strKeys** für die Ergebnisse auf den aktuellen Index aus **lngIndex** und fügt den Namen des mit **strSave** ermittelten Unterschlüssels an der neuen Position in das Array ein. Zuvor entfernt die Funktion **StripTerminator** wieder die überzähligen **vbNullChar**-Zeichen aus **strSave**.

Ribbon: Callback-Signaturen für VBA und VB6

Wer das Ribbon um benutzerdefinierte Erweiterungen ergänzen möchte, kommt früher oder später nicht um die Programmierung von Callbackfunktionen herum. Das sind Funktionen, die für Callback-Attribute von Ribbon-Elementen angegeben werden und die für verschiedene Aktionen aufgerufen werden – beispielsweise beim Anklicken einer Schaltfläche, beim Ändern des Inhalts eines Textfeldes oder schlicht, um vor dem Anzeigen dynamisch Einstellungen für Attribute des Ribbons einzulesen. Diese Callbackfunktionen haben eine bestimmte Signatur (sprich Definition der ersten Zeile). Da diese für den Einsatz von VBA und VB6 unterschiedlich aussehen und aktuell twinBASIC als Ersatz für VB6 heranreift, wollen wir eine Referenz der Callback-Signaturen für diese beiden Programmiersprachen anbieten.

Warum verschiedene Callback-Signaturen?

Als ich neulich angefangen habe, mit der neuen Programmiersprache twinBASIC COM-Add-Ins für die Benutzeroberfläche von Access zu programmieren, wollte ich natürlich auch Callbackfunktionen einsetzen.

Also habe ich einfach die gleichen Callbackfunktionen verwendet, die ich sonst unter Access und VBA einsetze. Der Aufruf lieferte allerdings immer wieder den gleichen Fehler – siehe Bild 1.

Ich bin fast verzweifelt, bis ich mich dann daran erinnerte, schon einmal mit dem alten Visual Studio unter VB6 ein COM-Add-In mit Ribbon programmiert zu haben – und dass dort die Signatur der Callbackfunktionen anders ausgesehen hat.

Unter VB6 sind Callbackfunktionen nämlich tatsächlich Funktionen – und nicht wie unter VBA Callbackprozeduren, die einen weiteren Parameter enthalten, der als Rückgabewert verwendet wird.

Schauen wir uns also die unterschiedlichen Signaturen für die einzelnen Steuerelemente des Ribbons für VBA und VB6 an!

Wichtige Informationen

Die wichtigste Information: Verwenden Sie alle Callback-Signaturen genau so, wie Sie hier abgebildet sind. Manchmal kann schon das Hinzufügen eines Datentyps zu einem Parameter, der hier keinen Datentyp aufweist, zu Problemen führen. Wichtig ist auch, auf die korrekte Verwendung von **Function/Sub** zu achten. Wenn Sie twinBASIC verwenden, können Sie das Ergebnis zum Zurückgeben entweder einer Variablen mit dem gleichen Namen wie die Funktion zuweisen, aber auch die **Return**-Anweisung nutzen:

```
Function GetText(control As IRibbonControl, ByRef text) _
                                                As String
    ...
    Return strText
End Function
```

Bei den folgenden Definitionen stellen wir zunächst die je nach Steuerelement individuellen Callbackfunktionen vor. Am Ende finden Sie eine Auflistung all jener Callbackfunk-

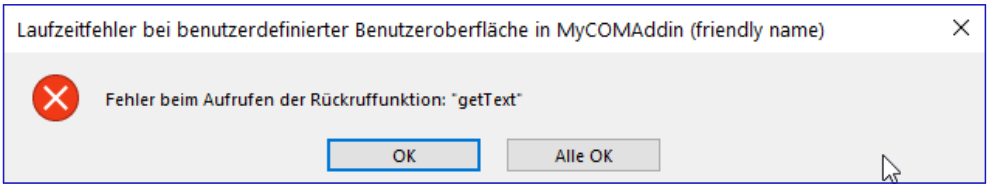


Bild 1: Fehler durch falsche Callback-Signatur

tionen, die von einer großen Anzahl an Steuerelementen verwendet wird.

Die Callbackfunktionen des button-Elements

Das **button**-Element bietet die folgenden Callbackfunktionen:

getShowImage: Fragt ab, ob ein Image angezeigt werden soll. Das geschieht unter VBA mit dem Rückgabe-Parameter **showImage**, unter VB6/twinBASIC mit dem Rückgabewert.

```
VBA: Sub GetShowImage (control As IRibbonControl, ByRef showImage)
VB6/twinBASIC: Function GetShowImage (control As IRibbonControl) As Boolean
```

getShowLabel: Fragt ab, ob eine Bezeichnung angezeigt werden soll. Der Wert **True** oder **False** wird mit dem Parameter **showLabel** oder dem Rückgabewert übergeben.

```
VBA: Sub GetShowLabel (control As IRibbonControl, ByRef showLabel)
VB6/twinBASIC: Function GetShowLabel (control As IRibbonControl) As Boolean
```

onAction: Wird beim Anklicken ausgelöst. Diese Signatur gilt für benutzerdefinierte **button**-Elemente.

```
VBA: Sub OnAction(control As IRibbonControl)
VB6/twinBASIC: Sub OnAction(control As IRibbonControl)
```

onAction, Alternative: Wird beim Anklicken eines eingebauten Elements ausgelöst, für das Sie ein **command**-Element definiert haben.

Wie Sie diese Variante von **onAction** einsetzen, lernen Sie im Beitrag **Funktion von eingebauten Ribbon-Steuerelementen überschreiben** (**www.access-im-unternehmen.de/1328**).

```
VBA: Sub OnAction(control As IRibbonControl, byRef CancelDefault)
VB6/twinBASIC: Sub OnAction(control As IRibbonControl, byRef CancelDefault)
```

Die Callbackfunktionen des checkBox-Elements

Das **checkBox**-Element bietet die folgenden Callbackfunktionen:

getPressed: Stellt den Zustand des **checkBox**-Steuerelements ein, also auf **True** oder **False**. Der Rückgabewert heißt unter VBA **returnValue**.

```
VBA: Sub GetPressed(control As IRibbonControl, ByRef returnValue)
VB6/twinBASIC: Function GetPressed(control As IRibbonControl) As Boolean
```

onAction: Wird ausgelöst, wenn der Benutzer auf ein **checkBox**-Steuerelement klickt und seinen Wert damit ändert. Liefert den neuen Wert mit dem **Boolean**-Parameter **pressed**.

```
VBA: Sub OnAction(control As IRibbonControl, pressed As Boolean)
VB6/twinBASIC: Sub OnAction(control As IRibbonControl, pressed As Boolean)
```

Die Callbackfunktionen des comboBox-Elements

Das **comboBox**-Element bietet die folgenden Callbackfunktionen:

getItemCount: Callbackfunktion zum Abfragen der Anzahl der im **comboBox**-Steuerelement anzuzeigenden Einträge. Zu liefern mit dem Parameter **count** oder dem Rückgabewert der Callbackfunktion.

```
VBA: Sub GetItemCount(control As IRibbonControl, ByRef count)
VB6/twinBASIC: Function GetItemCount(control As IRibbonControl) As Integer
```

getItemID: Ruft den Wert für die **ItemID** eines der Einträge des **comboBox**-Steuerelements ab, dessen Index unter VBA mit dem Parameter **index** abgefragt wird – oder mit dem Rückgabewert unter Visual Basic 6/ twinBASIC.

```
VBA: Sub GetItemID(control As IRibbonControl, index As Integer, ByRef id)
VB6/twinBASIC: Function GetItemID(control As IRibbonControl, index As Integer) As String
```

getItemImage: Ruft den Namen des Bildes eines der Einträge des **comboBox**-Steuerelements ab, dessen Index unter VBA mit dem Parameter **index** oder unter VB6/ twinBASIC mit dem Rückgabewert der Callbackfunktion abgefragt wird.

```
VBA: Sub GetItemImage(control As IRibbonControl, index As Integer, ByRef image)
VB6/twinBASIC: Function GetItemImage(control As IRibbonControl, index As Integer) As IPictureDisp
```

getItemLabel: Ruft die Beschriftung eines der Einträge des **comboBox**-Steuerelements ab, dessen Index mit dem Parameter **index** oder dem Rückgabewert übergeben wird.

```
VBA: Sub GetItemLabel(control As IRibbonControl, index As Integer, ByRef label)
VB6/twinBASIC: Function GetItemLabel(control As IRibbonControl, index As Integer) As String
```

getItemScreenTip: Ruft den **screenTip** eines der Einträge des **comboBox**-Steuerelements ab, dessen Index mit dem Parameter **index** oder dem Rückgabewert übergeben wird.

```
VBA: Sub GetItemScreenTip(control As IRibbonControl, index As Integer, ByRef screentip)
VB6/twinBASIC: Function GetItemScreentip(control As IRibbonControl, index As Integer) As String
```

getItemSuperTip: Ruft den **superTip** eines der Einträge des **comboBox**-Steuerelements ab, dessen Index mit dem Parameter **index** oder dem Rückgabewert übergeben wird.

```
VBA: Sub GetItemSuperTip (control As IRibbonControl, index As Integer, ByRef supertip)
VB6/twinBASIC: Function GetItemSuperTip (control As IRibbonControl, index As Integer) As String
```

getText: Ruft den aktuell im **comboBox**-Element anzuzeigenden Text ab. Dies geschieht über den Parameter **text** oder dem Rückgabewert.

```
VBA: Sub GetText(control As IRibbonControl, ByRef text)
VB6/twinBASIC: Function GetText(control As IRibbonControl) As String
```

onChange: Wird ausgelöst, wenn der Benutzer einen anderen Eintrag im **comboBox**-Steuerelement auswählt. Liefert den neuen Wert mit dem Parameter **text**.

```
VBA: Sub OnChange(control As IRibbonControl, text As String)
VB6/twinBASIC: Sub OnChange(control As IRibbonControl, text As String)
```

Die Callbackfunktionen des customUI-Elements

Das **customUI**-Element bietet die folgenden Callbackfunktionen:

loadImage: Wird beim Anzeigen eines Ribbons für jedes Steuerelement im Ribbon einmal ausgelöst, für das im Attribut **image** ein Bildname hinterlegt ist.

```
VBA: Sub LoadImage(imageId As string, ByRef image)
VB6/twinBASIC: Function LoadImage(imageId As String) As IPictureDisp
```

onLoad: Wird beim erstmaligen Laden des Ribbons ausgelöst und erlaubt das Referenzieren der mit **ribbon** gelieferten Instanz der Ribbon-Definition.

```
VBA: Sub OnLoad(ribbon As IRibbonUI)
VB6/twinBASIC: Function OnLoad(ribbon As IRibbonUI)
```

Die Callbackfunktionen des dropDown-Elements

Das **dropDown**-Element bietet die folgenden Callbackfunktionen:

getItemCount: Callbackfunktion zum Abfragen der Anzahl der im **dropDown**-Steuerelement anzuzeigenden Einträge, zu liefern mit dem Parameter **count** oder dem Rückgabewert als Integer.

```
VBA: Sub GetItemCount(control As IRibbonControl, ByRef
count)
VB6/twinBASIC: Function GetItemCount(control As IRibbon-
Control) As Integer
```

getItemID: Ruft den Wert für die **ItemID** eines der Einträge des **dropDown**-Steuerelements ab, dessen Index mit dem Parameter **index** übergeben wird. Die **ItemID** wird mit dem Parameter **id** oder dem Rückgabewert im Format **String** erwartet.

```
VBA: Sub GetItemID(control As IRibbonControl, index As
Integer, ByRef id)
VB6/twinBASIC: Function GetItemID(control As IRibbonCon-
trol, index As Integer) As String
```

getItemImage: Ruft den Namen des Bildes eines der Einträge des **dropBox**-Steuerelements ab, dessen Index mit dem Parameter **index** geliefert wird. Der Bildname wird mit dem Parameter image oder mit dem Rückgabewert des Datentyps **IPictureDisp** übergeben.

```
VBA: Sub GetItemImage(control As IRibbonControl, index As
Integer, ByRef image)
VB6/twinBASIC: Function GetItemImage(control As IRibbon-
Control, index As Integer) As IPictureDisp
```

getItemLabel: Ruft die Beschriftung eines der Einträge des **dropDown**-Steuerelements ab, dessen Index mit dem Parameter **index** übergeben wird und die Beschriftung mit dem Parameter **label** beziehungsweise mit dem Rückgabewert als **String** erwartet.

```
VBA: Sub GetItemLabel(control As IRibbonControl, index As
Integer, ByRef label)
VB6/twinBASIC: Function GetItemLabel(control As IRibbon-
Control, index As Integer) As String
```

getItemScreenTip: Ruft den **screenTip** eines der Einträge des **getItemScreenTip**-Steuerelements ab, dessen Index mit dem Parameter **index** geliefert und dessen Wert mit dem Parameter **screenTip** oder dem Rückgabewert übergeben wird.

```
VBA: Sub GetItemScreenTip(control As IRibbonControl, index
As Integer, ByRef screenTip)
VB6/twinBASIC: Function GetItemScreentip(control As IRib-
bonControl, index As Integer) As String
```

getItemSuperTip: Ruft den **superTip** eines der Einträge des **dropDown**-Steuerelements ab, dessen Index mit dem Parameter **index** übergeben wird und dessen Wert mit dem Parameter **superTip** oder mit dem **String**-Rückgabewert zurückgegeben wird.

```
VBA: Sub GetItemSuperTip (control As IRibbonControl, index
As Integer, ByRef superTip)
VB6/twinBASIC: Function GetItemSuperTip (control As IRib-
bonControl, index As Integer) As String
```

getSelectedItemID: Fragt den Wert des Attributs **itemID** für das aktuell zu selektierende Element für ein **dropDown**-Steuerelement ab – entweder über den Parameter **index** oder den Rückgabewert der Callbackfunktion im **Integer**-Format.

```
VBA: Sub GetSelectedItemID(control As IRibbonControl,
ByRef index)
VB6/twinBASIC: Function GetSelectedItemID(control As IRib-
bonControl) As Integer
```

Benutzerdefinierte Bilder in twinBASIC

Die Programmiersprache/Entwicklungsumgebung twinBASIC entwickelt sich aktuell stetig weiter. Der Entwickler Wayne Philips fügt ständig neue Elemente hinzu. In den ersten Beiträgen mussten wir das Ribbon eines COM-Add-Ins noch mit den eingebauten Icons ausstatten, da es nicht möglich war, Bilddateien als Resourcen in die DLL zu integrieren. Das hat sich nun geändert: Sie können Bilddateien zum Projekt hinzufügen und diese sehr einfach im Ribbon nutzen. Dieser Beitrag zeigt, wie das gelingt.

Voraussetzungen

Wie Sie ein COM-Add-In erstellen, erfahren Sie beispielsweise in den beiden Beiträgen **twinBASIC – COM-Add-Ins für Access** (**www.access-im-unternehmen.de/1306**) und **Access-Optionen per Ribbon ändern** (**www.access-im-unternehmen.de/1327**).

Icons nutzen

Icons verwenden wir bei COM-Add-Ins aktuell für Schaltflächen und anderen Elemente im Ribbon. Um Icons zu nutzen, sind zwei Schritte nötig:

- Hinzufügen zum Projekt
- Laden, wenn das Ribbon angezeigt wird.

Beides schauen wir uns nun an.

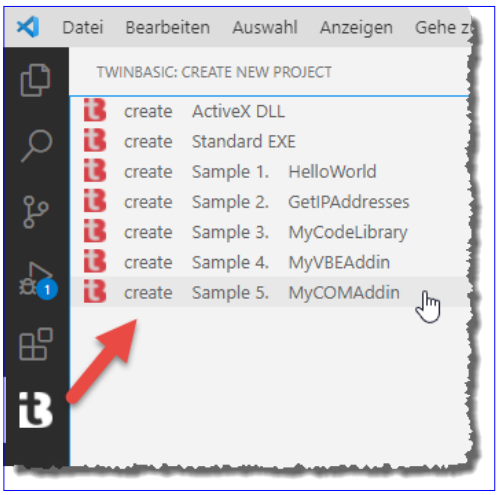


Bild 1: Erstellen eines COM-Add-In-Projekts

Icons zum Projekt hinzufügen

Das Beispielprojekt für ein COM-Add-In, dass Sie mit dem Befehl aus Bild 1 erstellen, enthält bereits zwei Ordner, die Sie für das Hinzufügen von Bildern benötigt werden.

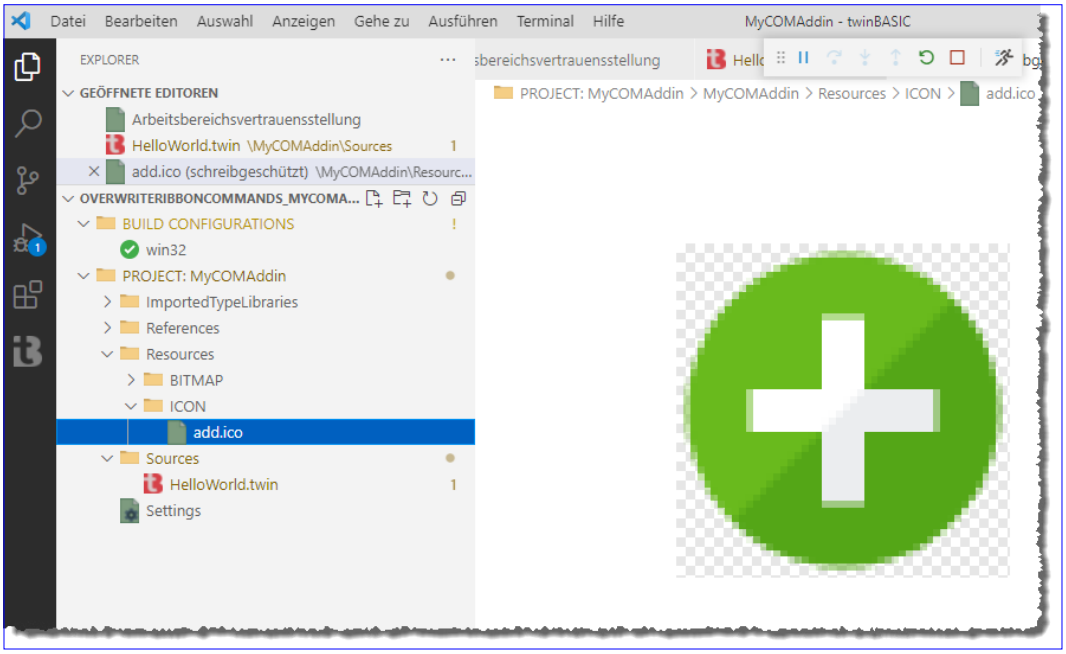


Bild 2: Hinzufügen eines Icons zum Projekt

Ziehen Sie **.ico**-Dateien in den **ICON**-Ordner, sodass diese dort wie in Bild 2 erscheinen. Achtung: Wenn Drag and Drop nicht funktioniert, kann es sein, dass

Setup für Access: Umsetzung mit InnoSetup

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

Im ersten Teil haben wir uns mit den Grundlagen eines Setups beschäftigt, nun geht es »in medias res«. Dieser Teil beinhaltet die konkrete Umsetzung der Gedanken. Wir schauen uns das Setup-Script im Detail an und lernen, was die einzelnen Bestandteile bedeuten. Wir werden neben der Access-Datenbank auch Startmenü-Einträge und Desktop-Icons anlegen. Die Setup-Sprache wird variabel gemacht, es darf eine Lizenzvereinbarung geben, und es wird für eine ordnungsgemäße Deinstallation gesorgt.

Organisatorisches

Vor der Umsetzung sollten wir uns kurz überlegen, was wir erreichen wollen, denn »wer nicht weiß, wo er hin will, darf sich nicht wundern, wenn er ganz woanders ankommt«.

Gehen wir zunächst von einem Minimalsetup aus. Was wäre das Minimum, das ein Access-Entwickler von »seinem« Setup erwartet?

- Einspielen der **.accdb**/**.accde**-Datei
- Einrichten von Startmenü- und Desktop-Icons
- Vorbereitung der Deinstallation

Als kleine Erweiterung behalten wir uns dann noch folgendes vor:

- Zu Beginn Darstellung von Betahinweis oder der Lizenzbedingungen, Bestätigung anfordern
- Zu Beginn Darstellung im Stile “Was ist neu in diesem Release?”
- Abschließend eine Readme-Datei anzeigen

Das ist zunächst sicher nicht viel, und wer den ersten Teil dieser kleinen Artikelserie gelesen hat, hätte sicher noch einiges mehr erwartet. Aber machen wir es uns zu Beginn nicht zu schwer, Erweiterungen sind ja jederzeit möglich.

Ich erlaube mir, an dieser Stelle noch ein anderes hilfreiches Werkzeug des Softwareentwicklers zu erwähnen. Wahrscheinlich werden Sie dies ohnehin nutzen, dann ist es nur eine Erinnerung, dass das auch für unser Setup-Script funktioniert.

Es geht um Versionsverwaltung, auch Quellcodeverwaltung genannt. Unser Setup-Script ist im »Quellcode« schließlich auch nur ein wenig »plain text«, sodass dieses Script genauso mit Git (oder oder einer anderen Quellcodeverwaltung) verwaltet werden kann – und sollte.

Die **.exe**-Datei würde ich dabei von der Verwaltung ausnehmen (Git: ***.exe** in die **.gitignore**-Datei). Einerseits wird sie recht groß sein, andererseits ist sie redundant, denn alles, was zu ihrer Erstellung benötigt wird, ist ja ohnehin komplett versioniert.

Grundprinzip des Setup-Scripts

Das Setup-Script ähnelt in weiten Bereichen streng genommen mehr einer **.ini**-Datei als einem Programm. Ich will es dennoch weiterhin »Script« nennen, da der Begriff durchaus als gängig gelten kann. Die Festlegungen in diesem Script werden vom InnoSetup-Compiler inter-

pretiert, der letztlich dann das eigentliche Setup (eine **.exe**-Datei) zusammenbaut. Ausgeliefert wird dann nur diese **.exe**-Datei.

Wie bei einer **.ini**-Datei üblich haben wir Sektionen mit Überschriften in eckigen Klammern wie zum Beispiel **[Setup]** gleich zu Beginn. Innerhalb dieser Sektionen stehen dann zahlreiche Key-Value-Einträge im Format **Key = Value**.

Beginnt eine Zeile mit einem Semikolon, wird diese komplett als Kommentarzeile interpretiert, also nicht für die Setup-Erstellung berücksichtigt. Dadurch können wir bestimmte Einträge vorbereiten und bei Bedarf aktivieren oder deaktivieren. Nur ein erneuter Compilerlauf ist dann erforderlich.

Wer ein neues Setup erstellen will, kann dies sowohl über InnoSetup selbst als auch über eine der im ersten Teil bereits angesprochenen GUIs tun. Dort ist in der Regel ein Assistent enthalten, mit dessen Hilfe das Grundgerüst schnell eingerichtet werden kann.

Da die Bedienung solcher Assistenten zumeist sehr intuitiv ist, will ich diese hier nicht näher besprechen, sondern lieber auf die Eintragungen im Script selbst eingehen. Ob Sie das Script mittels Assistenten oder von Hand erstellen oder einfach das am Ende dieses Artikels bereitgestellte fertige Script verwenden, bleibt dabei Ihnen überlassen.

Zu Beginn definieren wir in der Sektion **[Setup]** den Namen des zu installierenden Programms und einige weitere organisatorische Dinge:

```
[Setup]
AppId=TestSetup
AppName=My Awesome AccessApp
AppVersion=1.0.0
AppVerName=AccessApp v1.0.0
AppPublisher=Vorname Name
AppPublisherURL=https://www.meine-homepage.de
AppSupportURL=https://www.meine-homepage.de
AppUpdatesURL=https://www.meine-homepage.de
```

Dadurch werden Name und Version der zu installierenden Software genannt, ebenso Name und Website des Herausgebers und zwei URLs für Support und Updates.

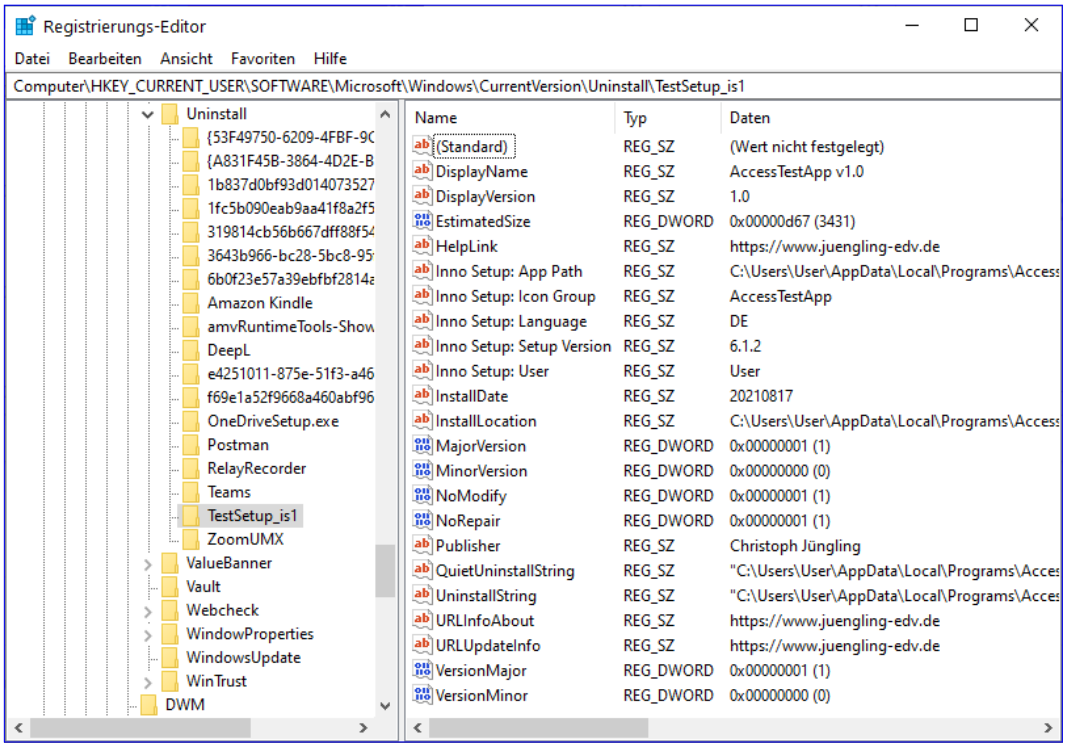


Bild 1: Setup-Eintrag in der Registry

Die **AppId** hat dabei eine Sonderstellung, denn diese Information wird in der Registry als Bezeichnung für den **Uninstall**-Bereich eingetragen (siehe Bild 1). Hier finden Sie detaillierte Informationen über die Deinstallation.

Anhand dieser Bezeichnung erkennt InnoSetup, ob es sich um eine Erstinstallation oder ein Update handelt. Dementsprechend sollte dieser Eintrag im Laufe des Projektes nicht mehr verändert werden!

Anstelle eines sprechenden Namens kann hier natürlich auch eine GUID verwendet werden. Die Compiler-GUI und auch Inno Script Studio besitzen zu deren Erzeugung einen Menübefehl.

Hierbei sehen wir bereits, dass einige Informationen mehrfach eingetragen werden. Zur besseren Übersicht bietet sich daher ein Verfahren an, das wir bereits vom Programmieren her kennen.

Wann immer wir eine immer gleiche Information an mehreren Stellen des Programms verwenden müssen, definieren wir eine globale Konstante und verwenden in der Folge dann nur noch diese anstelle des tatsächlichen Wertes. So ist eine Änderung des Wertes schnell gemacht und die Semantik ist auch sofort klar. Das geht natürlich auch in InnoSetup.

Konstanten im Setup-Script

Die Definition der Konstanten erfolgt dabei zwingend am Anfang des Script vor der **[Setup]**-Sektion nach folgendem Muster:

```
#define MyAppName "My Awesome AccessApp"
#define MyAppVersion "1.0.0"
#define MyAppPublisher "Christoph Jüngling"
#define MyAppURL "https://www.meine-homepage.de"
#define MyAppExeName "Testdatenbank.accdb"
```

Nun können wir das obige Script entsprechend überarbeiten. Dabei werden die Konstanten nach dem Muster **{#NameDerKonstante}** an der Stelle eingefügt, wo deren Wert stehen soll.

```
[Setup]
AppId={#MyAppName}
AppName={#MyAppName}
AppVersion={#MyAppVersion}
AppVerName={#MyAppName} v{#MyAppVersion}
AppPublisher={#MyAppPublisher}
AppPublisherURL={#MyAppURL}
AppSupportURL={#MyAppURL}
AppUpdatesURL={#MyAppURL}
```

Das obige sind natürlich nur Vorschläge. Nach dem nun bekannten Muster lassen sich leicht weitere Konstanten hinzufügen, zum Beispiel um die drei URLs auch wirklich unterschiedlich zu gestalten.

Mindestens eine weitere Information müssen wir noch angeben, und zwar den Ort, an dem die Installation erfolgen soll:

```
DefaultDirName={userpf}\{#MyAppName}
```

Diese heißt **DefaultDirName**, da das Verzeichnis während der Installation durch den User noch geändert werden kann. Die vordefinierte Konstante **{userpf}** wird von InnoSetup erst bei der Installation ausgewertet.

Es handelt sich um das benutzerspezifische Programm-Verzeichnis **C:\Users\USERNAME\AppData\Local\Programs** (siehe auch im Beitrag **Setup für Access-Anwendungen**, **www.access-im-unternehmen.de/1316** unter **Wohin mit dem Frontend?**).

In diesem Zusammenhang wird auch noch eine weitere Frage wichtig: Welche Rechte benötigt der User, der dieses Setup ausführen will?

Bei »normalen« Setups ist es üblich und auch notwendig, dass man Admin-Berechtigungen hat. Wenn wir

aber unsere Applikation wirklich nur im userspezifischen Teil der Festplatte installieren wollen, ist das nicht notwendig, dann genügen die Rechte des Users völlig. Das teilen wir dem Setup-Script natürlich ebenfalls mit:

```
PrivilegesRequired=lowest
```

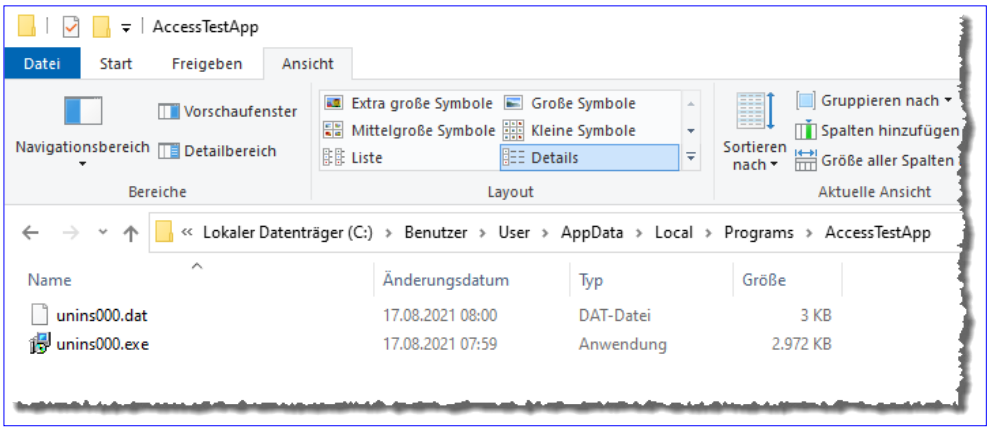


Bild 2: Weitgehend leeres Installationsverzeichnis

Nun können Sie das Setup bereits compilieren, obwohl im Moment natürlich noch gar nichts installiert wird. Zu diesem Zweck drücken Sie **F9** in InnoSetup oder Inno Script Studio, lassen das Setup durchlaufen und schauen dann in das angegebene Verzeichnis. Dort finden wir im Moment nur zwei Dateien (siehe Bild 2).

Es handelt sich um den Uninstaller und die von ihm benötigte Daten-Datei. Zugleich finden wir in der Systemsteuerung unter **Programme hinzufügen oder entfernen** einen passenden Eintrag zu unserer Installation (siehe Bild 3).

Den Button **Deinstallieren** sollten Sie zum Kennenlernen ruhig betätigen, und vergewissern Sie sich bitte anschließend, dass das Installationsverzeichnis wieder vollständig verschwunden ist.

Nun haben wir herausgefunden, dass das Setup grundsätzlich funktioniert. Jetzt wollen wir die Access-Datenbank hinzufügen.

Was und wohin installieren?

Für diese Frage ist die Sektion **[Files]** zuständig. Hier werden alle Dateien aufgeführt, die in das Setup einge-

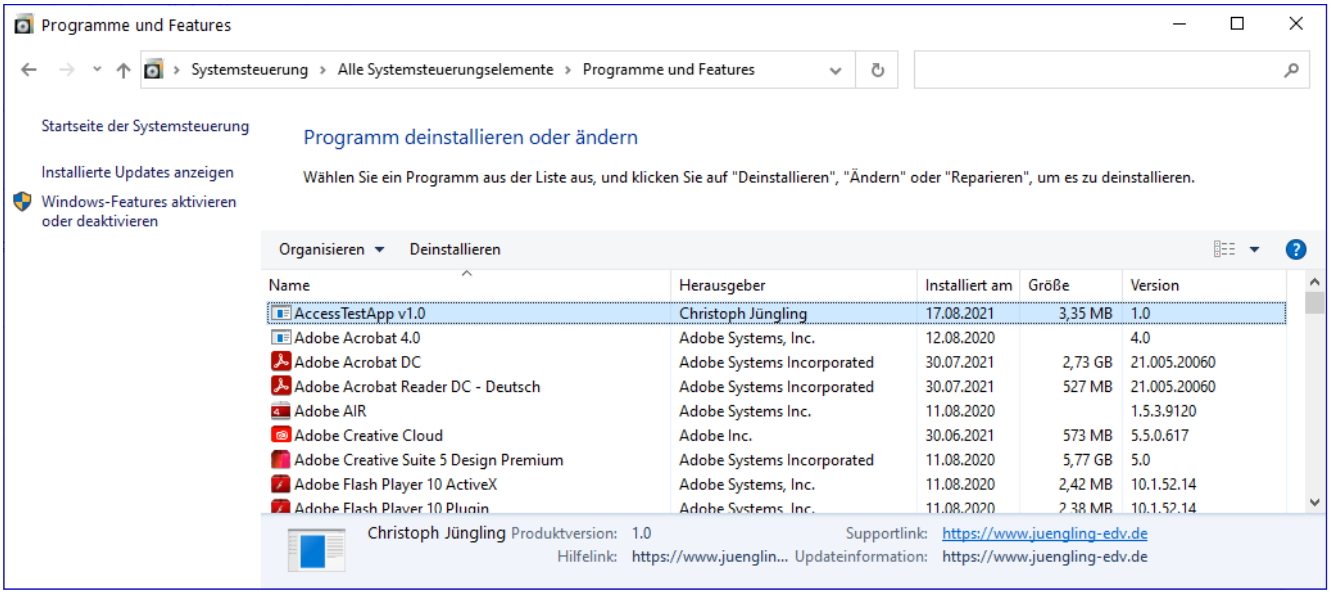


Bild 3: Eintrag in der Systemsteuerung

Access-Optionen per Ribbon ändern

Es gibt einige Access-Optionen, die man immer wieder nutzt. In meinem Fall ist es zum Beispiel die Einstellung, ob Formulare nun als überlappende Fenster oder als Dokumente im Registerkartenformat angezeigt werden sollen. Sicher haben Sie ähnliche Einstellungen, die Sie oft ändern oder die Sie vielleicht einfach nur schnell einsehen können – was bei dem mittlerweile recht umfangreich gewordenen Optionen-Dialog schon einige Sekunden kosten kann. Warum also nicht ein COM-Add-In bauen, dass die Informationen der wichtisten Access-Einstellungen immer direkt im Ribbon anzeigt – anstatt irgendwo versteckt im Optionen-Dialog? Und da wir mit twinBASIC auch noch ein praktisches Tool zum Erstellen von COM-Add-Ins zur Hand haben, können wir direkt loslegen!

Voraussetzung: Visual Studio Code und twinBASIC

Wenn Sie dauerhafte Ergänzungen oder Änderungen am Ribbon vornehmen wollen, benötigen Sie ein COM-Add-In. Eine andere Möglichkeit gibt es nur, wenn Sie mit den eingebauten Funktionen zum Anpassen der Benutzeroberfläche arbeiten wollen – und die ist bei Weitem nicht ausreichend.

Ein COM-Add-In konnten Sie früher mit Visual Studio 6 bauen, heutzutage mit Visual Studio .NET. Das ist allerdings nicht besonders komfortabel und erfordert umfangreichere Softwarevoraussetzungen als Visual Studio 6 gebaute COM-Add-Ins. Seit kurzem gibt es allerdings eine Erweiterung namens twinBASIC für Visual Studio Code, mit der Sie zum Beispiel COM-Add-Ins für Access und den VBA-Editor programmieren können. Wie das geht, haben wir bereits in den Beiträgen **twinBASIC – VB/VBA mit moderner Umgebung** (**www.access-im-unternehmen.de/1303**), **twinBASIC – COM-Add-Ins für Access** (**www.access-im-unternehmen.de/1306**) und anderen erläutert.

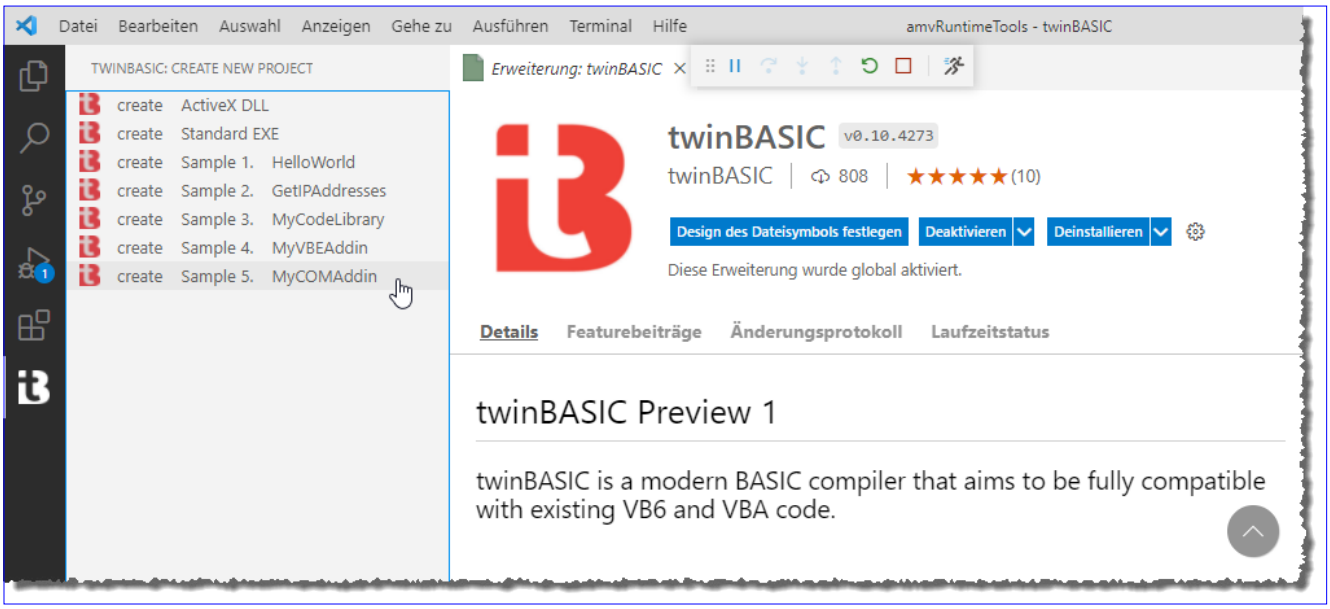


Bild 1: Erstellen eines neuen COM-Add-Ins

Im erstgenannten Beitrag erfahren Sie alles über die Installation, in letzterem die Grundlagen für das Erstellen von COM-Add-Ins. Darauf setzt der genannte Beitrag auf, Details zu diesen Themen finden Sie in den genannten Beiträgen.

Neues COM-Add-In anlegen

Um ein neues COM-Add-In anzulegen, klicken Sie in Visual Studio Code links auf das twinBASIC-Symbol. Es erscheint eine Liste mit Projektvorlagen, wo Sie die Vorlage **Sample 5 MyCOMAddin** auswählen (siehe Bild 1).

Anschließend erscheint ein Dialog, in dem Sie die anzulegende Datei samt Verzeichnis auswählen. Hier geben wir **amvAccessOptionsGoRibbon.twinproj** an. Dies fügt einige Dateien zum gewählten Verzeichnis hinzu, von denen die folgenden für uns interessant sind:

- **amvAccessOptionsGoRibbon_ACCESS_RegisterAddin32.reg**: Datei zum Hinzufügen der Registry-Einträge für das COM-Add-In
- **amvAccessOptionsGoRibbon_ACCESS_UnregisterAddin32.reg**: Datei zum Entfernen der Registry-Einträge für das COM-Add-In
- **amvAccessOptionsGoRibbon_myCOMAddin.code-workspace**: Arbeitsumgebung
- **amvAccessOptionsGoRibbon_myCOMAddin.twinproj**: Projektdatei

Das Projekt wird allerdings auch direkt in Visual Studio Code angezeigt.

Was bietet die Vorlage?

Mit der Vorlage erhalten Sie bereits fast alles, was Sie benötigen:

- Die Implementierung der Schnittstelle **IDTExtensibility2**, die einige Ereignisprozeduren bereitstellt, die zu verschiedenen Zeitpunkten beim Verwenden des COM-Add-Ins ausgelöst werden – zum Beispiel beim Start (**OnConnection**)
- Die Implementierung der Schnittstelle **IRibbonExtensibility** in Form der Ereignisprozedur **GetCustomUI**. Diese wird beim Start des COM-Add-Ins ausgelöst und stellt das Ribbon zusammen, über das die Benutzeroberfläche beziehungsweise die Funktionen des COM-Add-Ins bereitgestellt werden sollen.
- Eine von dieser Ereignisprozedur aufgerufene Callbackfunktion, die zeigt, wie Sie Funktionen über das Ribbon aufrufen können.

COM-Add-In zum Laufen bringen

Dementsprechend brauchen Sie nur zwei Schritte zu erledigen, damit das COM-Add-In läuft:

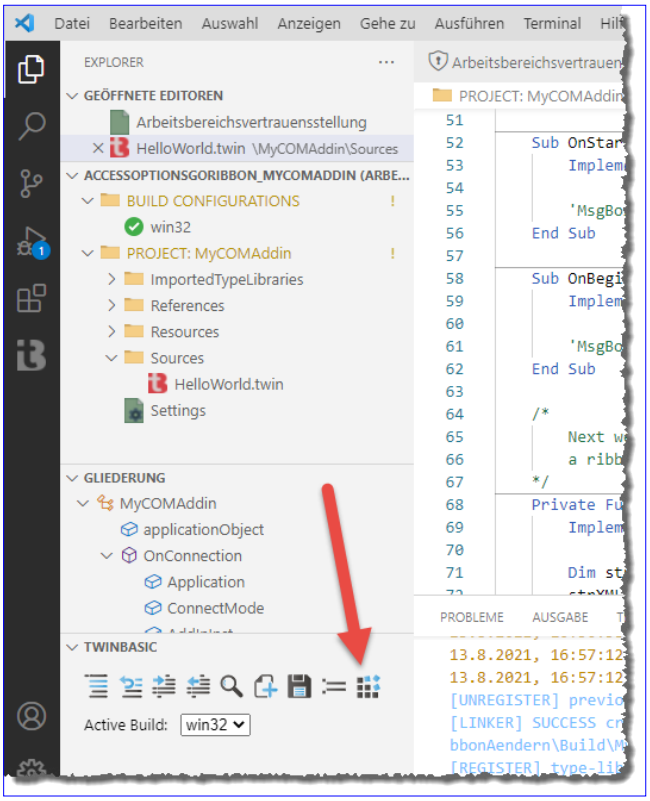


Bild 2: COM-Add-In kompilieren

- Das COM-Add-In kompilieren, indem Sie auf die Schaltfläche Build klicken (siehe Bild 2).
- Die Registrierungsdatei **amvAccessOptionsGoRibbon_ACCESS_RegisterAddin32.reg** starten, indem Sie diese doppelt anklicken und die folgenden Meldungen bestätigen.

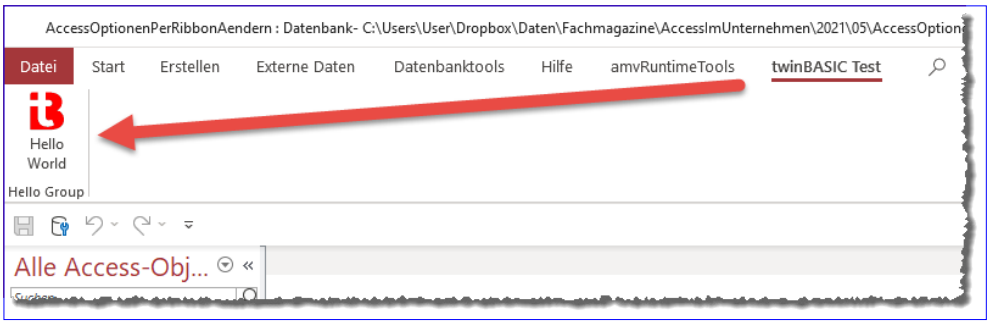


Bild 3: Das COM-Add-In ist nach wenigen Mausklicks einsatzbereit

Wenn Sie nun eine Access-Datenbank öffnen, erscheint ein neuer Eintrag namens **twinBASIC Test** im Ribbon mit einer eigenen Schaltfläche (siehe Bild 3).

Im Gegensatz zu dem COM-Add-In, das wir wie im Beitrag **twinBASIC – COM-Add-Ins für Access** erstellt haben, kann twinBASIC nun auch benutzerdefinierte Icons anzeigen. Wie Sie diese hinzufügen und über entsprechende Callbackfunktionen aufrufen, zeigen wir im Beitrag **Benutzerdefinierte Bilder in twinBASIC** (**www.access-im-unternehmen.de/1325**).

COM-Add-In vorbereiten

Da wir wissen, dass wir das COM-Add-In mit Access einsetzen wollen, können wir eine Objektvariable speziell zum Aufnehmen der aktuellen Access-Instanz deklarieren:

```
Private objAccess As Access.Application
```

Außerdem weisen wir diese in der Prozedur **OnConnection** wie folgt gezielt zu:

```
Private Function GetCustomUI(ByVal RibbonID As String) As String _
    Implements IRibbonExtensibility.GetCustomUI
    Dim strXML As String
    strXML &= "<?xml version="1.0"?>" & vbCrLf
    strXML &= "<customUI xmlns="http://schemas.microsoft.com/office/2009/07/customui">" & vbCrLf
    strXML &= "  <ribbon>" & vbCrLf
    strXML &= "    <tabs>" & vbCrLf
    strXML &= "      <tab id="tabOptions" label="Optionen">" & vbCrLf
    strXML &= "        <group id="grpThisDatabase" label="&lt;Aktuelle Datenbank&gt;">" & vbCrLf
    strXML &= "          <editBox label="Datenbanktitel:"id="txtTitle" maxLength="255" " _
           & " sizeString="xxxxxxxxxxxxxxxxxxxxxxxxx" getText="getText" onChange="onChange"/>" & vbCrLf
    strXML &= "        </group>" & vbCrLf
    strXML &= "      </tab>" & vbCrLf
    strXML &= "    </tabs>" & vbCrLf
    strXML &= "  </ribbon>" & vbCrLf
    strXML &= "</customUI>" & vbCrLf
    Return strXML
End Function
```

Listing 1: Einlesen des Ribbons mit dem Textfeld zum Festlegen des Datenbanktitels

```
Sub OnConnection(ByVal Application As Object, _
        ByVal ConnectMode As ext_ConnectMode, _
        ByVal AddInInst As Object, _
        ByRef custom As Variant()) _
    Implements IDTExtensibility2.OnConnection
    Set objAccess = Application
End Sub
```

Hier tragen wir den mit dem Parameter **Application** übergebenen Verweis auf die aktuelle Access-Instanz in die Variable **objAccess** ein.

Optionen mit dem Ribbon einstellen

Um Optionen mit dem Ribbon einzustellen, benötigen wir mehr als nur ein paar Schaltflächen. Immerhin wollen wir ja vor dem Ändern einer Option auch wissen, wie ihr aktueller Wert lautet! Also benötigen wir Steuerelemente wie Kontrollkästchen, Textfelder et cetera.

Da wir diese dynamisch einlesen und auch anpassen wollen, stellen wir ihre Werte zu Beginn mit entsprechenden Callback-Funktionen ein. Später ändern wir diese dann über das Ribbon und sorgen dafür, dass die Änderungen sich auch in den entsprechenden Access-Optionen niederschlagen.

Wir beginnen mit einer ersten Option, um uns den Ablauf anzusehen. Dies soll der Titel der Access-Anwendung sein. Der Vorteil gegenüber einigen anderen Optionen ist, dass wir Änderungen direkt in der Titelleiste angezeigt bekommen – im Gegensatz zu vielen anderen Optionen, die sich erst beim erneuten Öffnen der Datenbankanwendung zeigen.

Also fügen wir der Ribbon-Definition ein Textfeld hinzu. Danach sieht Prozedur **GetCustomUI** zur Definition des Ribbons wie in Listing 1 aus. Hier finden wir statt der bisher vorhandenen Schaltfläche ein **editBox**-Element. Für dieses haben wir die Eigenschaft **label** auf **Datenbanktitel** und **maxLength** auf **255** eingestellt. Letztere sorgt dafür, dass der Titel die maximale Zeichenzahl von **255** nicht überschreitet. Damit die **editBox**-Steuerelemente breite genug dargestellt werden, haben außerdem das Attribut **sizeString** auf den Wert **xxxxxxxxxxxxxxxxxxxxxxxx** eingestellt. Außerdem haben wir gleich zwei Callbackattribute definiert:

- **getText**: Gibt eine Callbackfunktion an, die beim erstmalige Anzeigen oder der Anforderung der Aktualisierung der Anzeige ausgelöst werden soll. Hier wollen wir den aktuellen Titel aus den Access-Optionen einlesen und anschließend im **editBox**-Steuerelement anzeigen.

- **onChange**: Gibt eine Callbackfunktion an, die beim Ändern des Wertes des **editBox**-Steuerelements ausgelöst wird. Hiermit wollen wir nach Änderungen durch den Benutzer den neuen Wert in die Access-Optionen übertragen.

Einlesen des aktuellen Datenbanktitels

Die Callbackfunktion **getText** wird ausgelöst, wenn das Ribbon geladen oder wenn per Code eine der Methoden **Invalidate** oder **InvalidateControl** der in einer Variablen gespeicherten Ribbon-Instanz ausgelöst wird und das Steuerelement danach erstmals sichtbar wird.

Sie erhält als ersten Parameter einen Verweis auf das aufrufende Steuerelement. Der zweite Parameter erwartet die im **editBox**-Steuerelement anzuzeigende Zeichenkette.

Wenn Sie die Prozedur wie nachfolgend in das twinBASIC-Projekt einfügen, werden einige Elemente rot unterstrichen. Das liegt daran, dass wir noch keinen Verweis auf die DAO-Bibliothek hinzugefügt haben:

```
Function GetText(control As IRibbonControl) As String
    Dim db As DAO.Database
    Dim prp As DAO.Property
    Dim strText As String
    On Error Resume Next
    Set db = objAccess.CurrentDb
    Set prp = db.Properties("AppTitle")
```

```
    On Error GoTo 0
    If prp Is Nothing Then
        strText = "<Kein
Titel>"
    Else
        strText = prp.Value
    End If
    On Error GoTo 0
    GetText = strText
End Function
```

Das holen wir nach, indem wir in twinBASIC zum Bereich **Settings** wechseln und dort unter **COM Type Library / Active X References** einen Verweis auf die Bibliothek **Microsoft Office 16.0 Access Database Engine Object Library [v12.0]** hinzufügen (siehe Bild 4). Danach unbedingt mit **Strg + S** speichern, damit die Änderungen wirksam werden!

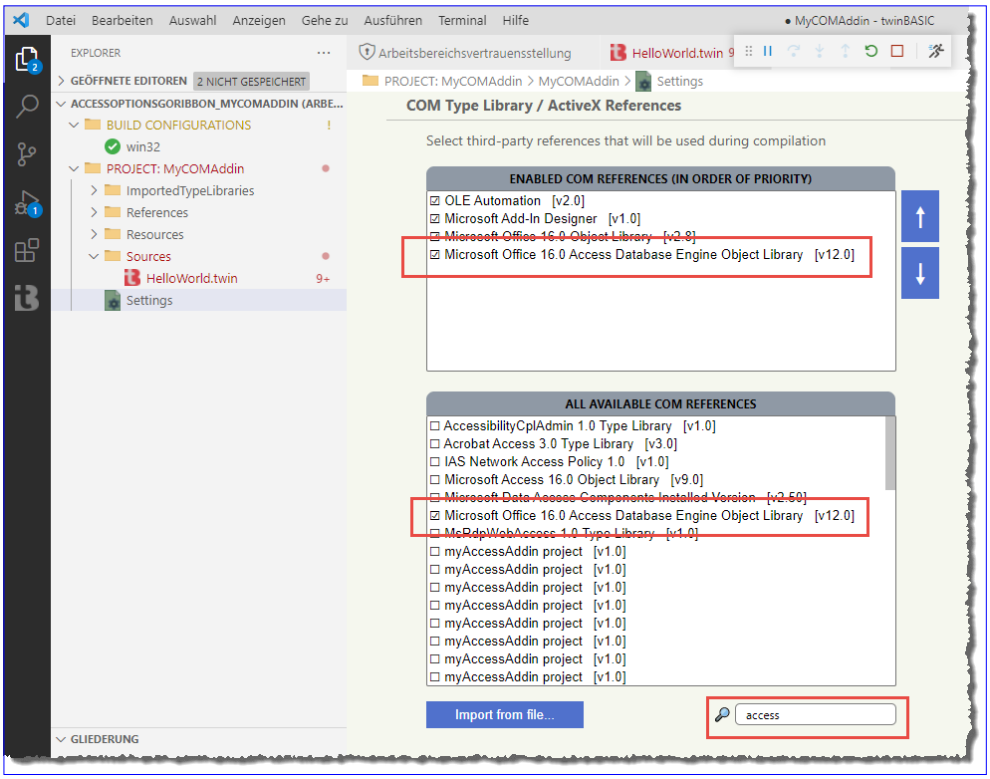


Bild 4: Hinzufügen eines Verweises auf die DAO-Bibliothek

Danach sind fast alle roten Unterstreichungen verschwunden, nur eine nicht – die unter **CurrentDb**. **CurrentDb** ist auch keine Eigenschaft der DAO-Bibliothek, sondern der Access-Bibliothek.

Also fügen wir auch diese noch mit dem Eintrag **Microsoft Access 16.0 Object Library [v9.0]** hinzu. Nach dem erneuten Speichern können wir uns nun den Code der Prozedur **getText** ohne rote Unterstreichungen ansehen.

Die Prozedur deaktiviert zunächst die eingebaute Fehlerbehandlung und ruft den Wert der Eigenschaft **AppTitle** des **Database**-Objekts der aktuellen Datenbank ab. Die Fehlerbehandlung deaktivieren wir dabei, weil ein Fehler ausgelöst wird, wenn diese Eigenschaft noch nicht oder nicht mehr vorhanden ist. Sie ist nur vorhanden, wenn diese bereits durch den Benutzer über den Optionen-Dialog von Access gesetzt wurde (oder per Code, wie wir gleich demonstrieren).

Danach aktivieren wir die Fehlerbehandlung wieder und prüfen per **If...Then**-Bedingung, ob **prp** eine Eigenschaft zugewiesen wurde.

Das ist nur der Fall, wenn die Eigenschaft aktuell vorhanden ist. Ist **prp** leer, übergeben wir die Zeichenkette **<Kein Titel>** als anzuzeigenden Text, enthält **prp** einen Wert, übergeben wir diesen an den Parameter **text**.

Andere Callback-Signatur in twinBASIC-Ribbons

Wie schon im Beitrag **twinBASIC – COM-Add-Ins für Access** festgestellt, sehen wir auch hier wieder eine andere Signatur für die Callbackfunktion. Sie lautet hier:

```
Function GetText(control As IRibbonControl) As String
```

Unter VBA würde diese lauten:

```
Sub getText(control As IRibbonControl,
ByRef text)
```

Die hier verwendete Signatur ist die gleiche, die wir auch in einem VB6-COM-Add-In nutzen würden. Da twinBASIC zum Nachfolger von VB6 avanciert und wir noch weitere Tools damit entwickeln werden, stellen wir in einem weiteren Beitrag namens **Ribbon: Callback-Signaturen für VBA und VB6** (**www.access-im-unternehmen.de/1324**) die Unterschiede vor.

```
Sub OnChange(control As IRibbonControl, text As String)
    Dim db As DAO.Database
    Dim prp As DAO.Property
    Set db = objAccess.CurrentDb
    If Len(text) = 0 Then
        db.Properties.Delete "AppTitle"
    Else
        On Error Resume Next
        Set prp = db.Properties("AppTitle")
        If Not Err.Number = 0 Then
            Set prp = db.CreateProperty("AppTitle", dbText, text)
            db.Properties.Append prp
        Else
            prp.Value = text
        End If
    End If
    objAccess.RefreshTitleBar
End Sub
```

Listing 2: Ändern einer Option nach Änderung im Ribbon

Speichern des geänderten Datenbanktitels

Mit der gleichen Eigenschaft beschäftigt sich die Callbackfunktion **onChange** (siehe Listing 2). Diese hat die gleichen Parameter. Allerdings wird sie ausgelöst, wenn der Benutzer den im **editBox**-Steuerelement enthaltenen Text ändert und die Änderung beispielsweise durch Verlassen des Textfeldes abschließt. Dementsprechend ist der zweite Parameter **text** diesmal auch kein Rückgabeparameter, sondern er liefert den neu vom Benutzer eingegebenen Text.

Den Inhalt von **text** prüft die Prozedur zunächst in einer **If...Then**-Bedingung. Es kann sein, dass der Benutzer das **editBox**-Steuerelement komplett geleert hat, dann soll auch der Titel geleert werden. Das funktioniert allerdings nur in der gewünschten Form, indem wir die Eigenschaft **AppTitle** wieder löschen. Das erledigt die Prozedur im **If**-Teil der Bedingung mit der **Delete**-Methode der **Properties**-Auflistung.

Hat der Benutzer jedoch einen Titel mit mindestens einem Zeichen eingegeben, dann wollen wir die Eigenschaft **AppTitle** auf diesen Wert einstellen. Dazu versucht die Prozedur, nach vorheriger Deaktivierung der Fehlerbehaldnung, die Eigenschaft mit der Variablen **prp** zu referenzieren. Ist dabei ein Fehler aufgetreten, erstellt die Prozedur die Eigenschaft neu und hängt diese an die Auflistung **Properties** des **Database**-Objekts der aktuellen Datenbank an.

Ist die Eigenschaft bereits vorhanden, stellt die Prozedur diese lediglich auf den Wert aus dem Parameter **text** ein.

In jedem Fall soll die Anzeige der Titelleiste anschließend aktualisiert werden, was die Methode **RefreshTitleBar** des **Application**-Objekts der aktuellen Access-Instanz erledigt. Diese haben wir ja zuvor in die Variable **objAccess** eingelesen.

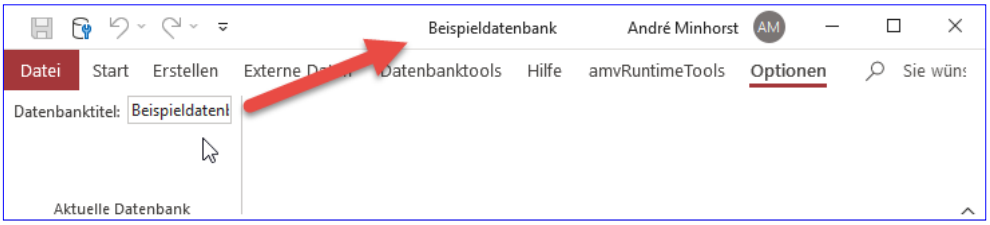


Bild 5: Die erste Access-Option im Access-Ribbon

Nachdem Sie die Ribbon-Definition in der Prozedur GetCustomUI angepasst und die beiden Callbackfunktionen hinzugefügt haben, können Sie die Änderungen speichern und das COM-Add-In neu kompilieren. Das Ergebnis der Kompilierung finden Sie übrigens unten in Visual Studio Code in der Debugging-Console.

Wenn Sie danach eine Access-Datenbank öffnen, finden Sie ein neues Tab namens **Optionen** im Ribbon. Klicken Sie es an, erscheint die von uns hinzugefügte Option zum Einstellen des Datenbanktitels (siehe Bild 5). Dieses zeigt direkt beim Öffnen den aktuellen Wert an – bei einer Datenbank, für die Sie noch keinen Anwendungstitel eingestellt haben, beispielsweise **<Kein Titel>**. Nach einer Änderung wird der neue Titel direkt in die Titelleiste übertragen. Leeren Sie die Eigenschaft im Ribbon, erscheint wieder der Standardtitel, bestehend aus Datenbankname und Pfad.

Option zum Ändern des Anwendungssymbols

Wir wollen gleich die nächste Option aus dem Optionen-Dialog ermitteln und ausprobieren, nämlich Anwendungssymbol. Hier haben wir die zusätzliche Herausforderung, dass wir einen **Datei auswählen**-Dialog hinzufügen wollen. Also benötigen wir nicht nur ein **editBox**-Steuerelement, sondern daneben auch noch ein **button**-Element. Damit diese nebeneinander erscheinen, fassen wir die beiden Elemente in einem **box**-Element zusammen. Die notwendige Erweiterung der Prozedur **GetCustomUI** sehen Sie in Listing 3.

Eigene Callback-Prozedur für gleiche Attribute für jedes Steuerelement?

Bei Ereignisprozeduren von Formularen, Berichten oder Steuerelementen wird für jedes Ereignis für jedes Element eine neue Prozedur erstellt, die den Namen des Elements, einen Unterstrich und die Bezeichnung des Ereignisses als Bezeichnung trägt. Bei Callbackfunktionen des Ribbons hat es sich eingebürgert, dass man zum Beispiel für das Ereignis **onAction** nur eine Callback-Funktion namens **onAction** definiert. Diese ermittelt dann über den Parameter **control** und dessen Eigenschaft **id** den Namen des aufrufenden Steuerelements. In einer **Select Case**-Bedingung wird dann der Name ausgewertet und in den **Case**-Zweigen finden sich dann die Anweisungen, die für das jeweilige Steuerelement ausgeführt werden sollen.

Sie können natürlich auch beispielsweise für das **onAction**-Attribut eines Elements namens **btnBeispiel** einen Wert wie **btnBeispiel_OnAction** anlegen und eine entsprechende Callbackfunktion hinterlegen. Je nachdem, wie umfangreich das Ribbon ist und wie aufwendig der Code, kann dies die bessere Variante sein. Wir haben

```
Private Function GetCustomUI(ByVal RibbonID As String) As String _
    Implements IRibbonExtensibility.GetCustomUI
    ...
    strXML &= "          <editBox label="Anwendungstitel:" id="txtTitle" maxLength="255" getText="getText"" _
        & " onChange="onChange"/>" & vbCrLf
    strXML &= "          <box id="box1">" & vbCrLf
    strXML &= "            <editBox label="Anwendungssymbol:" id="txtIcon" maxLength="255"" _
        & " getText="getText" onChange="onChange"/>" & vbCrLf
    strXML &= "            <button screentip="Symbol auswählen ..." label="..." id="btnChooseSymbol"" _
        & " onAction="onAction"/>" & vbCrLf
    strXML &= "          </box>" & vbCrLf
    ...
End Function
```

Listing 3: Erweiterung der Ribbon-Definition für das Anpassen des Anwendungssymbols