HAUFE.
Ausgabe 03/2021

ACCESS
IM UNTERNEHMEN

PROGRAMMIEREN MIT TWINBASIC

Eine neue Möglichkeit, COM-Add-Ins und DLL-Funktionsbibliotheken zu programmieren – auf Basis von VB und VBA! (ab S. 7)

In diesem Heft:

ABHÄNGIGE LISTENFELDER

Zeigen Sie in Listenfeld B nur Einträge an, die zu denen aus Listenfeld A passen.

SEITE 2

SCHALTFLÄCHEN-ASSISTENT

Erstellen Sie OK- und Abbrechen-Schaltflächen per Mausklick mit unserem Assistenten.

SEITE 49

SCHALTFLÄCHEN PER CODE ANLEGEN

Erfahren Sie, wie Sie Schaltflächen in Access-Formularen per VBA-Code anlegen.

SEITE 57

twinBASIC – Perfekte Ergänzung

Die Entwicklungsumgebung von Microsoft Access wurde seit 2010 nicht mehr aktualisiert, der VBA-Editor noch länger nicht mehr. Gleiches gilt für VB6: Visual Studio 6 wurde zwar durch Visual Studio .NET mit den entsprechenden Programmiersprachen wie Visual Basic.NET oder C# ersetzt, aber wer bestehende Projekte hatte, wurde von Microsoft im Regen stehen gelassen. Dem schafft nun der Programmierer Wayne Philips Abhilfe, indem er mit twinBASIC eine tolle neue Möglichkeit zur Programmierung mit VB schafft.

Gerade in diesen Zeiten, wo immer mehr Office-Pakete in der 64-Bit-Variante installiert werden, macht sich Wehmut breit: Liebgewonnene DLLs, die mit VB6 erstellt wurden, lassen sich damit nicht mehr nutzen. Und auch Elemente der Benutzeroberfläche von Drittherstellern, die von ihren Entwicklern nicht nach 64-Bit migriert wurden, müssen ersetzt werden.

Da kommt es doch gelegen, dass ein in Access-Kreisen schon recht bekannter Entwickler namens Wayne Philips sich aufmacht, eine neue Möglichkeit zum Erstellen von VB-DLLs und -Anwendungen zu schaffen, die ohne die Tools von Microsoft auskommt. Genau genommen hat Wayne eine Erweiterung namens **twinBASIC** programmiert für die Entwicklungsumgebung Visual Studio Code. Das ist wiederum ein kostenloser Code-Editor.

Was Sie schon jetzt, wenige Wochen nach der Erstveröffentlichung mit twinBASIC alles erledigen können, schauen wir uns im Detail in dieser Ausgabe an. Im ersten Beitrag namens **twinBASIC – VB/VBA mit moderner Umgebung** schauen wir uns ab Seite 7 an, wie Sie die Tools installieren und welche Möglichkeiten es bietet.

Der zweite Beitrag zum Thema namens **COM-DLLs mit twinBASIC** zeigt ab Seite 15, wie Sie mit DLLs programmieren können, mit denen Sie beispielsweise wichtige Funktionen kapseln und dann über die DLL in allen Datenbankanwendungen verfügbar machen können.

Da auch beim Programmieren mit twinBASIC Fehler anfallen können, zeigen wir im Beitrag **Debugging in twinBASIC**, welche Debugging-Möglichkeiten twinBASIC in Visual Studio Code anbietet (ab S. 22).

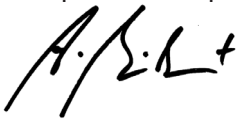
Nach dem Aufwärmen geht es an die richtig interessanten Themen: Wir erstellen mit twinBASIC ein COM-Add-In für Access – und damit können wir tatsächlich sehr komfortabel die Access-Benutzeroberfläche um eigene Funktionen erweitern! Wie das im Detail gelingt, lesen Sie unter dem Titel **twinBASIC: COM-Add-Ins für Access** ab S. 28.

Das Gleiche können Sie praktischerweise mit dem gleichen Tool auch für den VBA-Editor durchführen. Diesen erweitern Sie dann ebenfalls um eigene, neue Funktionen. Die wichtigsten Infos dazu finden Sie ab Seite 39 im Artikel **twinBASIC: COM-Add-Ins für den VBA-Editor**.

Und wo wir schon beim Erweitern sind, stellen wir Ihnen gleich auch noch einen Schaltflächen-Assistenten vor, mit dem Sie leicht benutzerdefinierte Schaltflächen zu einem neuen Formular hinzufügen können. Die Techniken lernen Sie in den beiden Beiträgen **Schaltflächen-Assistent** ab S. 49 und **Schaltflächen per Code anlegen** ab S. 57 kennen.

Und schließlich beschäftigen wir uns im Beitrag **Abhängige Listenfelder** ab Seite 2 noch mit abhängigen Listenfeldern.

Viel Spaß beim Ausprobieren!



Ihr André Minhorst

Abhängige Listenfelder

Die Programmierung abhängiger Kombinationsfelder haben wir bereits mindestens einmal behandelt. Hier soll ein Kombinationsfeld nur Werte abhängig von der Auswahl eines anderen Kombinationsfeldes anzeigen, also zum Beispiel die Artikel zu einer vorher gewählten Kategorie. Mit Listenfeldern geht das auch – dazu sind nur wenige Umstellungen nötig. Zumindest, wenn Sie im ersten Listenfeld nur die Auswahl eines Eintrags gleichzeitig erlauben. In diesem Beitrag wollen wir uns nicht nur das anschauen, sondern auch die Anzeige abhängiger Daten bei einem Listenfeld mit Mehrfachauswahl.

Abhängigkeit bei Kombinationsfeldern

Bei abhängigen Kombinationsfeldern wie denen aus Bild 1 gelingt das Abbilden der Abhängigkeit wie folgt:

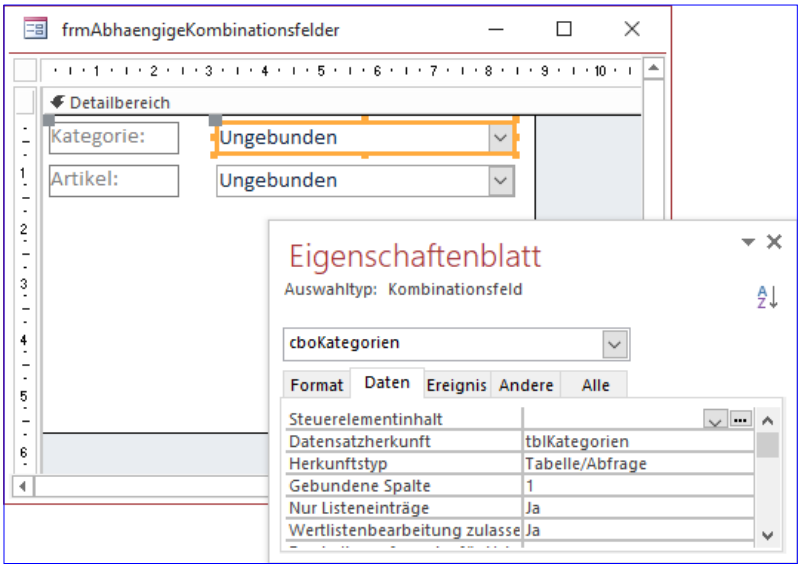


Bild 1: Abhängige Kombinationsfelder in der Entwurfsansicht

Das Kombinationsfeld **cboKategorien** zeigt die Werte der Felder **KategorieID** und **Kategorie** der Tabelle **tblKategorien** an. Wenn der Benutzer einen der Einträge auswählt, soll das zweite Kombinationsfeld **cboArtikel** nur noch die Artikel anzeigen, die zu der im ersten Kombinationsfeld gewählten Kategorie passen. Die Aktualisierung führen wir in der Ereignisprozedur aus, die durch das Wählen eines neuen Eintrags im Listenfeld **cboKategorien** ausgelöst wird, nämlich **Nach Aktualisierung**. Die Ereignisprozedur sieht so aus:

```
Private Sub cboKategorien_AfterUpdate()
    Dim strSQL As String
    strSQL = "SELECT ArtikelID, Artikelname FROM tblArtikel
             WHERE KategorieID = " & Nz(Me!cboKategorien)
    Me!cboArtikel.RowSource = strSQL
End Sub
```

Dadurch weisen wir der Eigenschaft **Datensatzherkunft** einen SQL-Ausdruck zu, der nur noch die Datensätze der Tabelle **tblArtikel** liefert, deren **KategorieID** im ersten Kombinationsfeld ausgewählt wurde. Das Ergebnis ist eine

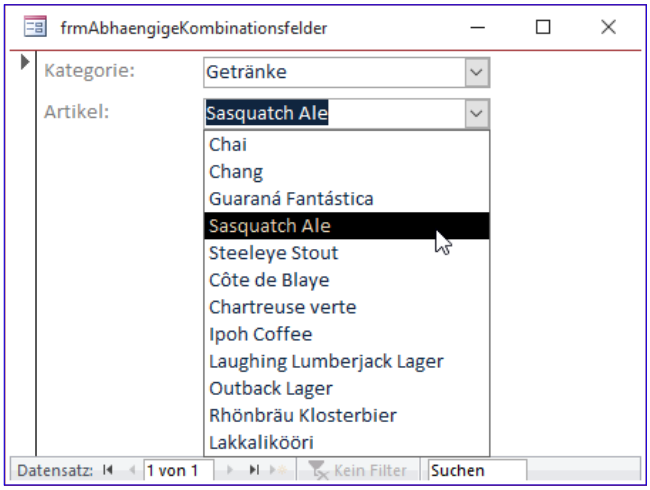


Bild 2: Auswahl abhängiger Datensätze

erheblich reduzierte Anzahl von Datensätzen im zweiten Kombinationsfeld (siehe Bild 2).

Umsetzung für abhängige Listenfelder

Wie unterscheidet sich dies, wenn wir statt Kombinationsfelder einfach Listenfelder verwenden? Dazu kopieren wir dieses Formular **frmAbhaengigeKombinationfselder** in ein neues Formular namens **frmAbhaengigeListenfelder** und wandeln die Kombinationsfelder mit dem Kontextmenü-Eintrag **Ändern zulListenfeld** jeweils in Listenfelder um (siehe Bild 3).

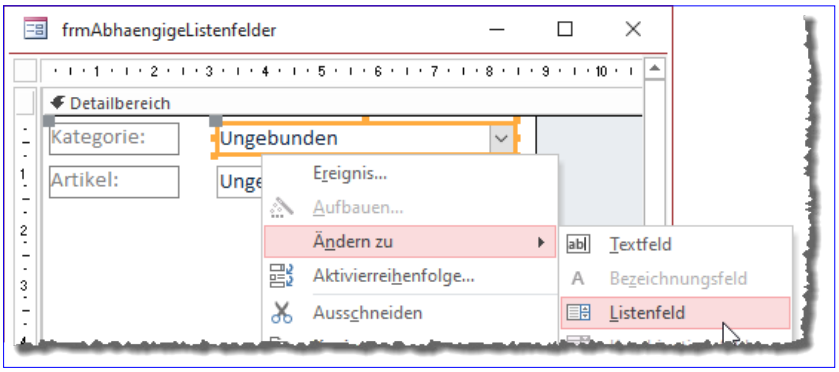


Bild 3: Umwandeln von Kombinationsfeld in Listenfeld

Danach ändern wir noch die Position und Größe der Listenfelder, damit diese ausreichend Datensätze anzeigen können und nebeneinander angezeigt werden (siehe Bild 4).

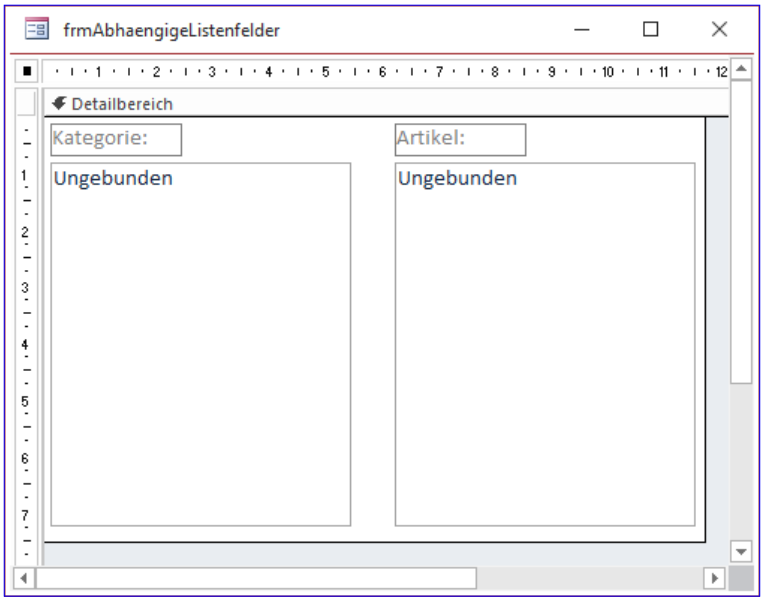


Bild 4: Ausrichten der Listenfelder

Außerdem benennen wir die Listenfelder noch um von **cboKategorien** und **cboArtikel** in **lstKategorien** und **lstArtikel**. Damit brauchen wir nun nur noch das Ereignis **Nach Aktualisierung** neu zu implementieren. Hier ändern sich die Bezeichnungen von Prozedurname und den betroffenen Steuerelementen:

```
Private Sub lstKategorien_AfterUpdate()
    Dim strSQL As String
    strSQL = "SELECT ArtikelID, Artikelname FROM tblArtikel
                WHERE KategorieID = " & Nz(Me!lstKategorien)
    Me!lstArtikel.RowSource = strSQL
End Sub
```

Das Ergebnis sehen Sie dann in Bild 5. Das Filtern gelingt genauso wie mit den Kombinationsfeldern.

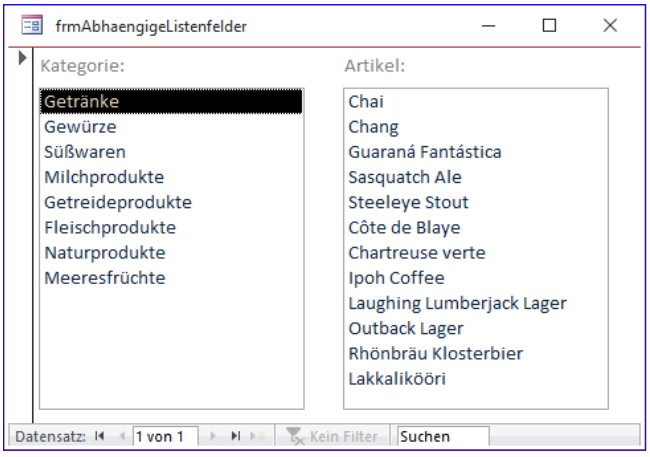


Bild 5: Filtern per Listenfeld

Damit kommen wir zu den anspruchsvolleren Aufgaben – dem Filtern nach mehreren ausgewählten Kategorien.

Filtern mit Mehrfachauswahl

Bevor wir zur Praxis schreiten, kopieren wir das Formular **frmAbhaengigeListenfelder** in ein neues Formular namens **frmAbhaengigeListenfelderMehrfach**.

Um das Filtern mit Mehrfachauswahl zu implementieren, müssen wir erst einmal das Listenfeld **lstKategorien** so umstellen, dass es die Mehrfachauswahl anbietet (siehe Bild 6).

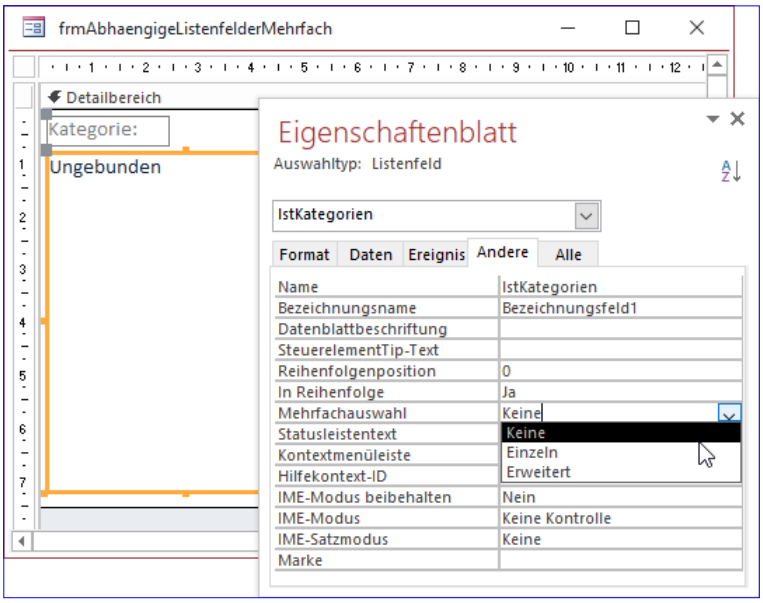


Bild 6: Aktivieren der Mehrfachauswahl

Dazu gibt es für die gleichnamige Eigenschaft neben der Standardeinstellung **Keine** die folgenden beiden Optionen:

- **Einzeln**: Sie können jeweils einen Eintrag per Mausklick zur Auswahl hinzufügen oder aus dieser entfernen.
- **Erweitert**: Ein einfacher Klick markiert jeweils das angeklickte Element. Klicken bei gedrückter **Strg**-Taste fügt das aktuell angeklickte Element zur Auswahl hinzu oder entfernt dieses aus der Auswahl. Bei gedrückter Umschalttaste werden alle Elemente zwischen dem zuletzt hinzugefügten Element und dem nun angeklickten Element zur Auswahl hinzugefügt.

Wenn wir die Einstellung einfach auf **Einzeln** einstellen, zur Formularansicht wechseln und dann einen Eintrag markieren, verschwinden alle Einträge aus dem Artikel-Listenfeld.

Der Grund ist, dass wir bei den Einstellungen **Einzeln** und **Erweitert** nicht mehr über **Me!lstKategorien** auf den aktuell markierten Wert zugreifen können. Das geschieht nunmehr über eine Liste namens **SelectedItems**, mit der wir den Index der aktuell markierten Einträge ermitteln können.

Um diese herauszufinden, nutzen wir eine Schleife wie die folgende:

```
Dim var As Variant
For Each var In Me!lstKategorien.ItemsSelected
    Debug.Print var
Next var
```

Wenn beispielsweise der erste, dritte und fünfte Eintrag markiert sind, wird **0**, **2** und **4** im Direktbereich ausgegeben.

Wir benötigen aber die Primärschlüsselfelder der ausgewählten **Kategorie**-Datensätze. Diese liefert die Eigenschaft **ItemData**, wenn wir dieser den Index des gesuchten Eintrags übermitteln:

```
Debug.Print Me!lstKategorien.ItemData(var)
```

Nun müssen wir die Abfrage nach den Artikeln noch so anpassen, dass nicht nur einer, sondern gleich mehrere Kategorien geprüft werden. Dazu verwenden wir die **IN**-Klausel wie in folgender Abfrage:

```
SELECT ArtikelID, Artikelname
FROM tblArtikel
WHERE KategorieID IN (1, 3, 5)
```

twinBASIC – VB/VBA mit moderner Umgebung

Wayne Philips ist Access-Entwicklern ein Begriff. Es ist der Erfinder von vbWatchDog, dem Tool für eine professionelle Fehlerbehandlung und er schafft es, .accde-Datenbanken wieder in .accdb-Datenbanken zurückzuverwandeln. Nun kommt sein nächster großer Coup: twinBASIC. Dabei handelt es sich um eine moderne Version der klassischen Programmiersprachen VB6 und VBA. Modern deshalb, weil Sie damit in einer modernen Entwicklungsumgebung arbeiten können statt im angestaubten VBA-Editor, nämlich in VS Code. Und weil es neue Elemente zu diesen Sprachen hinzufügt. Das Ergebnis sind eigenständige Anwendungen, DLLs und mehr. Wir schauen uns in diesem Beitrag an, wie Sie twinBASIC einsatzbereit machen. In weiteren Beiträgen geht es dann in die Details – hier lernen Sie dann, wie Sie beispielsweise eine COM-DLL bauen, die Sie unter Access referenzieren können.

VS Code für twinBASIC

Bevor wir starten: Was ist dieses VS Code eigentlich? Es ist ein recht neuer Quelltext-Editor von Microsoft, der plattformübergreifend verfügbar ist, also für Windows, macOS und Linux. Es bietet einige Funktionen, die im Visual Basic Editor nur teilweise oder gar nicht vorhanden sind, zum Beispiel Syntaxhervorhebung, Code-Faltung, Debugging, Autovervollständigung oder Versionsverwaltung.

Es ist nicht zu verwechseln mit Visual Studio. Visual Studio arbeitet mit Projektdateien, VS Code mit Quelltextdateien und Ordnern. Diese werden in einem Workspace verwaltet und der aktuelle Zustand des Workspaces kann gespei-

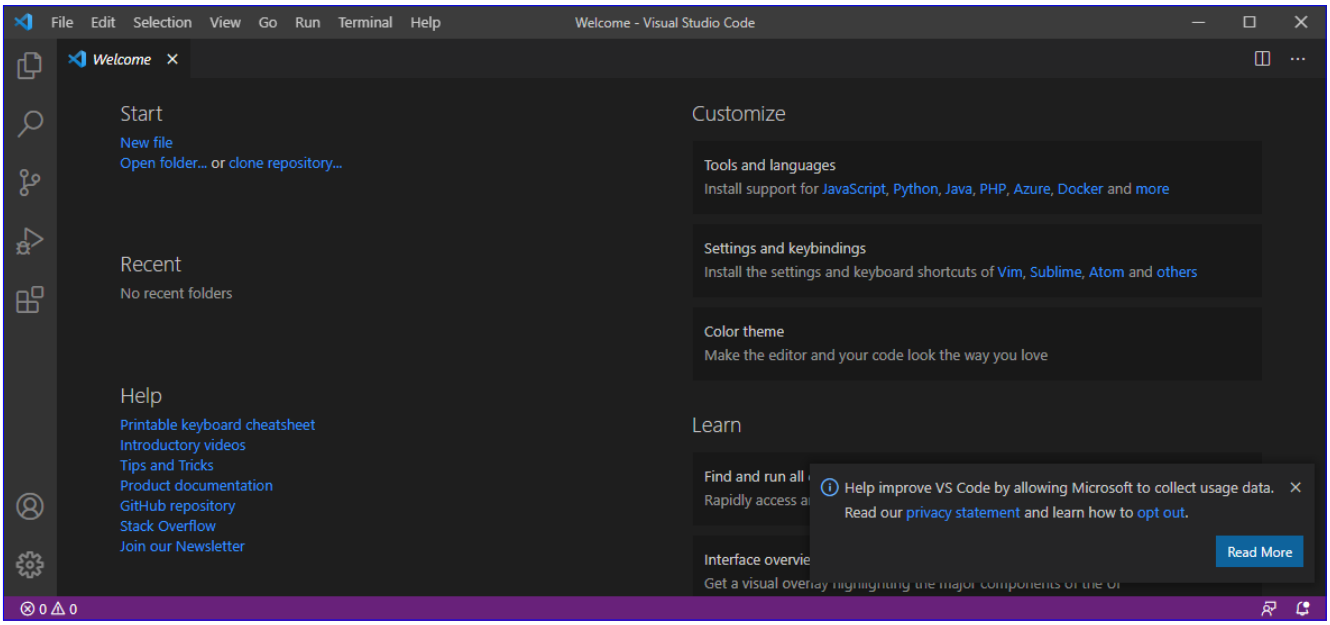


Bild 1: VS Code direkt nach der Installation

chert und wiederhergestellt werden.

Es bietet ein Plug-In-System, das sich auch twinBASIC zunutze macht. Wayne Philips hat ein Plug-In programmiert, mit dem Sie in VS Code in der Sprache twinBASIC programmieren und auch die ausführbaren Dateien kompilieren können.

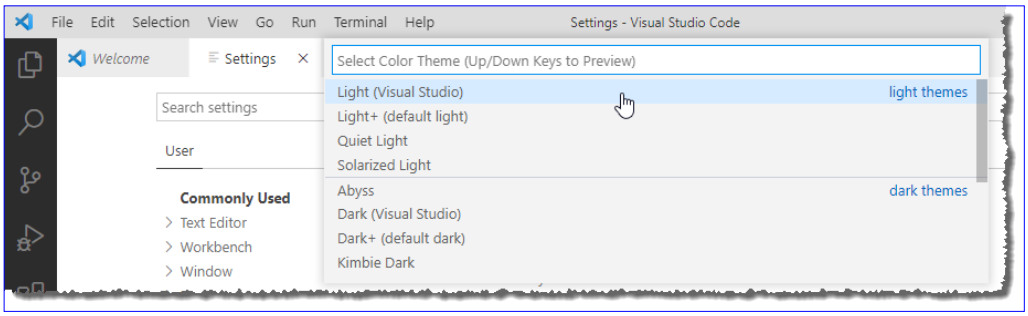


Bild 2: Einstellen des Farbschemas

VS Code ist Open Source und wird von einem Entwicklerteam rund um Erich Gamma entwickelt. Erich Gamma hat bereits das Buch **Design Patterns - Elements of Reusable Object-Oriented Software** geschrieben und die Entwicklung der Entwicklungsumgebung **Eclipse** geleitet.

VS Code herunterladen und installieren

Den Download von VS Code finden Sie unter folgendem Link:

```
https://code.visualstudio.com/download
```

Hier wählen Sie die Windows-Version für Ihr System aus und laden die entsprechende Datei herunter. Anschließend starten Sie die heruntergeladene **.exe**-Datei, um VS Code zu installieren. Im Setup können Sie noch angeben, ob VS Code im Kontextmenü für Dateien untergebracht werden soll oder ob bestimmte Dateitypen direkt damit verknüpft werden sollen.

Danach können wir VS Code direkt starten. Es erscheint dann wie in Bild 1. Auch wenn der schwarze Hintergrund toll aussieht und die meisten sicher der englischen Sprache mächtig sind, nehmen wir hier gleich einmal zwei

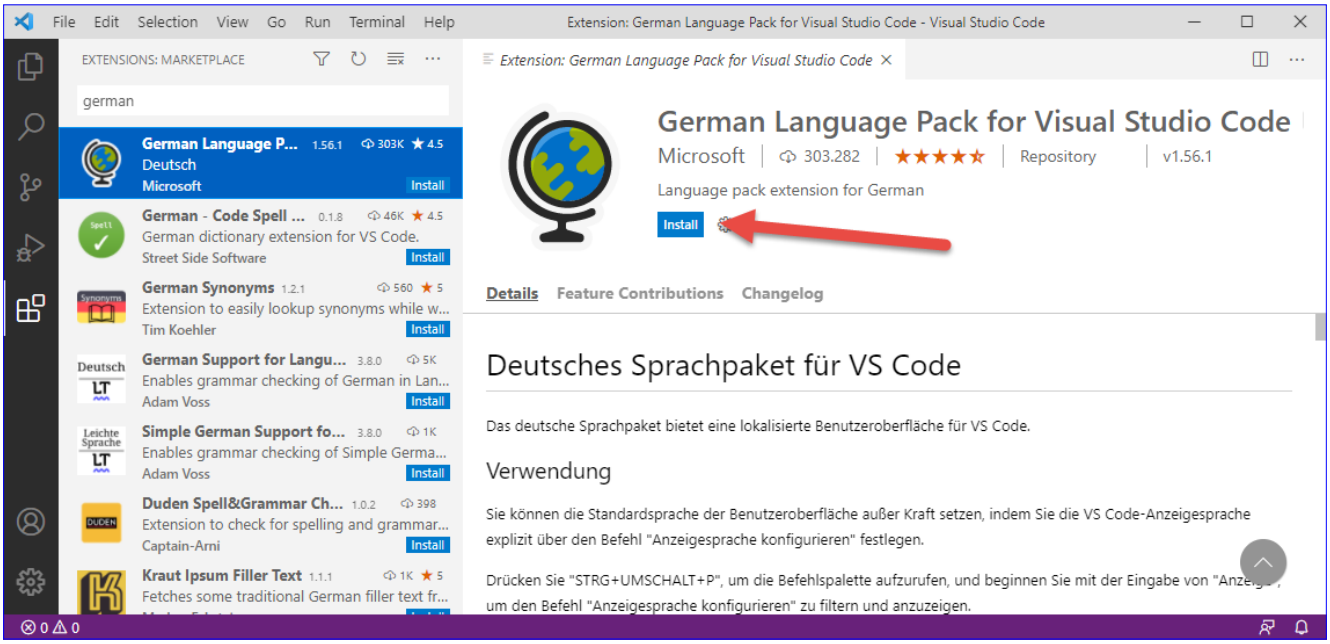


Bild 3: Installieren des deutschen Sprachpakets

Änderungen bezüglich dieser Eigenschaften vor.

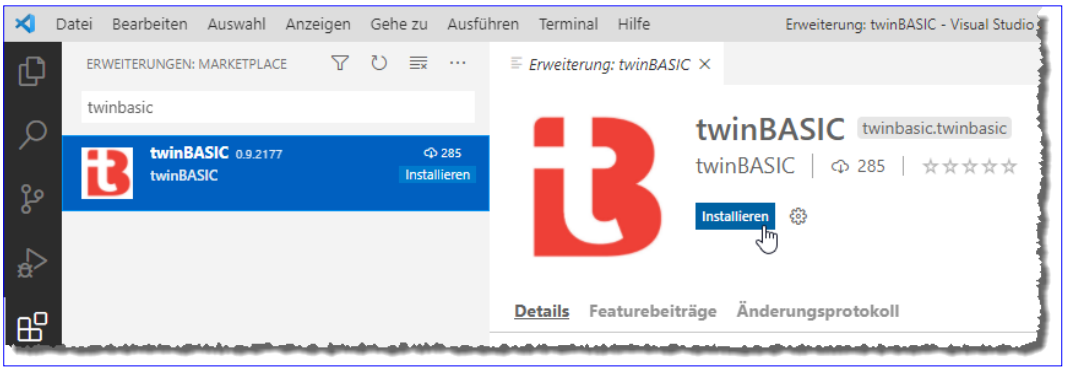


Bild 4: Installieren der twinBASIC-Erweiterung

Farbschema einstellen

Als Erstes stellen wir die Farbe ein. Dazu nutzen wir dem Menübefehl **File|Preferences|Color Theme**. Hier wählen wir das Schema **Light (Visual Studio)** aus (siehe Bild 2).

Sprache einstellen

Wir hätten gern deutsche Befehle und dazu ist die Installation eines Language Packs nötig. Dazu wählen Sie den Menübefehl **File|Preferences|Extensions** aus. Im nun erscheinenden Bereich **Extensions: Marketplace** geben Sie einfach **German** ein und erhalten direkt das **German Language Pack for Visual Studio Code**.

Dazu klicken Sie auf die in Bild 3 sichtbare Schaltfläche **Install**. Die Installation geschieht in etwa einer Sekunde, der anschließende Neustart liefert dann direkt die deutschen Bezeichnungen für die Befehle der Benutzeroberfläche.

twinBASIC-Plugin installieren

Da wir nun schon Übung im Installieren von Extensions haben, fügen wir gleich noch die Erweiterung für twinBASIC hinzu. Also öffnen wir wieder den Bereich von eben, diesmal aber über den deutschsprachigen Menübefehl **Datei|Einstellungen|Erweiterungen**. Im nun erscheinenden Bereich **Erweiterungen** geben wir als Suchbegriff **twinBASIC** ein.

Anschließend klicken Sie auch hier auf die Schaltfläche **Installieren** (siehe Bild 4).

twinBASIC nutzen: Hello World!

Natürlich starten wir mit einem **Hello World!**-Beispiel in die Nutzung von twinBASIC. Wir haben nun VS Code installiert und die twinBASIC-Erweiterung installiert – wie geht es nun weiter? Wenn wir über den Befehl **Datei|Neue Datei** eine neue Datei anlegen, geschieht jedenfalls nichts besonderes – es erscheint eine leere Textdatei, die Sie füllen können. Kein IntelliSense oder ähnliches in Sicht.

Hello World!-Projekt öffnen

twinBASIC verfügt zu diesem Zeitpunkt noch nicht über die Möglichkeit, ein komplett neues Projekt zu erstellen, daher starten wir mit einem vorgefertigten Projekt. Dieses enthält zwei Dateien:

- helloword.code-workspace
- helloworld.twinproj

Nach dem Herunterladen der Dateien wählen Sie in VS Code den Befehl **Datei|Arbeitsbereich öffnen** aus und öffnen über den **Arbeitsbereich öffnen**-Dialog die Workspace-Datei.

Wenn Sie den Explorer anzeigen und den Arbeitsbereich **HelloWorld** aufklappen, finden Sie wie in Bild 5 die Datei **HelloWorld.twin**, die Sie per Doppelklick öffnen und so den Code des Moduls anzeigen.

Klicken Sie nun in den Code der Prozedur **Public Sub Main**, können Sie diese mit der Taste **F5** ausführen. Dies zeigt das erwartete Meldungsfenster an.

Da es sich hier um die **Main**-Prozedur handelt, die auch beim Starten einer auf Basis dieses Projekts erstellten **.exe**-Datei aufgerufen wird, können Sie auch einfach auf die **Build**-Schaltfläche klicken.

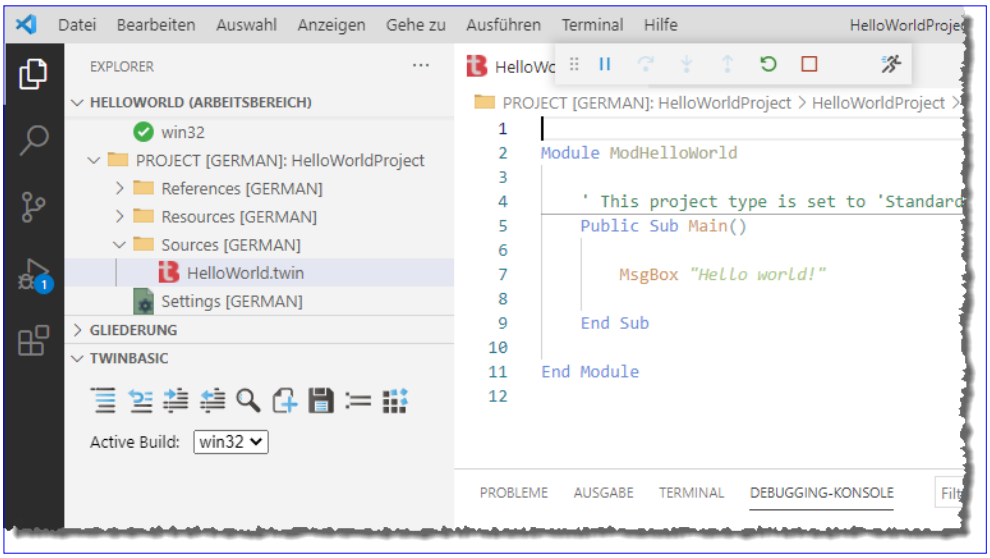


Bild 5: Das Hello World!-Projekt in VS Code

.exe erstellen

Jetzt kommt der Clou: Mit einem Klick auf die **Build**-Schaltfläche erstellen Sie direkt die **.exe**-Datei für das Projekt (siehe Bild 6).

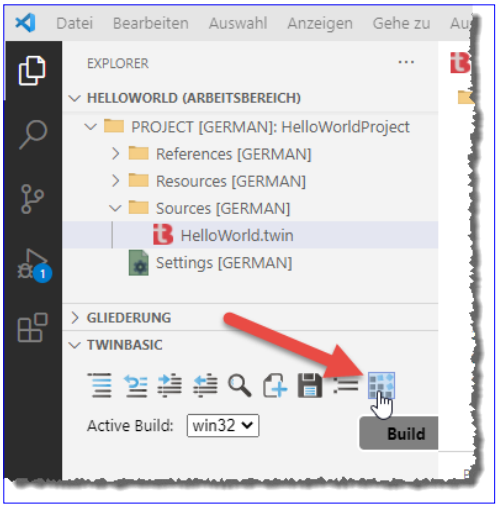


Bild 6: Erstellen des Projekts

Im Ordner **Build** finden wir dann die Datei **HelloWorld-Project_win32.exe** vor, die wir per Doppelklick ausführen können (siehe Bild 7). Dies liefert die gewünschte Meldung. Richtig spannend wird es bei der Dateigröße: Diese beträgt nämlich gerade mal 8 KB! Und wir benötigen neben dieser Datei keine weiteren Dateien.

Dateien des Projekts

Weiter oben haben wir bereits die beiden Dateien des ersten Projekts erwähnt. Die Datei mit der Dateiendung **.code-workspace** ist etwa mit der Solution aus Visual Studio-Projekten vergleichbar.

Die Datei mit der Endung **.twinproj** ist die Projektdatei, die ein virtuelles Dateisystem mit den verschiedenen Elementen des Projekts enthält. Diese beiden Dateien

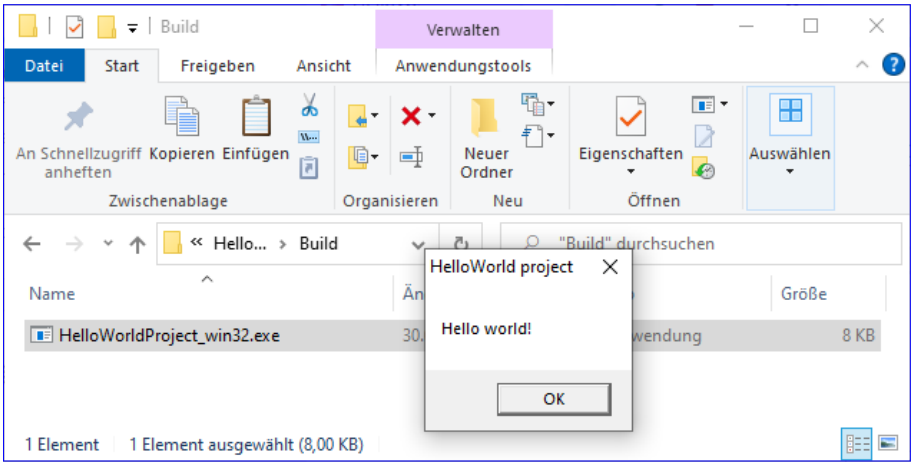


Bild 7: Die .exe-Datei und die dadurch angezeigte Meldung

COM-DLLs mit twinBASIC

Bisher haben wir COM-DLLs entweder mit Visual Studio 6 erstellt oder mit Visual Studio .NET. Mit twinBASIC kommt nun eine weitere Möglichkeit hinzu. twinBASIC ist noch in der Entwicklung, aber Sie können damit durchaus bereits lauffähige COM-DLLs bauen. Dieser Beitrag zeigt, wie Sie das Beispiel-COM-DLL des Entwicklers von twinBASIC, Wayne Philips, verwenden und wie Sie eigene Funktionen hinzufügen.

Vorbereitungen zum Erstellen von COM-DLLs mit twinBASIC

Den Großteil der Vorbereitungen für den Einsatz von twinBASIC haben wir im Beitrag **twinBASIC – VB/VBA mit moderner Umgebung** (**www.access-im-unternehmen.de/1303**) beschrieben. Hier erfahren Sie, wie Sie die Entwicklungsumgebung VS Code installieren und die benötigte Erweiterung twinBASIC hinzufügen. Außerdem haben wir dort ein kleines Beispielprojekt demonstriert.

Aktuelle Versionen

Neben der Version, die wir für die Beschreibungen in diesem Beitrag genutzt haben, wird twinBASIC aktuell sehr schnell weiterentwickelt.

Da es für uns Access-Entwickler sehr ungewohnt ist, dass eine Entwicklungsumgebung weiterentwickelt wird (und das auch noch schnell), weisen wir hier explizit darauf hin, dass es aktuelle Versionen und Informationen an folgenden Orten gibt:

- Die aktuellen Downloads finden Sie unter **https://twinbasic.com**.
- Bugs et cetera können Sie hier melden oder auch einsehen: **https://github.com/WaynePhillipsEA/twinbasic/issues**

Auf **https://twinbasic.com** finden Sie auch einige Beispiele, unter anderem das hier als Grundlage verwendete Beispiel für eine ActiveX-DLL.

Beispiel herunterladen

Die zum Zeitpunkt der Erstellung dieses Beitrags verwendete Version des Beispiels finden Sie wie gewohnt im Download zu diesem Beitrag, aktuellere Versionen gegebenenfalls unter dem oben genannten Link.

Wenn Sie die Zip-Datei heruntergeladen haben, finden Sie darin zwei Dateien – eine mit der Dateiendung **.code-workspace** und eine mit der Dateiendung **.twinproj**. Um das Projekt zu öffnen, klicken Sie doppelt auf die Datei mit der Dateiendung **.code-workspace** (gegebenenfalls müssen Sie die Dateiendung noch mit dieser Anwendung verknüpfen).

Dies öffnet den Arbeitsbereich, der sowohl das Projekt aus der **.twinproj**-Datei anzeigt als auch alle notwendigen Einstellungen enthält. Das Ergebnis sieht nach einem Doppelklick auf den Eintrag **HelloWorld.twin** im Ordner **Sources** wie in Bild 1 aus.

Wir sehen hier eine Klasse namens **MyTestLibrary** mit einer einzigen öffentlich deklarierten Funktion namens **MultipleByTen**.

Projekt erstellen

Wir schauen uns zunächst an, wie wir die COM-DLL zum Laufen bringen. Dazu brauchen Sie lediglich auf die Schaltfläche **Build** im Bereich **TWINBASIC** zu klicken.

Dies erzeugt die **.dll**-Datei im Ordner **Build** des Projektordners und legt gleichzeitig die benötigten Einträge in der Registry an – dazu später mehr.

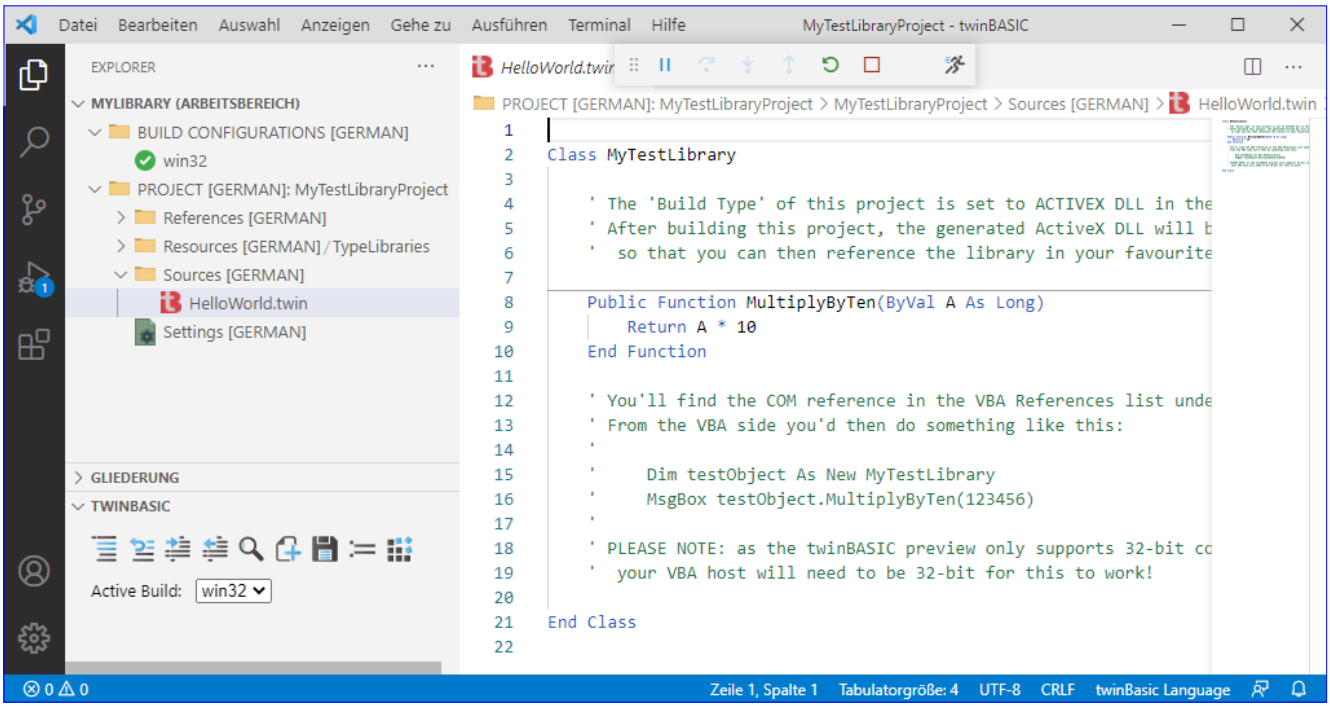


Bild 1: COM-Add-In-Projekt in VS Code

COM-DLL einbinden

Wir öffnen eine Access-Datenbank und starten den VBA-Editor. Hier rufen wir mit dem Menü-Eintrag **Extras|Verweise** den **Verweise**-Dialog auf. Dieser bietet unter dem Namen **MyTestLibrary project** einen neuen Eintrag zum Einbinden der DLL an (siehe Bild 2). Diesen aktivieren wir durch Platzieren eines Hakens im entsprechenden Kontrollkästchen.

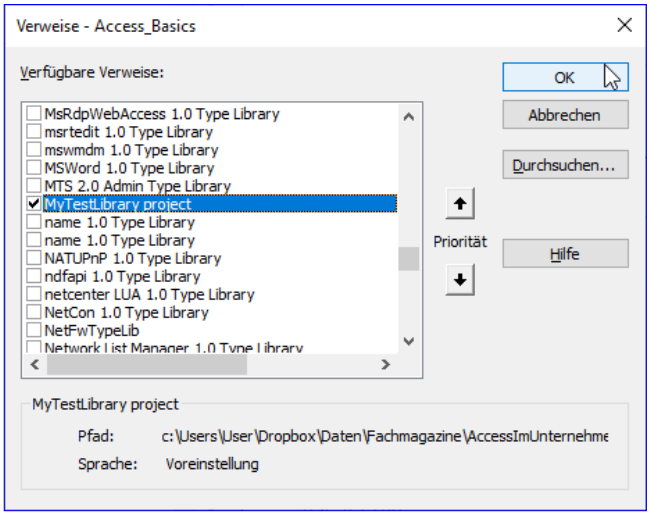


Bild 2: Verweis auf die DLL

Befehle der COM-DLL ausprobieren

Danach können Sie im VBA-Editor folgende Prozedur anlegen:

```
Public Sub TestCOMDLL()
    Dim obj As MyTestLibrary
    Set obj = New MyTestLibrary
    Debug.Print obj.MultiplyByTen(10)
End Sub
```

Rufen Sie die Prozedur auf, liefert diese mit dem Wert **100** das korrekte Ergebnis. Auch IntelliSense funktioniert sehr gut mit der Funktion (siehe Bild 3).

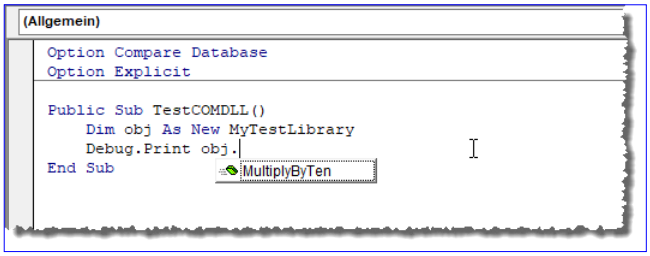


Bild 3: IntelliSense mit der COM-DLL

Late Binding

Das Erstellen funktioniert auch mit Late Binding. Allerdings müssen Sie die Bibliothek zusätzlich zur Klasse angeben. Das sieht dann beispielsweise wie folgt aus:

```
Public Sub TestCOMDLL1()
    Dim obj As Object
    Set obj = CreateObject(_
                  "MyTestLibraryProject.MyTestLibrary")
    Debug.Print obj.MultiplyByTen(10)
End Sub
```

Geben Sie die Bibliothek nicht an, führt dies zu dem Fehler aus Bild 4.

Registrierung der COM-DLL

Die Registrierung der COM-DLL erfolgt beim Build-Vorgang. Diese Information erhalten Sie, wenn Sie sich das Debug-Fenster von VS Code ansehen (siehe Bild 5). Diesen Bereich aktivieren Sie mit dem Menüpunkt **AnzeigenlDebugging-Konsole**.

Sie können dazu nicht die übliche Anweisung **Regexp32.exe <Pfad zur DLL>** verwenden, mit der normalerweise DLLs registriert werden.

Somit steht auch der Befehl **Regexp32.exe <Pfad zur DLL> -u** nicht zur Verfügung, mit dem Sie auf diese Weise registrierte DLLs normalerweise wieder deregistrieren können.

Einstellungen für das Projekt

Wenn Sie doppelt auf den Eintrag **Settings** des Explorers klicken, erscheint der Bereich mit den Einstellungen. Dieser zeigt im oberen Teil den Namen des Projekts sowie die Verweise an (siehe Bild 6).

Hier ist die Beschreibung einiger der Einstellungen:

- **Project: Name**: Name des Projekts. Dieser wird beispielsweise im Dateinamen der zu erstellenden DLL-

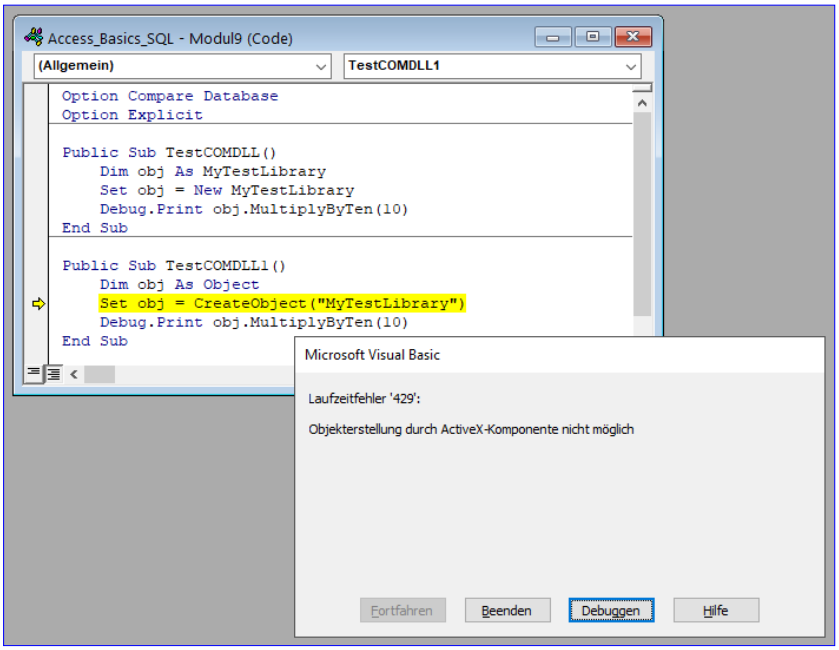


Bild 4: Fehler beim Versuch, die Klasse per Late Binding zu nutzen

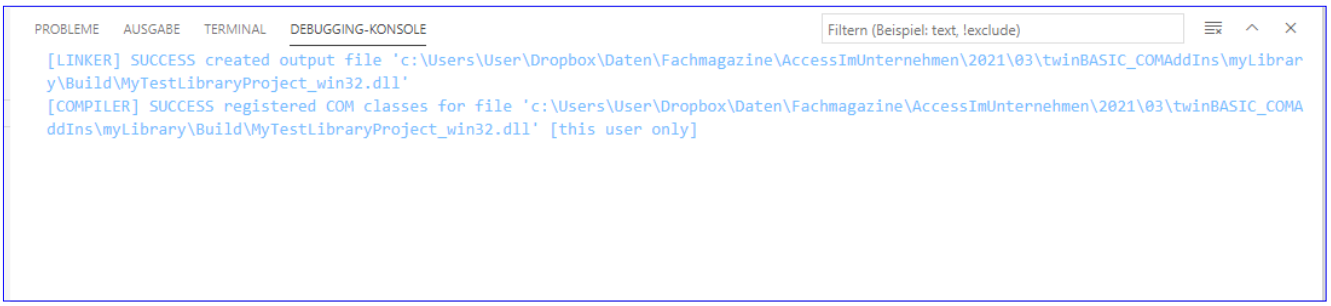


Bild 5: Ausgabe beim Build-Vorgang

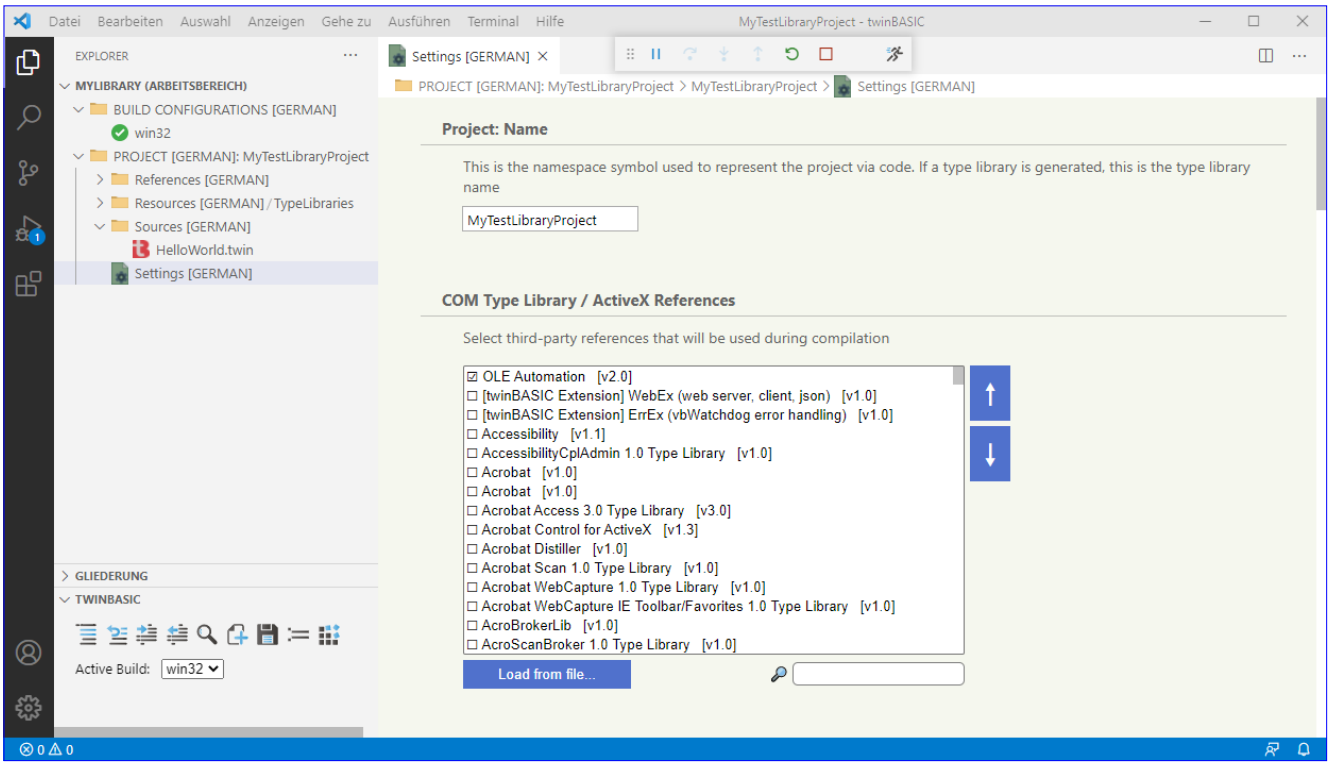


Bild 6: Einstellungen des Projekts

Datei verwendet, wenn Sie den weiter unten beschriebenen Platzhalter **ProjectName** verwenden.

- **COM Type Library/ActiveX References**: Liste der Verweise, die in dieser EXE oder DLL verwendet werden sollen.
- **Project: GUID**: GUID, die beispielsweise in der Registry für das Projekt verwendet wird.
- **Project: Build Output Path**: Gibt den Pfad an, unter dem die EXE- oder DLL-Datei erstellt werden soll. Sie können hier Platzhalter wie **{SourcePath}**, **{ProjectName}**, **{ProjectID}**, **{Architecture}** oder **{FileExtension}** verwenden.
- **Project: Build Type**: Gibt an, ob eine **ActiveX DLL** erstellt werden soll oder eine **Standard EXE**.
- **Project: Run After Build**: Nur für EXEs. Gibt an, ob diese direkt nach dem Erstellen ausgeführt werden soll.
- **Project: Description**: Text, der im **Verweise**-Dialog angezeigt wird, wenn Sie diese DLL in ein Projekt einbinden wollen.

Verweise auf andere Bibliotheken nutzen

Wie Sie oben gesehen haben, können Sie über den Bereich **COM Type Library/ActiveX References** externe Bibliotheken per Verweis einbinden. Das wollen wir nun nutzen, um beispielsweise eine Bibliothek für den Datenzugriff in einer Funktion verfügbar zu machen.

Dazu fügen wir zunächst einen Verweis auf die Bibliothek **Microsoft DAO 3.6 Object Library** hinzu. Dies erledigen wir in den Einstellungen für das Projekts in VS Code (Doppelklick auf den Eintrag **Settings**).

Um das Hinzufügen zu beschleunigen, geben wir einfach **dao** in das Suchfeld rechts unter der Liste der Bibliotheken ein. Danach können wir den gefundenen Eintrag mit einem Haken versehen (siehe Bild 7).

Debugging in twinBASIC

Der Entwickler von twinBASIC, Wayne Philips, ist vor allem für sein Fehlerbehandlungstool vbWatchDog bekannt. Kein Wunder, dass er bei der Programmierung der Entwicklungsumgebung darauf geachtet hat, dass ausreichende Debugging-Möglichkeiten vorhanden sind. Wie Sie diese nutzen, zeigt der vorliegende Beitrag. Dabei lernen Sie die verschiedenen Bereiche wie Variablen, Überwachen, Aufrufliste, Haltepunkte und den Debug-Bereich kennen.

Debugging in twinBASIC

Wenn Sie das Beispielprojekt aus dem Download geöffnet und die einzige enthaltene Datei **HelloWorld.twin** angezeigt haben, können Sie mit einem Klick auf die Schaltfläche **Ausführen und Debuggen** am linken Rand die Debugging-Bereiche der twinBASIC-Erweiterung von Visual Studio Code einblenden.

Sie können auch die Tastenkombination **Strg + Umschalt + D** nutzen, um den linken Bereich sichtbar zu machen. Hier finden Sie gleich vier Bereiche vor:

- **Variablen**: Zeigt die Inhalte der sichtbaren Variablen an.
- **Überwachen**: Hier können Sie Ausdrücke zur Überwachung hinzufügen.
- **Aufrufliste**: Hier werden die aktuell aufgerufenen Routinen abgebildet sowie die Routinen, von denen diese aufgerufen wurden.
- **Haltepunkte**: Zeigt die aktuellen Haltepunkte an. Per Klick springen Sie zur jeweiligen Zeile im Code.

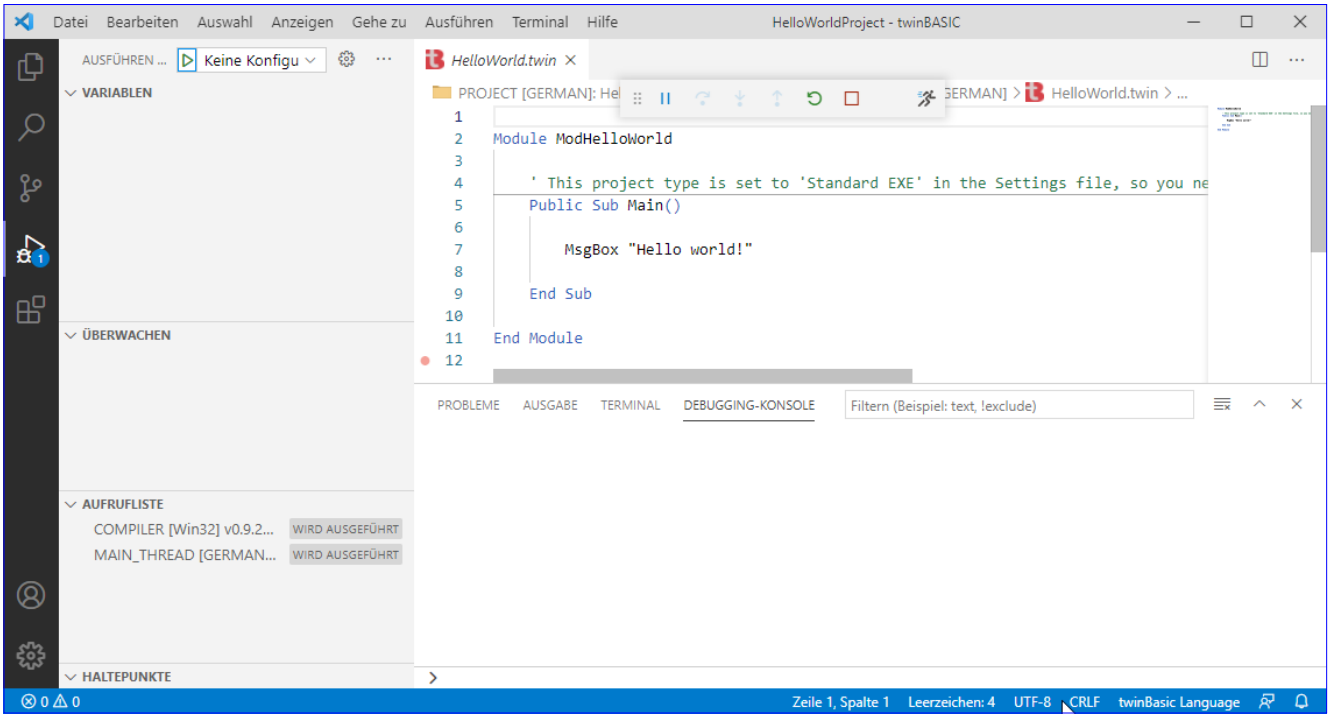


Bild 1: Die verschiedenen Debugging-Bereiche

Außerdem zeigt das Fenster im unteren Bereich noch die Debugging-Konsole an (siehe Bild 1).

Routinen starten und durchlaufen

Derzeit ist twinBASIC noch eine reine Entwicklungsumgebung für EXEs ohne Benutzeroberfläche, DLLs, COM-Add-Ins et cetera. Da findet Debugging natürlich verstärkt in der Entwicklungsumgebung statt. Standardprojekte auf Basis der **HelloWorld**-Vorlage enthalten eine **Main**-Routine, die automatisch beim Start der Anwendung aufgerufen wird.

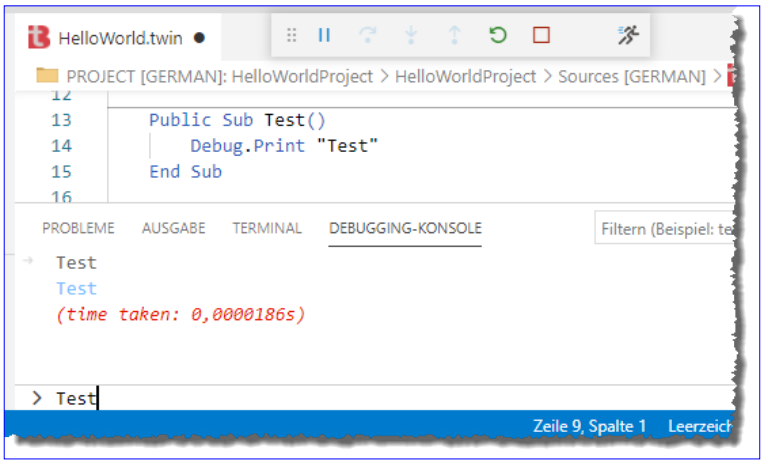


Bild 2: Starten einer Routine über den Debuggen-Bereich

Um diese Routine direkt in der Entwicklungsumgebung zu starten, haben Sie mehrere Möglichkeiten:

- über den Debugging-Bereich
- durch Markieren der Routine und Betätigen der Taste **F5** oder der Schaltfläche zum Starten der aktuell markierten Prozedur
- per Menüeintrag

Routine vom Debugging-Bereich aus aufrufen

Die erste Möglichkeit ist der Aufruf der Routine aus dem Debugging-Bereich heraus. Hier geben Sie den Namen der Routine und gegebenenfalls der Parmaeter an und betätigen die Eingabetaste.

Der Debugging-Bereich sieht etwas anders aus als im VBA-Editor. Sie können hier nicht einfach Befehle in den Bereich eintippen, sondern verwenden dazu den kleinen Bereich mit dem Größer-Zeichen (**>**). In dieser starten wir die Prozedur **Test** durch Eingabe des Routinennamens **Test** und anschließendes Betätigen der Eingabetaste.

Im Debugging-Bereich erfolgt dann die Ausgabe: erstens der Name der Routine, die gerade aufgerufen wurde, dann der Output der Routine mit der **Debug.Print**-Anweisung und schließlich die für den Aufruf benötigte Zeit (siehe Bild 2).

Sie können die **Debug.Print**-Anweisung auch direkt von der **>**-Zeile aus aufrufen. Dann wird die **Debug.Print**-Anweisung ausgeführt und samt Aufruf und Laufzeit im Debugging-Bereich ausgegeben.

Routine mit F5 starten

Wenn Sie keine Lust haben, den Routinennamen erst in den Debugging-Bereich einzutippen, können Sie auch einfach die Einfügemarke in der auszuführenden Routine platzieren und dann die Taste **F5** betätigen – oder die Schaltfläche zum Starten des Debuggings (siehe Bild 3).

Diesen Befehl können Sie auch über den Menüpunkt **Ausführen|Debugging starten** aufrufen (siehe Bild 4). Hier finden wir noch weitere Befehle, die allerdings zum Zeitpunkt der ersten Preview noch nicht funktionierten:

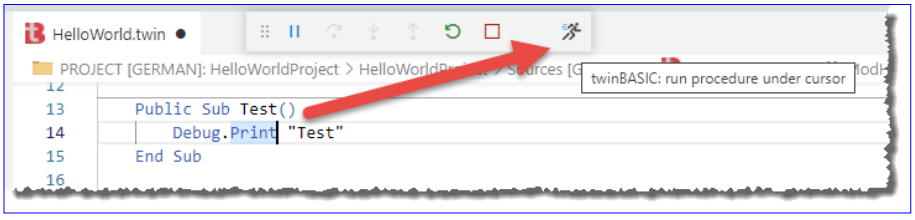


Bild 3: Starten des Debuggens per Schaltfläche

- **Debugging starten**: Startet die aktuell markierte Routine (Taste **F5**).
- **Ohne Debuggen ausführen** (Tastenkombination **Strg + F5**)

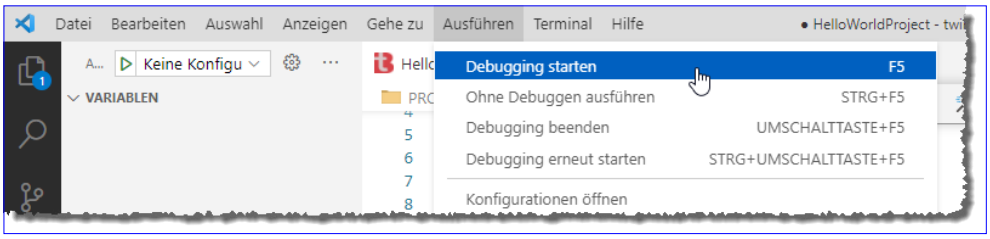


Bild 4: Debugging-Befehle per Menü

- **Debugging beenden**: Beendet das Debuggen (Tastenkombination **Umschalt + F5**)
- **Debugging erneut starten**: Startet das Debuggen erneut (Tastenkombination **Strg + Umschalt + F5**)

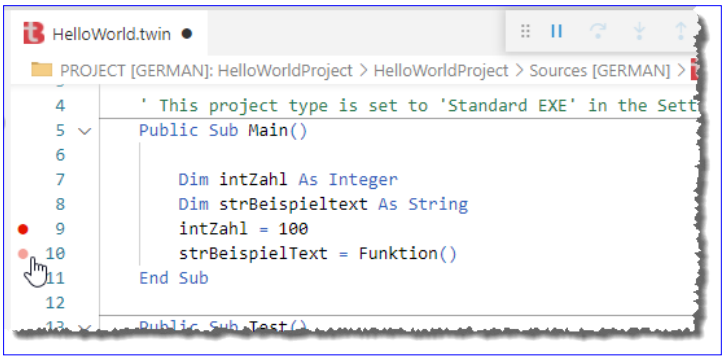


Bild 5: Setzen eines Haltepunktes

Haltepunkte setzen

Genau wie im VBA-Editor können Sie auch in VS Code unter twinBASIC Haltepunkte setzen. Dazu klicken Sie einfach links von der Zeilennummer der Codezeile, in der Sie anhalten möchten. Dort erscheint dann wie in Bild 5 ein roter Punkt.

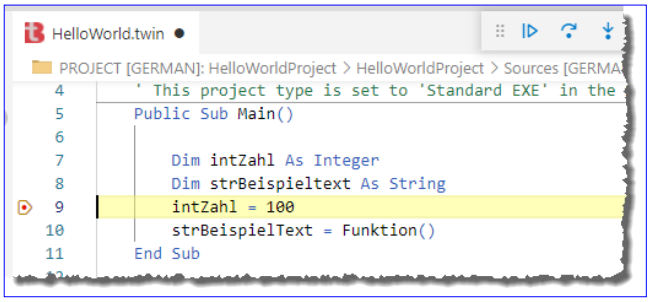


Bild 6: Erreichen eines Haltepunktes

Wenn Sie die Routine etwa mit der Taste **F5** gestartet haben, bleibt die Abarbeitung in der Zeile mit dem Haltepunkt stehen (siehe Bild 6).

Code fortsetzen

Um die Abarbeitung des Codes bis zum Ende oder bis zum nächsten Haltepunkt fortzusetzen, betätigen Sie einfach nochmal die Taste **F5**.

Code im Einzelschritt-Modus durchlaufen

Wenn Sie jedoch nach Erreichen des Haltepunktes den Code Schritt für Schritt durchlaufen wollen, um diesen zu untersuchen, betätigen Sie die Taste **F11** – also nicht mit **F8** wie im VBA-Editor.

Die Debug.Print-Methode

Das Debugging-Fenster können Sie für die Ausgabe per **Debug.Print** aus dem Code heraus genauso nutzen wie das des VBA-Editors. Sprich: Sie fügen irgendwo im Code eine **Debug.Print**-Anweisung ein, geben den auszugebenden Inhalt als Parameter an und lassen den Code laufen.

Die **Debug.Print**-Methode arbeitet allerdings nicht genau so wie die im VBA-Editor: Es ist (noch) nicht möglich, den auszugebenden Text mit dem Komma- oder Semikolon abzuschließen.

Das führt im VBA-Editor dazu, dass kein Zeilenumbruch an die Ausgabe angehängt wird und die Ausgabe der nächsten **Debug.Print**-Anweisung in der gleichen Zeile erfolgt – beim Komma durch einen Tabulator vom vorherigen Text

twinBASIC: COM-Add-Ins für Access

Neben COM-DLLs können Sie mit twinBASIC auch COM-Add-Ins programmieren, deren Funktionen dann in der Benutzeroberfläche von Access angezeigt und genutzt werden können. Die Möglichkeiten sind unbegrenzt – Sie können damit beispielsweise Ribbon-Einträge hinzufügen, die dauerhaft und unabhängig von der jeweils geöffneten Datenbank verfügbar sind und damit selbst programmierte Funktionen aufrufen. Welche Funktionen sinnvoll sind und sich hier umsetzen lassen, schauen wir uns in weiteren Beiträgen an. Dieser Beitrag beleuchtet zunächst einmal die technischen Grundlagen für die Erstellung von COM-Add-Ins für die Access-Benutzeroberfläche.

COM-Add-Ins und Access-Add-Ins

Bevor wir in die Materie einsteigen, wollen wir den Unterschied zwischen COM-Add-Ins und Access-Add-Ins klären. Access-Add-Ins sind Access-Datenbanken, die eine spezielle Dateiendung enthalten und eine Tabelle namens **USysRegInfo** mit Einträgen, welche Informationen über das Add-In zum Eintragen in die Registry enthalten.

Diese rufen Sie schließlich über den Ribbon-Eintrag **Datenbanktools|Add-Ins|Add-Ins** auf. Bild 1 zeigt einige Add-Ins in der Access-Installation des Autors dieses Beitrags. Ein Klick auf einen dieser Einträge startet diesen und zeigt beispielsweise ein Formular mit den enthaltenen Funktionen an. Es gibt noch weitere Möglichkeiten, Access-Add-Ins zu platzieren – zum Beispiel als Steuerelement-Assistent oder Formular-Assistent.

Der wichtigste Unterschied zwischen Access-Add-Ins und den hier vorgestellten COM-Add-Ins für Access ist, dass Sie für COM-Add-Ins keine speziellen Orte vorfinden, an denen Sie deren Aufruf platzieren können. Stattdessen fügen Sie einfach im gewünschten Ribbon einen Eintrag hinzu oder auch ein eigenes Ribbon, über das Sie das COM-Add-In starten können. Der zweite wichtige Unterschied ist, dass Sie Access-Add-Ins komplett mit Access erstellen. Sie brauchen dann nur die Dateiendung von **.accdb** auf **.accda** zu ändern und die bereits erwähnte Tabelle namens **USysRegInfo** mit Informationen über

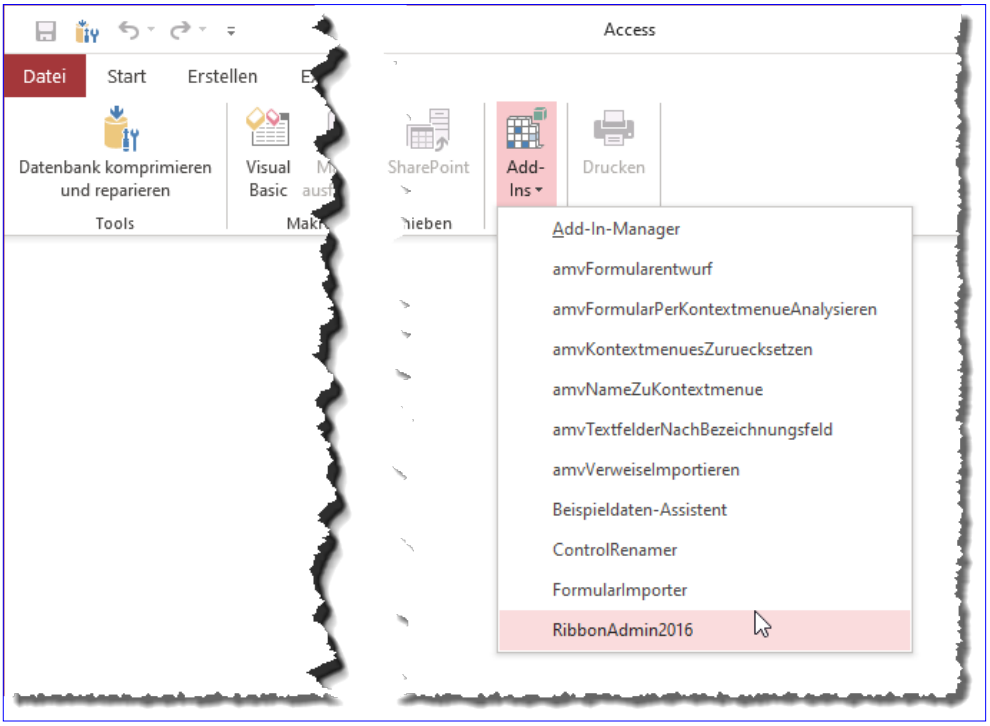


Bild 1: Access-Add-Ins

die Art des Access-Add-Ins, den Namen et cetera sowie die auszuführende Funktion hinzuzufügen.

Ein COM-Add-In hingegen können Sie nicht einfach mit Access erstellen, sondern Sie benötigen dazu eine externe Entwicklungsumgebung. Früher hat man das mit Visual Studio 6 erledigt. Auch der Nachfolger Visual Studio .NET hat im Laufe der Zeit verschiedene Tools zum Erstellen von Office-Add-Ins angeboten.

Für die Erstellung von Access-Add-Ins musste man jedoch unter Visual Studio .NET immer improvisieren: Vorlagen gab es nämlich nur für die populäreren Office-Anwendungen wie Word, Outlook oder Excel.

Kein 64-Bit unter Visual Studio 6

Der Nachteil bei der COM-Add-In-Entwicklung unter Visual Studio 6 zeigt sich aktuell zunehmend darin, dass immer mehr Installationen von Office beziehungsweise Access die 64-Bit-Version verwenden. Und Visual Studio 6 erstellt nur Projekte in der 32-Bit-Version. Insofern sind mit **twinBASIC** erstellte COM-Add-Ins auch zukunftssicherer.

COM-Add-Ins mit twinBASIC

Im April 2021 hat jedoch Wayne Philips, der bereits mit dem Fehler- und Debugging-Tool vbWatchdog von sich hören machte, eine erste Version seiner Entwicklungsumgebung für VB- und VBA-kompatible Projekte veröffentlicht. Diese basiert auf Visual Studio Code und nutzt eine eigene Erweiterung für das Programmieren von VB- und VBA-Projekten.

In Kürze soll man damit auch Anwendungen mit Benutzeroberfläche damit programmieren können, derzeit ist das aber noch nicht möglich. Für uns spielt das keine Rolle – wir schauen uns erst einmal die grundlegenden Möglichkeiten der Erstellung von COM-Add-Ins für die Access-Benutzeroberfläche an.

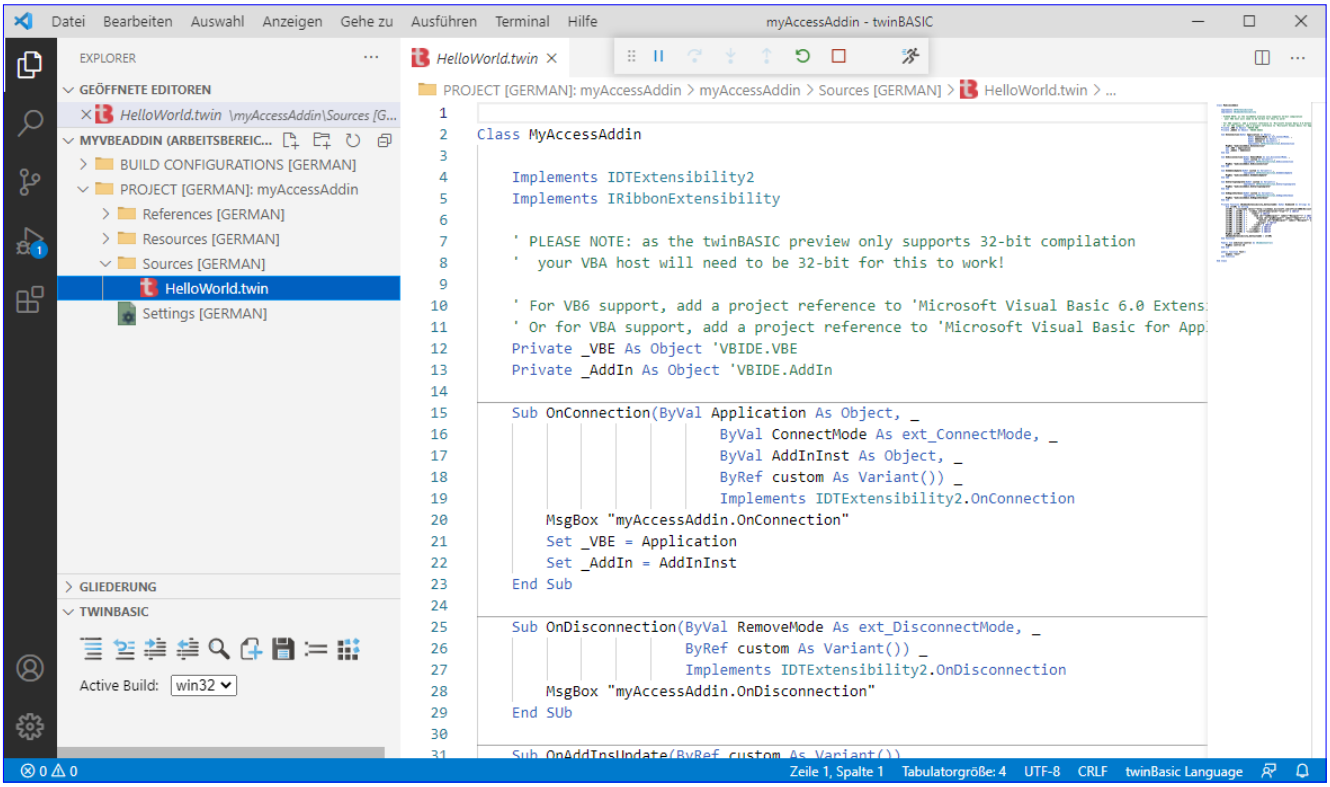


Bild 2: Aufbau eines Projekts für die Erstellung eines COM-Add-Ins

Die Grundlagen zum Einrichten von Visual Studio Code und der für die VB-Programmierung benötigten Erweiterung haben wir bereits im Beitrag **twinBASIC – VB/VBA mit moderner Umgebung** (**www.access-im-unternehmen.de/1303**) beschrieben. Im vorliegenden Beitrag bauen wir auf einem einfachen Template für ein Access-COM-Add-In auf und erläutern die einzelnen Bestandteile.

Anschließend fügen wir dem COM-Add-In einige Beispielfunktionen hinzu.

Wir starten mit einem Beispielprojekt, das Sie per Doppelklick auf die Datei **myAccessAddIn.code-workspace** starten. Dies öffnet das Projekt im twinBASIC-Editor beziehungsweise in Visual Studio Code wie in Bild 2.

```
Class MyAccessAddin
    Implements IDTExtensibility2
    Implements IRibbonExtensibility

    Private _Access As Object 'VBIDE.VBE
    Private _AddIn As Object 'VBIDE.AddIn

    Sub OnConnection(ByVal Application As Object, ByVal ConnectMode As ext_ConnectMode, _
            ByVal AddInInst As Object, ByRef custom As Variant()) _
            Implements IDTExtensibility2.OnConnection
        MsgBox "myAccessAddin.OnConnection"
        Set _Access = Application
        Set _AddIn = AddInInst
    End Sub

    Sub OnDisconnection(ByVal RemoveMode As ext_DisconnectMode, ByRef custom As Variant()) _
            Implements IDTExtensibility2.OnDisconnection
        MsgBox "myAccessAddin.OnDisconnection"
    End SUb

    Sub OnAddInsUpdate(ByRef custom As Variant()) _
            Implements IDTExtensibility2.OnAddInsUpdate
        MsgBox "myAccessAddin.OnAddInsUpdate"
    End Sub

    Sub OnStartupComplete(ByRef custom As Variant()) _
            Implements IDTExtensibility2.OnStartupComplete
        MsgBox "myAccessAddin.OnStartupComplete"
    End Sub

    Sub OnBeginShutdown(ByRef custom As Variant())_
            Implements IDTExtensibility2.OnBeginShutdown
        MsgBox "myAccessAddin.OnBeginShutdown"
    End Sub

End Class
```

Listing 1: Implementierung der Schnittstelle für COM-Add-Ins

Die hier sichtbaren Prozeduren sind die Implementierung einer Schnittstelle namens **IDExtensibility2**, die bereits seit VB6 zum Implementieren der Funktionen für COM-Add-Ins für die verschiedenen Office-Anwendungen und auch für den VBA-Editor verwendet wird (siehe Listing 1).

Um diese in der Klasse unseres twinBASIC-Projekts zu implementieren, fügen wir dem Modul **HelloWorld.twin** die folgende Zeile hinzu:

```
Implements IDTExtensibility2
```

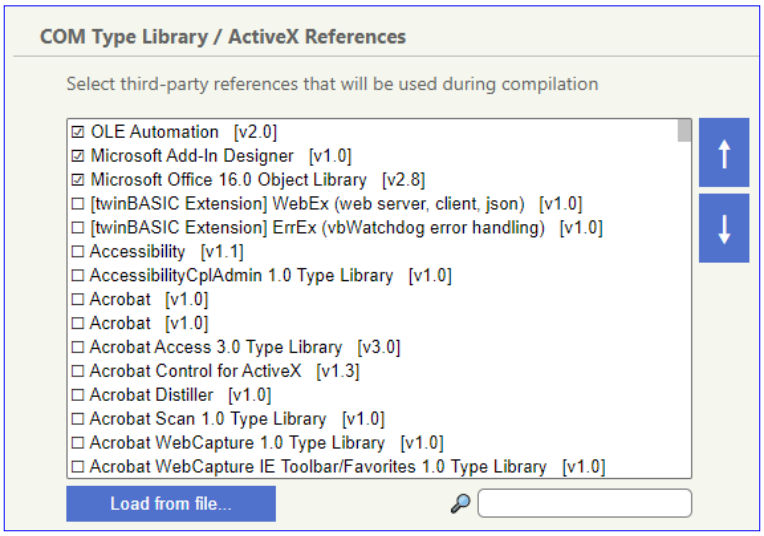


Bild 3: Notwendige Verweise für ein Access-Add-In

Diese Schnittstelle ist in der Bibliothek **Microsoft Add-In Designer** definiert, die genau wie die Implementierung der Schnittstelle bereits im Beispielprojekt enthalten ist. Die Verweise auf Bibliotheken finden Sie übrigens im Bereich **Settings** unter **COM Type Library / ActiveX References** (siehe Bild 3).

Hier haben wir auch gleich noch die Bibliothek **Microsoft Office 16.0 Object Library** hinzugefügt, denn wir wollen später auch noch ein Ribbon zum Aufrufen der Befehle des COM-Add-Ins zum Projekt hinzufügen. Diesen Verweis finden Sie schnell in der Liste, wenn Sie im Suchen-Feld Teile des Verweisnamens eingeben, beispielsweise **Office**.

Diese Schnittstelle enthält einige Ereignisprozeduren, die zu verschiedenen Gelegenheiten ausgelöst werden. Das Besondere an einer Schnittstelle ist, dass Sie alle dafür definierten Ereignisse implementieren müssen – auch wenn diese gar keine Befehle enthalten und somit nichts tun.

Die wichtigste Ereignisprozedur für diese Implementierung für uns ist in der Regel die Methode **OnConnection**. Diese wird aufgerufen, wenn Access startet und nachdem es im entsprechenden Registry-Zweig nachgesehen hat, ob irgendwelche COM-Add-Ins geladen werden müssen – dazu später mehr.

Diese Ereignisprozedur stellt zwei wichtige Parameter zur Verfügung:

- **Application**: Liefert einen Verweis auf das **Application**-Objekt der Anwendung, die das COM-Add-In aufruft.
- **AddInInst**: Liefert einen Verweis auf das **AddIn**-Objekt.

Mit dem Verweis auf das **Application**-Objekt der Anwendung können Sie dann vom Code des COM-Add-Ins aus auf die Anwendung selbst zugreifen. Um diesen Verweis während der Verwendung des COM-Add-Ins ständig im Zugriff zu haben, speichern wir es in einer Variablen, die wir wir folgt deklarieren:

```
Private _Access As Access.Application
```

Sie ahnen es bereits: Wenn wir Elemente der Access-Bibliothek verwenden, benötigen wir auch noch einen Verweis auf die Bibliothek **Microsoft Access 16.0 Object Library**. Diesen fügen Sie noch über den oben bereits erwähnten Bereich **COM Type Library / ActiveX References** hinzu.

Wichtig: Damit die enthaltenen Elemente auch im Code zur Verfügung stehen, müssen Sie die geänderten Einstellungen einmal speichern.

```
Windows Registry Editor Version 5.00

[HKEY_CURRENT_USER\SOFTWARE\Microsoft\Office\Access\Addins\MyAccessAddIn.MyAccessAddin]
"FriendlyName"="MyAccessAddin (friendly name)"
"Description"="MyAccessAddin (description)"
"LoadBehavior"=dword:00000003
```

Listing 2: Registrierungseintrag für das COM-Add-In

Das gelingt am einfachsten, wenn Sie noch während der Anzeige des Einstellungen-Bereichs die Tastenkombination **Strg + S** betätigen. Den mit dem Parameter **AddInInst** gelieferten Verweis speichern wir in der folgenden Variablen:

```
Private _AddIn As COMAddIn
```

Damit die beiden Parameter auch in diesen Variablen landen, fügen wir neben einer Meldungsanweisung die folgenden beiden Zeilen in die Ereignisprozedur **OnConnection** ein:

```
Sub OnConnection(ByVal Application As Object, _
        ByVal ConnectMode As ext_ConnectMode, _
        ByVal AddInInst As Object, _
        ByRef custom As Variant()) _
        Implements IDTExtensibility2.OnConnection
    MsgBox "myAccessAddin.OnConnection"
    Set _Access = Application
    Set _AddIn = AddInInst
End Sub
```

Mit der Meldung prüfen wir, ob das COM-Add-In überhaupt geladen wird.

Erstellen des COM-Add-Ins

Wenn Sie die **Build**-Schaltfläche in der twinBASIC-Erweiterung von Visual Studio Code verwenden, erstellt dies nicht nur die DLL-Datei mit dem COM-Add-In, sondern es werden bereits die grundlegenden Registrierungseinträge vorgenommen. Das sind die Einträge, die auch für die Registrierung einer einfachen COM-DLL notwendig sind.

Ob das COM-Add-In erfolgreich erstellt wurde, entnehmen Sie dem Bereich **Debugging-Console** von Visual Studio Code (siehe Bild 4).

Registrierung des COM-Add-Ins

Für ein COM-Add-In sind aber noch ein paar weitere Einträge notwendig, die Sie aktuell noch manuell hinzufügen müssen. Wobei manuell bedeutet, dass Sie eine Registrierungsdatei aufrufen müssen, die dann die einzelnen Schritte für Sie erledigt.

Dazu legen Sie eine Textdatei mit der Dateiendnung **.reg** an, der Sie den Inhalt aus Listing 2 hinzufügen. Nachdem Sie diese Datei erstellt und gespeichert haben, rufen Sie diese per Doppelklick auf.

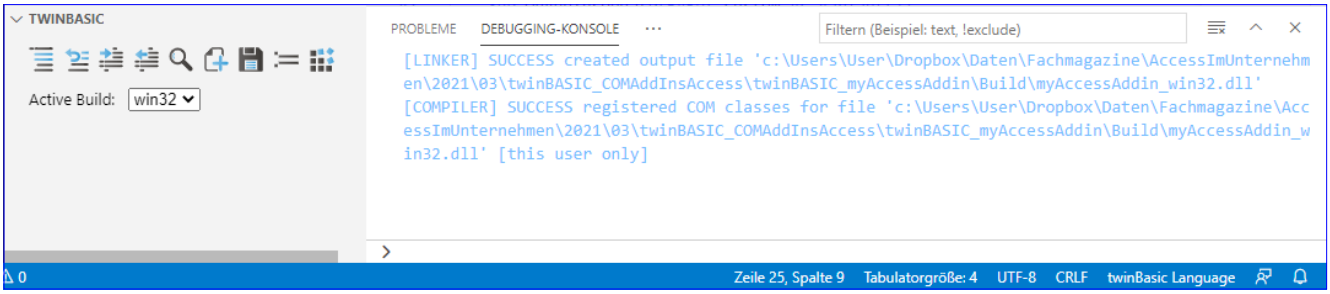


Bild 4: Ausgabe beim erfolgreichen Erstellen des COM-Add-Ins

twinBASIC: COM-Add-Ins für den VBA-Editor

Neben COM-Add-Ins für Access selbst (und natürlich auch für die übrigen Office-Anwendungen) können Sie mit twinBASIC auch COM-Add-Ins für den VBA-Editor programmieren. Dieser Beitrag beleuchtet detailliert, welche Schritte dazu notwendig sind. Dabei beginnen wir mit einer COM-Add-In-Vorlage, die wir für unseren Anwendungszweck anpassen. Anders als bei COM-Add-Ins für Access können Sie die Funktionen von COM-Add-Ins für den VBA-Editor nicht per Ribbon-Eintrag aufrufen, sondern benötigen einen Menüleisteneintrag dafür. Wir erläutern, wie dieser erzeugt wurd und wie Sie dann auf die Elemente des VBA-Editors zugreifen können.

twinBASIC

Im Beitrag **twinBASIC – VB/VBA mit moderner Umgebung** (**www.access-im-unternehmen.de/1303**) zeigen wir, was twinBASIC ist und wie Sie es installieren. twinBASIC ist eine Programmierumgebung auf Basis von Visual Studio Code mit einer Erweiterung von Wayne Philips, den manche vielleicht schon als Entwickler des Fehlerbehandlungstools **vbWatchDog** kennen. Sie können damit aktuell EXE-Dateien ohne Benutzeroberfläche erstellen oder DLLs, die Sie in Access-VBA-Projekte einbinden – siehe **COM-DLLs mit twinBASIC** (**www.access-im-unternehmen.de/1304**). Außerdem können Sie damit COM-Add-Ins erstellen. Im Beitrag **twinBASIC – COM-Add-Ins für Access** (**www.access-im-unternehmen.de/1306**) erfahren Sie, wie Sie COM-Add-Ins für Access-Anwendungen programmieren.

Wozu COM-Add-Ins für den VBA-Editor?

Wer immer nur mit Access programmiert und keine anderen Entwicklungsumgebungen als den VBA-Editor nutzt, hat sich vielleicht schon so sehr an die recht betagte Entwicklungsumgebung gewöhnt, dass er nichts vermisst. Wer jedoch gelegentlich über den Tellerrand schaut und sieht, welche Möglichkeiten andere Entwicklungsumgebungen bieten, würde sich vielleicht doch über die eine oder andere Weiterentwicklung freuen. Nun ist die Benutzeroberfläche von Access an sich seit der Version 2010 nicht mehr weiterentwickelt worden – bis auf einige unbedeutende Änderungen. Der Funktionsumfang wurde hingegen eher eingeschränkt – man denke an die Replikation und das Sicherheitssystem. Ein ähnliches Bild liefert der VBA-Editor. Genau genommen liegen hier die letzten Änderungen noch länger zurück.

Gelegentlich hat ein Entwickler Ideen, wie er den VBA-Editor verbessern könnte, was theoretisch auch mit .NET-COM-Add-Ins möglich wäre. Es gibt einige Vorlagen, mit denen man Word-, Excel- oder Outlook-Erweiterungen programmieren kann. Für Access oder auch für den VBA-Editor gibt es jedoch keine Projektvorlagen, also müsste man schon komplett selbst herausfinden, an welchen Stellen in den Vorlagen für die anderen Office-Anwendungen man schrauben müsste.

Da ist es doch erfrischend, dass Wayne Philips eine Möglichkeit bietet, COM-Add-Ins in einer neuen, alternativen Entwicklungsumgebung zum VBA-Editor und zu Visual Studio (.NET) zu entwickeln – von Visual Studio 6 reden wir erst gar nicht mehr, da damit nur die Entwicklung von Programmen auf Basis von 32-Bit möglich ist.

Mit twinBASIC können Sie jedoch, wie auf den folgenden Seiten beschrieben, auch COM-Add-Ins für den VBA-Editor entwickeln und es ist sogar geplant, dass dies auch für die 64-Bit-Version von Office möglich sein soll (aktuell ist twinBASIC noch in einem frühen Entwicklungsstadium, Features wie die Programmierung von Userinterfaces beispielsweise folgen erst später).

Projektvorlage umwandeln

Auf der Webseite von Wayne Philips finden Sie unter dem Link **https://twinbasic.com/preview.html** einige Beispiele, darunter auch das Beispiel eines COM-Add-Ins. Dieses verwenden wir als Basis für unser erstes eigenes COM-Add-In für den VBA-Editor.

Der Ordner in der Zip-Datei enthält vier Dateien:

- **myVBEAddin.code-workspace**: Datei mit den Workspace-Einstellungen
- **myVBEAddin.twinproj**: Datei mit den eigentlichen Projektdaten
- **RegisterAddin32.reg**: Datei mit Informationen zum Registrieren des COM-Add-Ins
- **UnregisterAddin32.reg**: Datei mit Informationen zum Entfernen der Registrierung des COM-Add-Ins

Die beiden ersten Dateien müssen immer den gleichen Namen aufweisen. Sie öffnen ein Projekt, indem Sie die Datei mit der Dateiendung **code-workspace** öffnen (gegebenenfalls müssen Sie die Dateiendung noch mit Visual Studio Code verknüpfen).

Dies liefert im nun erscheinenden Visual Studio Code die Ansicht aus Bild 1. Um weitere Dateien müssen Sie sich noch keine Gedanken machen, aktuell reicht eine Projektdatei mit dem Code völlig aus.

Einstellungen anpassen

Außerdem finden Sie noch Einstellungen, die wir uns nun ansehen und gegebenenfalls erweitern. Klicken Sie auf den Bereich **Settings**, finden Sie einige interessante Eigenschaften vor, die wir an dieser Stelle kurz vorstellen wollen (siehe Bild 2):

- **Project: Name**: Name des Projekts. Erscheint jedoch im Gegensatz zu COM-DLLs nirgends.

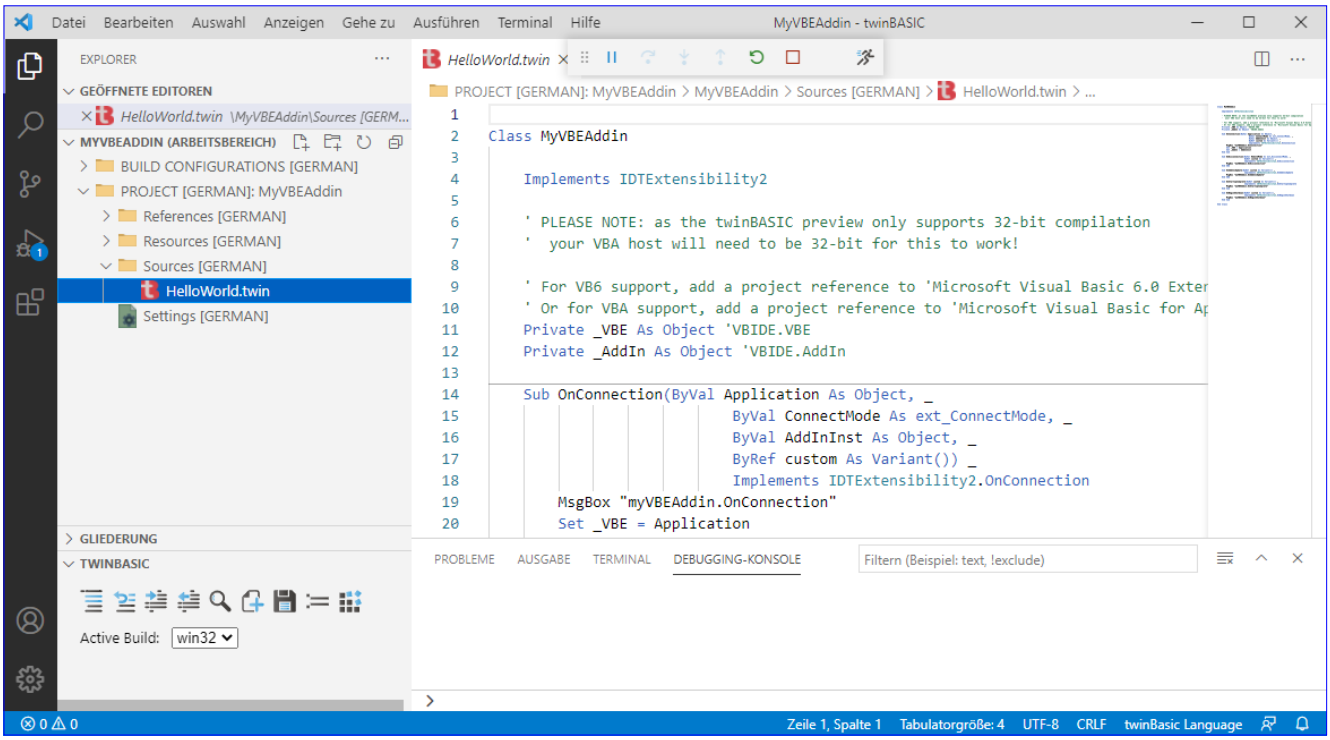


Bild 1: Entwicklungsumgebung mit dem Code des COM-Add-Ins für den VBA-Editor

- **COM Type Library / ActiveX References**: Liste der Verweise des Projekts, die Sie erweitern können.
- **Project: GUID**: GUID des Projekts. Diese sollten Sie anpassen, wenn Sie ein neues Projekt auf Basis dieser Vorlage erstellen.
- **Project: Build Output Path**: Pfad mit Platzhaltern, der angibt, wo die zu erstellende Datei gespeichert werden soll.
- **Project: Description**: Beschreibung des Projekts. Erscheint jedoch im Gegensatz zu COM-DLLs nirgends.

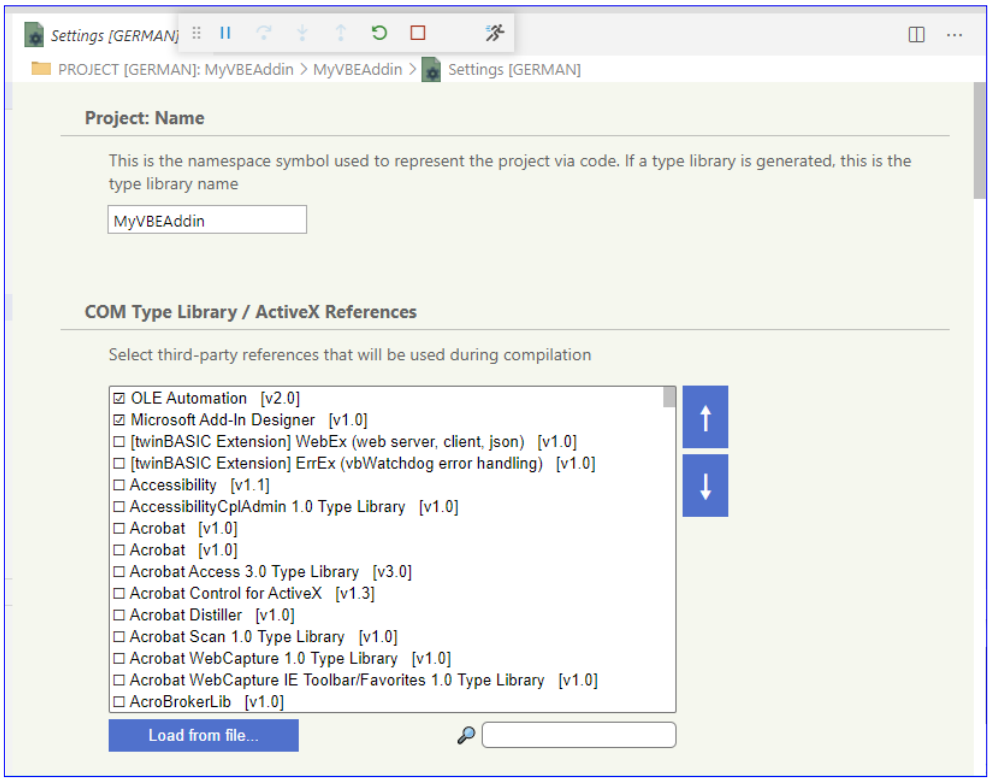


Bild 2: Einstellungen für das Projekt

Sie haben nun schon an zwei Stellen gelesen, dass Eigenschaften nirgends erscheinen. Das ist ein Unterschied zu COM-DLLs, bei denen die hier angegebenen Informationen zum Beispiel im Verweise-Dialog erscheinen. Wo und wie Sie den Namen für das COM-Add-In für den VBA-Editor einstellen und wo es dann erscheint, lesen Sie weiter unten.

Code des COM-Add-Ins für den VBA-Editor

Wenn Sie das Modul **HelloWorld.twin** ansehen, finden Sie eine Klasse vor, welche die Schnittstelle **IDTExtensibility2** implementiert. Das bedeutet, dass die **Class**-Anweisung den Zusatz **Implements IDEXTENSIBILITY2** enthält und dass die Klasse alle Elemente dieser Schnittstelle implementieren muss, was bedeutet, dass die in der Klasse enthaltenen Methoden vorliegen müssen (siehe Listing 1).

In den Methoden finden Sie zunächst nur jeweils eine **MsgBox**-Anweisung, die nur dazu dient, die grundlegende Funktion des COM-Add-Ins für den VBA-Editor zu demonstrieren, solange Sie noch keine echte Funktionalität eingebaut haben.

Außerdem enthält die Klasse die Deklaration zweier Variablen:

```
Private _VBE As Object 'VBIDE.VBE
Private _AddIn As Object 'VBIDE.AddIn
```

Diese sind noch mit dem Datentyp **Object** ausgestattet, damit das Add-In flexibel mit verschiedenen Hosts eingesetzt werden kann. Wir ersetzen dies gleich, benötigen aber zuvor zwei weitere Verweise. Diese fügen wir im Bereich Settings unter **COM Type Library / ActiveX References** zwei neue Verweise hinzu, nämlich **Microsoft Visual Basic For Applications Extensibility 5.3** und **Microsoft Office 16.0 Object Library** (siehe Bild 3). An-

```
Class MyVBEAddin
    Implements IDTExtensibility2
    Private _VBE As Object 'VBIDE.VBE
    Private _AddIn As Object 'VBIDE.AddIn

    Sub OnConnection(ByVal Application As Object, ByVal ConnectMode As ext_ConnectMode, ByVal AddInInst As Object, _
            ByRef custom As Variant()) Implements IDTExtensibility2.OnConnection
        MsgBox "myVBEAddin.OnConnection"
        Set _VBE = Application
        Set _AddIn = AddInInst
    End Sub

    Sub OnDisconnection(ByVal RemoveMode As ext_DisconnectMode, ByRef custom As Variant()) _
            Implements IDTExtensibility2.OnDisconnection
        MsgBox "myVBEAddin.OnDisconnection"
    End SUb

    Sub OnAddInsUpdate(ByRef custom As Variant()) Implements IDTExtensibility2.OnAddInsUpdate
        MsgBox "myVBEAddin.OnAddInsUpdate"
    End SUb

    Sub OnStartupComplete(ByRef custom As Variant()) Implements IDTExtensibility2.OnStartupComplete
        MsgBox "myVBEAddin.OnStartupComplete"
    End Sub

    Sub OnBeginShutdown(ByRef custom As Variant()) Implements IDTExtensibility2.OnBeginShutdown
        MsgBox "myVBEAddin.OnBeginShutdown"
    End Sub
End Class
```

Listing 1: Aufbau der Klasse des COM-Add-Ins für den VBA-Editor

schließend müssen Sie die Änderungen an den Einstellungen speichern, was Sie beispielsweise mit der Tastenkombination **Strg + S** erledigen können.

Danach können Sie die Deklaration der beiden Variablen wie folgt ändern:

```
Private _VBE As VBIDE.VBE
Private _AddIn As VBIDE.AddIn
```

Diese werden in der Prozedur **OnConnection** gefüllt:

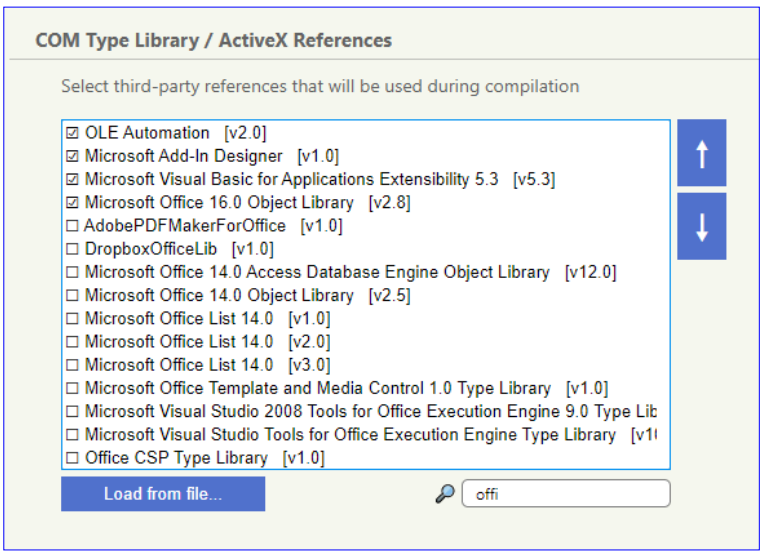


Bild 3: Hinzufügen einiger Verweise auf Projektbibliotheken

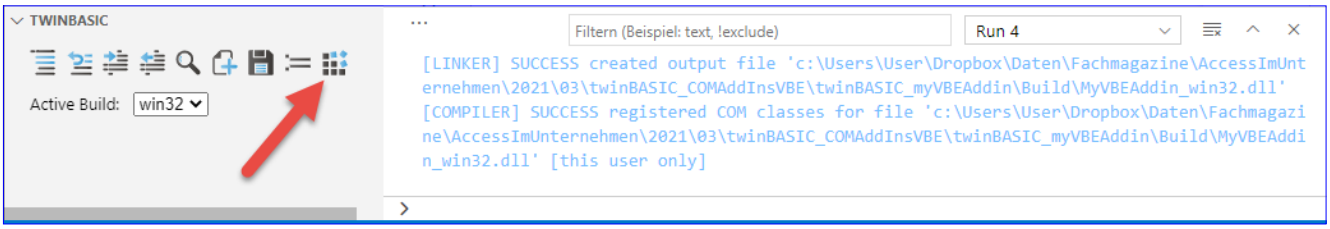


Bild 4: Erstellen des Projekts

```
Sub OnConnection(ByVal Application As Object, _
        ByVal ConnectMode As ext_ConnectMode, _
        ByVal AddInInst As Object, _
        ByRef custom As Variant()) _
        Implements IDTExtensibility2.OnConnection
    MsgBox "myVBEAddin.OnConnection"
    Set _VBE = Application
    Set _AddIn = AddInInst
End Sub
```

Erstellen des COM-Add-Ins für den VBA-Editor

Damit können wir das COM-Add-In nun erstellen. Dazu klicken Sie im Bereich **TWINBASIC** auf die in Bild 4 markierte Schaltfläche.

Die Ausgabe im Debugging-Bereich sollte danach wie in der gleichen Abbildung aussehen.

Wenn nicht, kann es daran liegen, dass das Projekt nicht kompiliert werden kann oder das noch ein VBA-Editor geöffnet ist, der das bereits erstellte COM-Add-In noch nutzt und dadurch einen Schreibschutz auf die Datei legt.

In letzterem Fall hilft meist das Schließen aller Office-Anwendungen, deren VBA-Editor innerhalb der aktuellen Session bereits einmal geöffnet war.

Registrieren des COM-Add-Ins für den VBA-Editor

Danach führen Sie die Datei **RegisterAddin32.reg** aus, um die notwendigen Registry-Einträge hinzuzufügen. Das sind nicht alle Einträge – einige Einträge legt bereits twinBASIC beim Erstellen des COM-Add-Ins an. Die Registrierungsdatei enthält lediglich diejenigen, die für das Erkennen des COM-Add-Ins beim Öffnen des VBA-Editors notwendig sind.

Anschließend ist das Add-In registriert und beim Öffnen des VBA-Editors einer beliebigen Office-Anwendung sollten die Meldungsfenster erscheinen, die in den verschiedenen Prozeduren definiert wurden.

Den Inhalt der Registrierungsdatei sehen Sie in Bild 5. Hier sollten Sie noch Anpassungen vornehmen, und zwar für die beiden Eigenschaften **FriendlyName** und **Description**.

Diese beiden tauchen nämlich im Add-In-Manager des VBA-Editors, den Sie mit dem Menüeintrag **Add-InsAdd-In-Manager...** öffnen, wie in Bild 6 auf. Hier finden Sie die Werte der Eigenschaften **FriendlyName** und **Description** wieder.

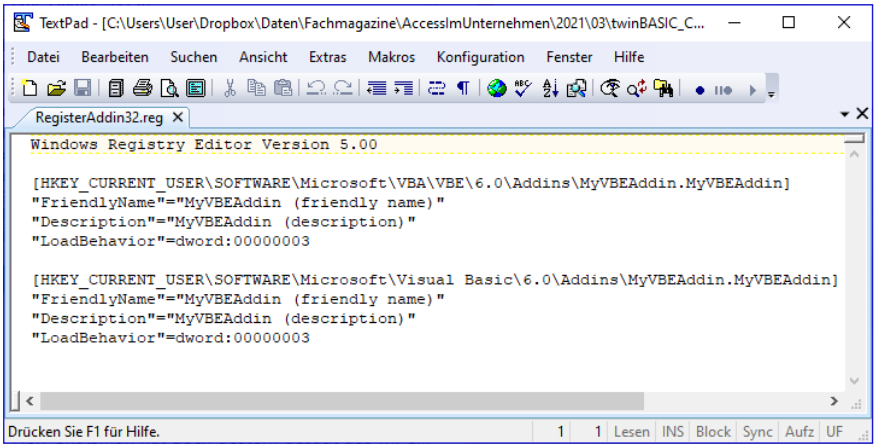


Bild 5: Registrierungsdatei für das COM-Add-In für den VBA-Editor

Schaltflächen-Assistent

Access bietet für Einsteiger bereits einen Befehlsschaltflächen-Assistent an, der grundlegende Operationen unterstützt. Die Funktionen sind allerdings stark begrenzt und das Ergebnis ist nicht an die aktuellen Möglichkeiten von Access angepasst. Wir zeigen in diesem Beitrag, wie Sie selbst einen praktischen Assistenten zum schnellen Erstellen von Schaltflächen programmieren können. Vorher schauen wir uns an, was der eingebaute Assistent kann und was wir verbessern und ergänzen wollen.

Steuerelement-Assistenten unter Access

Neben den Assistenten, die Sie über den Ribbon-Eintrag **Datenbanktools\|Add-Ins\|Add-Ins** starten können, gibt es auch einige Assistenten, die Sie beim Erstellen von Steuerelementen unterstützen sollen. Diese bietet Access standardmäßig beim Erstellen von Steuerelementen an.

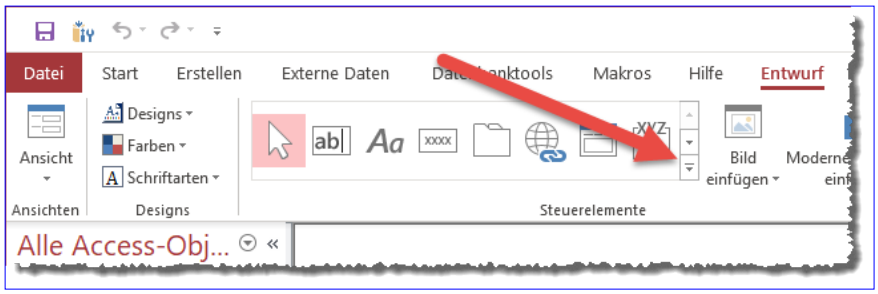


Bild 1: Ausklappen weiterer Steuerelemente und Optionen

Falls dies bei Ihnen nicht der Fall ist, wurde gegebenenfalls die entsprechende Option deaktiviert. Um die Steuerelement-Assistenten wieder zu aktivieren, öffnen Sie zunächst ein Formular in der Entwurfsansicht. Dann klicken Sie im Ribbon auf den Tab **Entwurf** und dann im Bereich **Steuerelemente** auf das Symbol zum Aufklappen weiterer Steuerelemente (siehe Bild 1).

Wenn die weiteren Optionen ausgeklappt sind, finden Sie unter der Liste aller eingebauten Steuerelemente den Eintrag **Steuerelement-Assistenten** verwenden (siehe Bild 2). Ist das Wizard-Symbol farbig hinterlegt, ist die Option aktiviert, sonst nicht. In diesem Fall aktivieren Sie die Option per Mausklick.

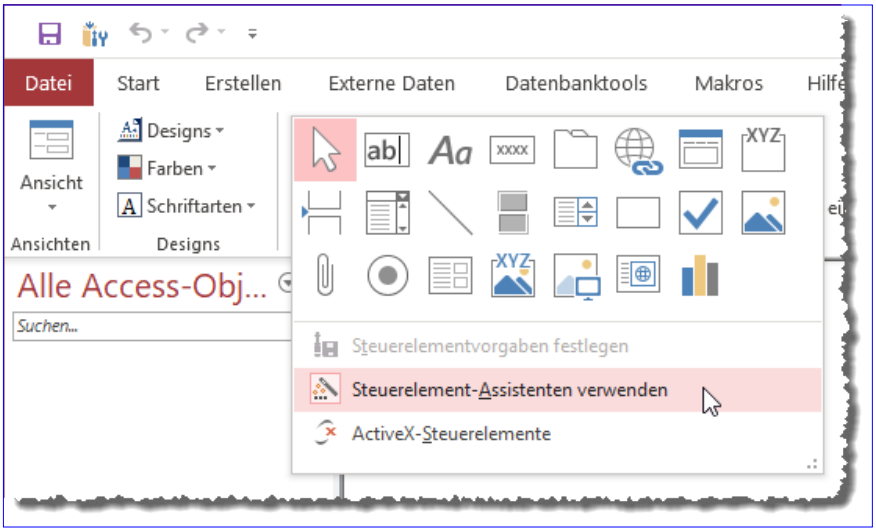


Bild 2: Aktivieren der Anzeige der Steuerelement-Assistenten

Der Befehlsschaltflächen-Assistent

Wenn Sie nun auf den Eintrag zum Anlegen einer neuen Schaltfläche klicken und dann auf die Stelle im Formularentwurf, an der die neue Schaltfläche erscheinen soll, zeigt Access den ersten Schritt des Befehlsschaltflächen-Assistenten an (siehe Bild 3). Hier können Sie im linken Listenfeld eine Kategorie auswählen

wie beispielsweise **Datensatznavigation** und im rechten Listenfeld eine der Aktionen für diese Kategorie, zum Beispiel **Datensatz suchen**.

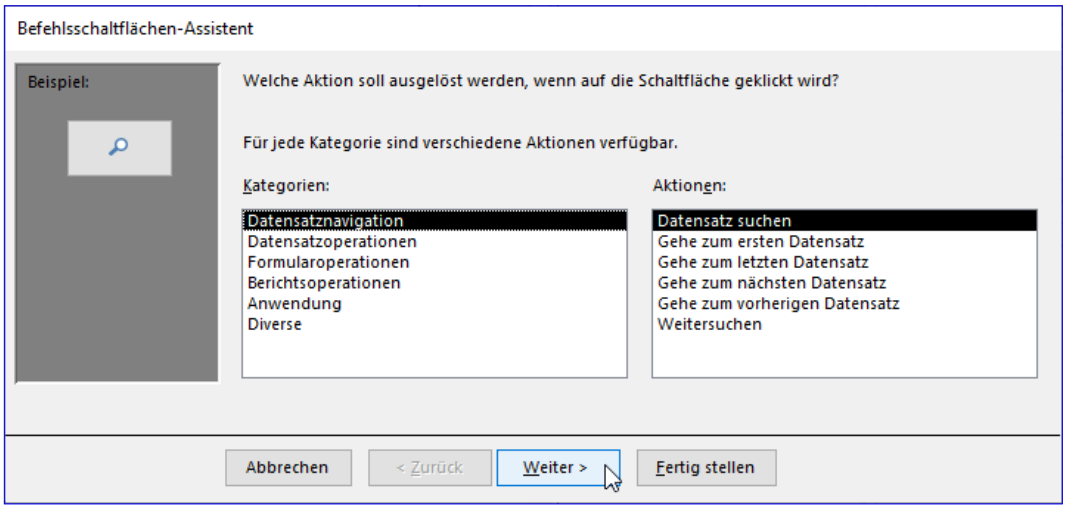


Bild 3: Erster Schritt des Befehlsschaltflächen-Assistenten

Mit einem Klick auf die Schaltfläche **Weiter** landen wir beim zweiten Schritt des Assistenten (siehe Bild 4).

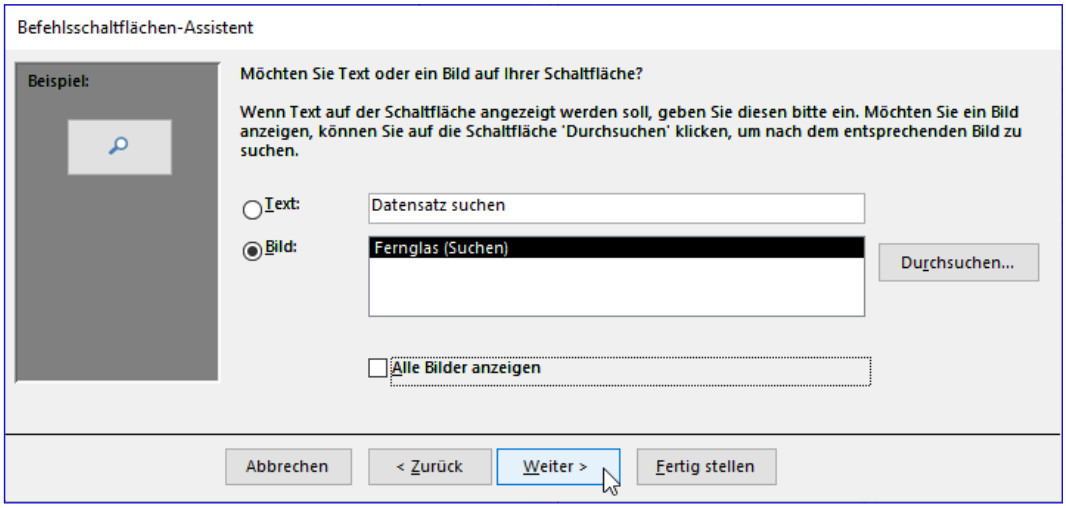


Bild 4: Auswahl von Text und Bild für die Schaltfläche

Hier finden wir zunächst die Möglichkeit, zwischen Text und Bild als Inhalt der Schaltfläche zu entscheiden. Den Text können Sie direkt neben der passenden Option in ein Textfeld eingeben beziehungsweise anpassen. Für die Auswahl eines Bildes gibt es verschiedene Möglichkeiten. Die erste ist, das angebotene Bild für diese Aktion beizubehalten. Die zweite, die Option **Alle Bilder anzeigen** zu aktivieren und im Listenfeld neben der Option **Bild** eines von mehreren Bildern zu selektieren. Wählen Sie ein anderes Bild aus, zeigt der Assistent dieses direkt in der Vorschau links an.

Sie können auch auf **Durchsuchen** klicken. Das öffnet einen **Bild auswählen**-Dialog, mit dem Sie aus dem Dateisytem **.bmp**- oder **.ico**-Dateien auswählen können. Eine so ausgewählte Datei erscheint anschließend ebenfalls in der Vorschau.

Einen Klick weiter geben Sie noch den Namen für die Befehlsschaltfläche ein. Hier schlägt der Assistent die Bezeichnung **Befehl1** vor. Wir ändern diese in **cmdDatensatzSuchen**.

Klicken Sie dann auf **Fertigstellen**, legt der Assistent die soeben konfigurierte Schaltfläche an – siehe Bild 5.

Schauen wir uns zunächst das Ergebnis des Assistenten an. Das Bild wurde direkt eingebettet. Die Schaltfläche hat den angegebenen Namen erhalten. Die Beschriftung, die ja nicht angezeigt wird, wurde entsprechend nicht geändert.

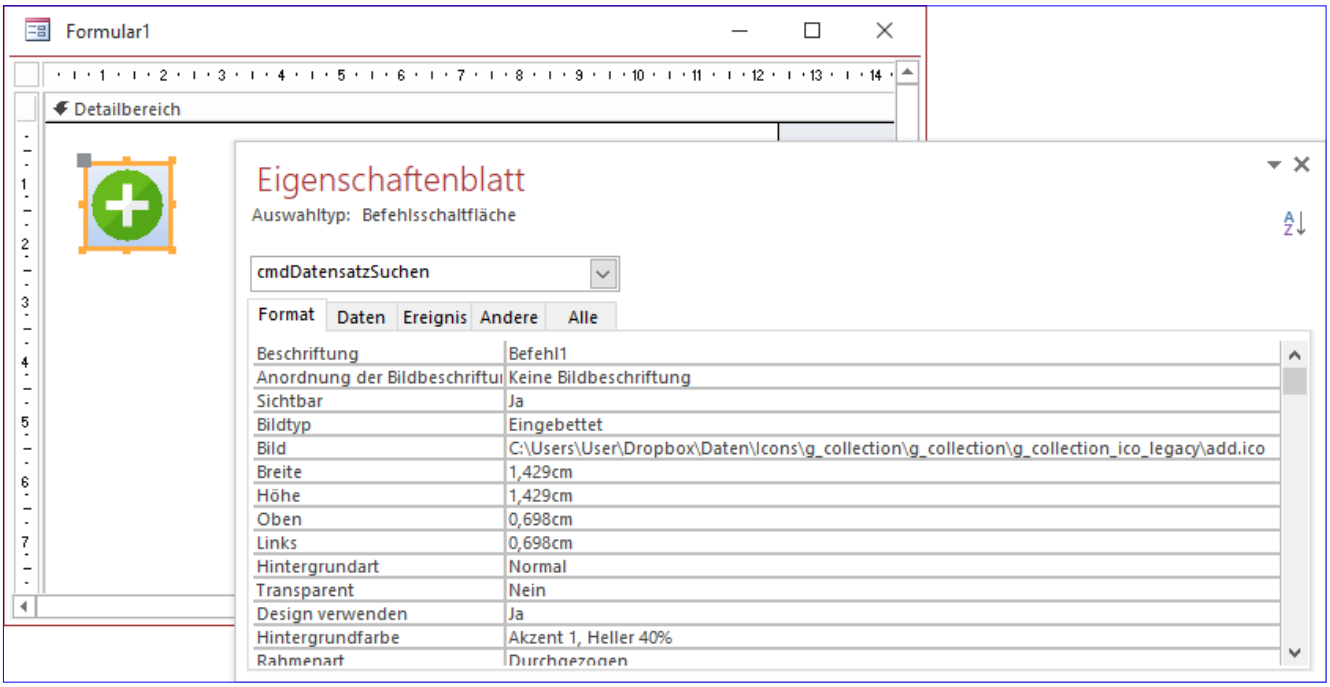


Bild 5: Eine per Befehlsschaltflächen-Assistent angelegte Schaltfläche

Ein Blick in den Bereich **Ereignis** des Eigenschaftenblatts zeigt, dass der Assistent ein Makro angelegt hat und keine VBA-Prozedur (siehe Bild 6). Wir werfen noch einen Blick auf das erstellte Makro.

Dazu klicken wir auf die Schaltfläche mit den drei Punkten neben der Eigenschaft **Beim Klicken**. Dies öffnet den Entwurf des Makros aus Bild 7, der die Makro-Variante des Suchen-Befehls definiert.

Verbesserungsideen

Beim Durchlaufen des eingebauten Befehlsschaltflächen-Assistenten sind uns die folgenden Verbesserungsvorschläge eingefallen:

- Warum sollte eine Schaltfläche nur Text oder Icon anzeigen? Wir wollen die Möglichkeit schaffen, beides anzuzeigen.

- Die Bild-Datei wird direkt in die Schaltfläche eingebettet. Access bietet aber die Möglichkeit, Bilder etwa für Schaltflächen und andere Anwendungszwecke in der Tabelle **MSysResources** zu speichern. Die dort gespeicherten Bilder können wiederverwendet werden. Das sollte unser Assistent realisieren.

- Der Benutzer hat kaum Gestaltungsmöglichkeiten bezüglich des Designs der Schaltfläche. Hier wollen wir die Vorgabe von verschiedenen Eigenschaften erlauben.

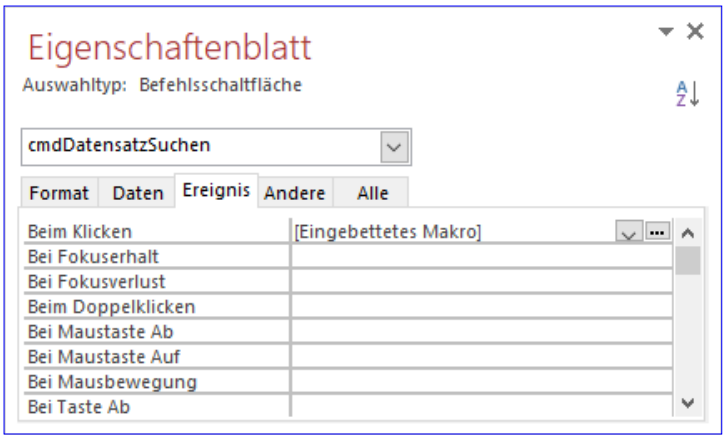


Bild 6: Der Assistent hat ein Makro für die Schaltfläche hinterlegt.

- Makros haben zwar den Vorteil, dass Einsteiger diese leichter programmieren können. Allerdings sind diese wesentlich unflexibler als VBA-Ereignisprozeduren. Daher wollen wir keine Makros verwenden.

- Der Benutzer soll verschiedene Sätze von Anweisungen anlegen können, die er beim Anlegen einer neuen Schaltfläche auswählt – oder er entscheidet sich, eine leere Ereignisprozedur zu definieren.

- Wenn der Benutzer die Beschriftung eingibt, soll nach bestimmten Kriterien direkt der Name des Steuerelements ermittelt werden.

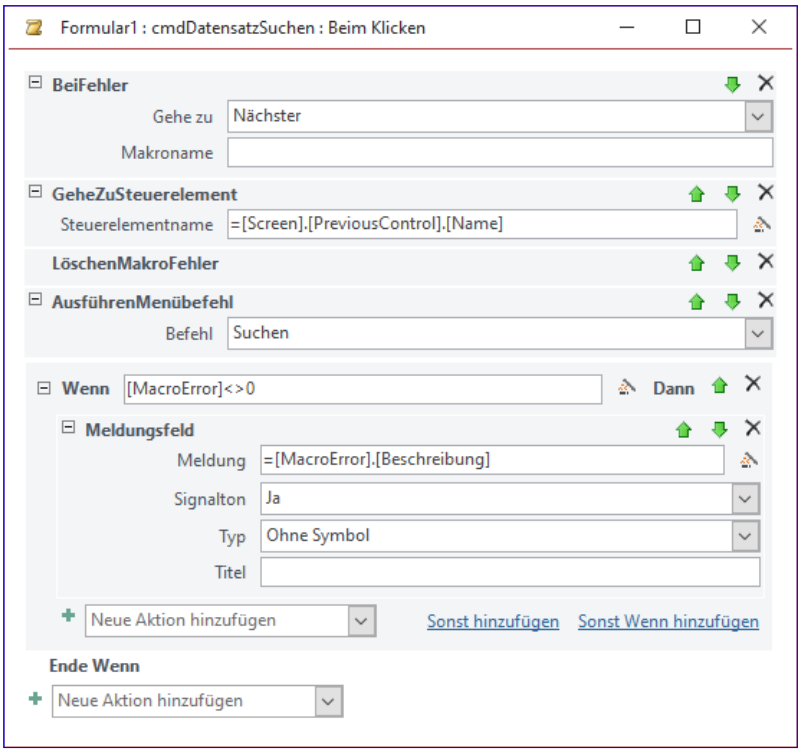


Bild 7: Das vom Assistenten angelegte Makro

Erstellen eines Steuerelement-Add-Ins

Damit machen wir uns an die Arbeit. In den ersten Schritten wollen wir den Assistenten erstellen, damit dieser beim Anlegen einer neuen Schaltfläche als Formular angezeigt wird. Dazu legen Sie zunächst eine neue Access-Datenbankdatei an und ändern ihre Dateiendung in **.accda**. Dann fügen Sie eine neue Tabelle namens **USysRegInfo** hinzu.

Diese füllen Sie wie in Bild 8. Die Daten, die in dieser Tabelle gespeichert sind, werden beim Installieren des Add-Ins über den Add-In-Manager von Access abgefragt und in die Registry eingetragen.

Die Einträge der Tabelle **USysRegInfo** haben die folgende Bedeutung. Das Feld **Subkey** enthält den Schlüssel, in

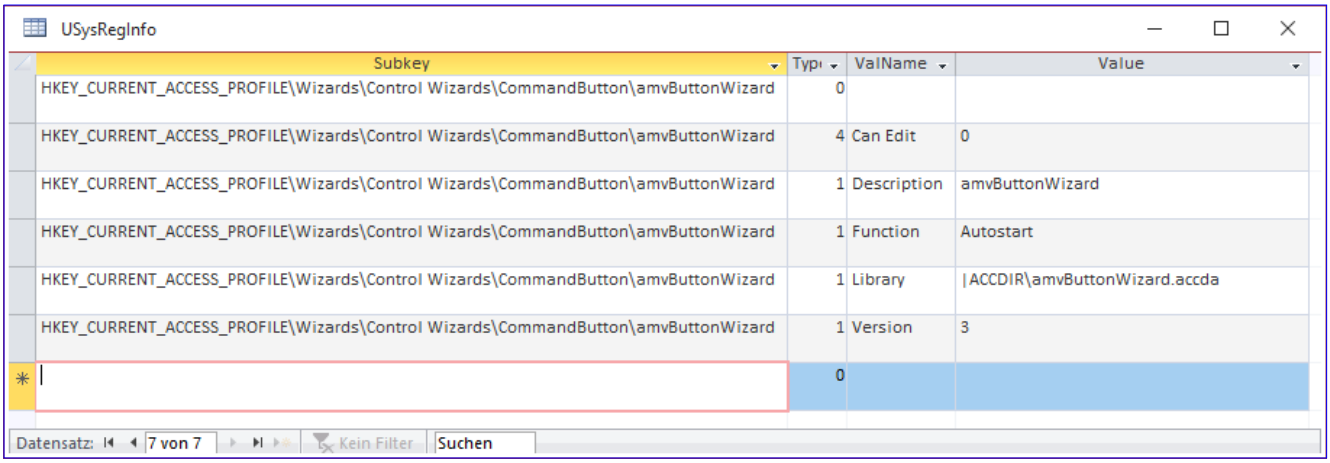


Bild 8: Die Tabelle **USysRegInfo** mit den Daten für die Registry

Schaltflächen per Code anlegen

Im Beitrag Schaltflächen-Assistent (www.access-im-unternehmen.de/1308) zeigen wir, wie Sie das Grundgerüst eines Schaltflächen-Assistenten definieren. Was Sie mit dem Schaltflächen-Assistenten anfangen können, zeigen wir Ihnen im vorliegenden Beitrag. Wir wollen zunächst das Anlegen bestimmter Standardschaltflächen erlauben. Die erste sind einfache OK- und Abbrechen-Schaltflächen. Diese Aufgabe kostet in jedem Formular, das sie neu erstellen, ein paar Minuten. Zeit, die Sie sich sparen können – indem Sie einmalig Zeit in die Entwicklung eines passenden Steuerelement-Assistenten investieren.

Ziel des Assistenten

Das Ziel des Assistenten ist das Anlegen einer neuen **OK**-Schaltfläche. Dies soll folgende Eigenschaften umfassen:

- Schaltfläche mit einem Icon (zum Beispiel grüner Haken) und dem Text **OK** als Beschriftung
- Einstellen einiger Eigenschaften wie Rahmenart, Hintergrundfarbe, Größe
- Hinzufügen einer Ereignisprozedur, die das Formular schließt oder alternativ andere Aktionen ausführt

Grundgerüst des Steuerelement-Assistenten

Das Grundgerüst des Assistenten haben wir bereits im oben genannten Beitrag beschrieben. Dieses nutzen wir als Basis für die im vorliegenden Beitrag beschriebenen Techniken.

Beim Starten des Steuerelement-Assistenten soll als Erstes ein Formular angezeigt werden, mit dem der Benutzer einige Einstellungen vornehmen kann, bevor er das Anlegen der Schaltfläche bestätigt. Dieses Formular soll auch eine Vorschau der zu erstellenden Schaltfläche anzeigen.

Formular zum Anpassen der zu erstellenden Schaltfläche

Für dieses Formular, das **frmButtonWizardOK** heißen soll, stellen wir zunächst einige Eigenschaften ein. Die Eigenschaften **Datensatzmarkierer**, **Navigationsschaltflächen**, **Trennlinien** und **Bildlaufleisten** sollen den Wert **Nein** erhalten, die Eigenschaft **Automatisch zentrieren** den Wert **Ja**.

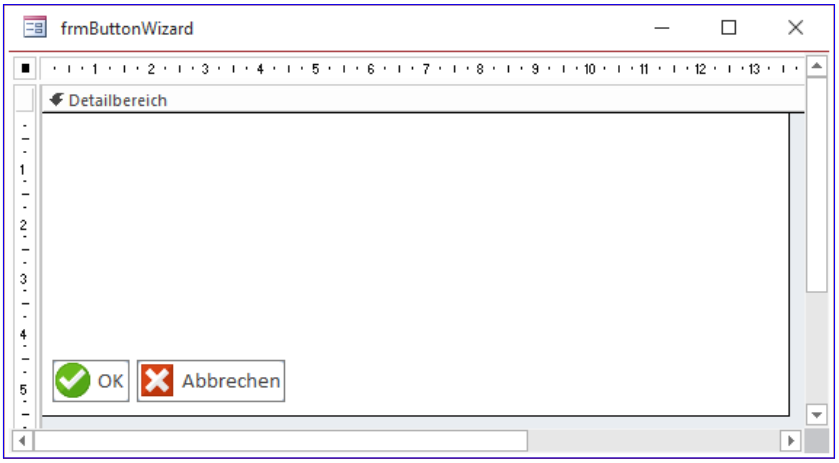


Bild 1: Erster Entwurf des Formulars zum Einstellen von Schaltflächen-Eigenschaften

Außerdem soll das Formular selbst eine **OK**-Schaltfläche sowie eine **Abbrechen**-Schaltfläche enthalten, damit der Benutzer die vorgenommenen Änderungen direkt anwenden oder auch verwerfen kann. Danach sieht das Formular wie in Bild 1 aus.

Beispiel-Schaltfläche

Nun wollen wir ja mit dem Formular festlegen, wie eine zu erstellende **OK**-Schaltfläche aussehen soll. Dazu fügen wir

Steuerelemente hinzu, mit denen wir die wichtigsten Einstellungen vornehmen. Diese haben wir in Bild 2 bereits hinzugefügt.

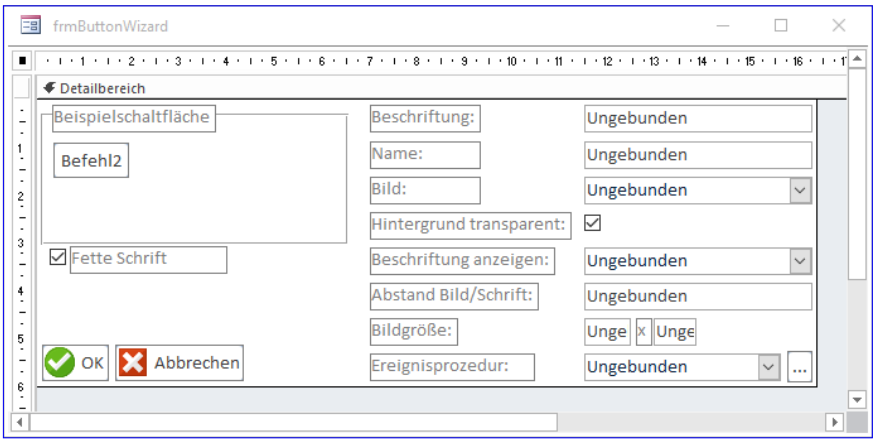


Bild 2: Formular mit Steuerelementen zum Einstellen der Schaltflächeneigenschaften

Dies sind die Steuerelemente:

- **txtBeschriftung**: Nimmt die Beschriftung für das Steuerelement entgegen.
- **txtName**: Enthält den Namen für das Steuerelement.
- **cboBilder**: Zeigt die Namen der im Add-In verfügbaren Bilder an. Diese Bilder werden in der Tabelle **MSysResources** gespeichert.
- **chkTransparent**: Gibt an, ob der Hintergrund transparent angezeigt werden soll.
- **cboBeschriftungAnzeigen**: Erlaubt die Auswahl der verschiedenen Anordnungen von Bild und Beschriftung oder auch nur die Anzeige eines Bildes.
- **txtAbstandBildSchrift**: Erlaubt das Einfügen von ein bis drei Leerzeichen zwischen Bild und Beschriftung.
- **txtBildbreite**: Breite des Bildes in Pixel
- **txtBildhoehe**: Höhe des Bildes in Pixel
- **cboEreignisprozeduren**: Dient der Auswahl der Anweisungen der Ereignisprozedur für die Schaltfläche.
- **cmdEreignisprozedurenBearbeiten**: Öffnet das Formular zum Bearbeiten der Ereignisprozeduren.

Natürlich hätten wir noch einige weitere Eigenschaften aufnehmen können, aber das würde den Rahmen dieses Beitrags sprengen.

Wichtig: Option Compare Binary statt Option Compare Database

In einer Prozedur dieses Formulars wollen wir Umlaute in Vokale umwandeln. Die Replace-Funktion ändert bei Verwendung von **Option Compare Database** im Kopf des Moduls **ä** in **ae**, aber auch **Ä** in **ae**. Damit **Ä** in **Ae** geändert wird, stellen wir im Klassenmodul des Formulars **frmButtonWizard** die Anweisung **Option Compare Binary** ein:

```
Option Compare Binary
Option Explicit
```

Hinzufügen einer Beschriftung

Das Eintragen einer Beschriftung ist der erste Schritt. Hierbei sollen direkt einige Aktionen erfolgen. Zum einen wollen wir dem Benutzer die Arbeit abnehmen, den Namen der Schaltfläche einzutragen.

Daher soll dieser dynamisch basierend auf der Beschriftung generiert werden – mehr dazu weiter unten. Außerdem soll die Größe der Vorschau immer direkt an die enthaltene Beschriftung, gegebenenfalls mit Icon, angepasst werden.

Da wir die Beschriftung auch noch in anderen Prozeduren benötigen, legen wir für diese eine modulweit deklarierte Variable namens **strBeschriftung** an:

```
Private Sub txtBeschriftung_Change()
    Dim strText As String
    Dim lngPosLeer As Long
    strText = Me!txtBeschriftung.Text
    strText = Trim(strText)
    strBeschriftung = strText
    Me!cmdBeispiel.Caption = strText
    lngPosLeer = InStr(1, strText, " ")
    Do While Not lngPosLeer = 0
        strText = Left(strText, lngPosLeer - 1) & UCase(Mid(strText, lngPosLeer + 1, 1)) & Mid(strText, lngPosLeer + 2)
        lngPosLeer = InStr(lngPosLeer + 1, strText, " ")
    Loop
    strText = UmlauteErsetzen(strText)
    Me!txtName = "cmd" & strText
    GroesseOptimieren
End Sub
```

Listing 1: Ereignisprozedur, die beim Ändern der Beschriftung ausgelöst wird

```
Dim strBeschriftung As String
```

Danach können wir in die Ereignisprozedur einsteigen, die aufgerufen wird, wenn der Benutzer den Text im Textfeld **txtBeschriftung** anpasst. Das löst mit der Eingabe eines jeden Zeichens die Prozedur aus Listing 1 aus.

Die Prozedur erfasst zunächst den aktuellen Text im Textfeld, und zwar über die Eigenschaft **Text**. Diese unterscheidet sich vom üblicherweise abgefragten Wert des Textfeldes, weil der Wert oder die Eigenschaft Value immer den zuletzt bestätigten Wert liefert. **Text** hingegen liefert den aktuell angezeigten Text. Der Text landet in der Variablen **strText**, den wir danach mit der **Trim**-Funktion von führenden und folgenden Leerzeichen befreien. Dann tragen wir den aktuellen Wert von **strText** in die Variable **strBeschriftung** ein, die wir später in weiteren Prozeduren abfragen. Außerdem landet die Beschriftung aus **strText** in der Eigenschaft **Caption** der Beispielschaltfläche **cmdBeispiel**, damit der Text direkt angezeigt wird.

CamelCase für den Schaltflächennamen

Danach folgen die Schritte, in denen wir den Namen für die Schaltfläche aus dem aktuellen Text für die Beschreibung ableiten. Der Schaltflächenname soll aus dem Präfix **cmd** plus der Beschriftung bestehen, wobei wir aus der Beschriftung die Leerzeichen entfernen, neue Wörter in der CamelCase-Notation schreiben und Umlaute in die entsprechenden Vokale umwandeln. Aus **Jetzt löschen** wird also der Schaltflächenname **cmdJetztLoeschen**.

Dazu ermitteln wir zuerst die Position des ersten Leerzeichens und speichern diese in der Variablen **lngPosLeer**. Ist diese nicht **0**, kommt also mindestens ein Leerzeichen in der Beschriftung vor, steigen wir in eine **Do While**-Schleife ein.

Diese stellt den Wert in **strText** neu zusammen, und zwar aus den Zeichen bis zum ersten Leerzeichen (**Left(strText, lngPosLeer - 1)**), dem ersten Buchstaben des Wortes nach dem Leerzeichen als Großbuchstaben (**UCase(Mid(strText, lngPosLeer + 1, 1))**) sowie dem Rest des Textes nach diesem Buchstaben (**Mid(strText, lngPosLeer + 2)**).

Aus **Jetzt alles schließen** wird dann **JetztAlles schließen**. Danach prüfen wir innerhalb der **Do While**-Schleife, ob es noch ein Leerzeichen im Ausdruck gibt. Das ist bei unserem Beispiel der Fall, also ist **lngPosLeer** noch nicht

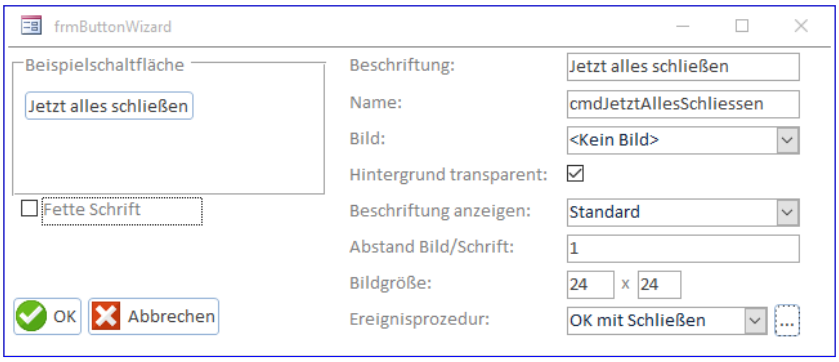


Bild 3: Zwischenergebnis nach der Eingabe der Beschriftung

0 und wir durchlaufen die **Do While**-Schleife ein zweites Mal. Danach hat **strText** den Inhalt **JetztAllesSchließen**.

Damit verlässt die Prozedur die Schleife und ruft die Funktion **UmlauteErsetzen** auf, die wie folgt aussieht:

```
Public Function UmlauteErsetzen(ByVal strText As String) _
                                                As String
    strText = Replace(strText, "ä", "ae")
    strText = Replace(strText, "ö", "oe")
    strText = Replace(strText, "ü", "ue")
    strText = Replace(strText, "Ä", "Ae")
    strText = Replace(strText, "Ö", "Oe")
    strText = Replace(strText, "Ü", "Ue")
    strText = Replace(strText, "ß", "ss")
    UmlauteErsetzen = strText
End Function
```

Die Funktion ersetzt alle Umlaute durch Vokale und **ß** durch **ss**, was in unserem Beispiel zum Ergebnis **JetztAllesSchliessen** führt. Dem stellt die Ereignisprozedur noch das Präfix **cmd** voran: **cmdJetztAllesSchliessen**. Schließlich wird die Prozedur **GroesseOptimieren** aufgerufen, welche die Größe der Beispielschaltfläche anpasst.

Die Prozedur **GroesseOptimieren** schauen wir uns weiter unten an, da diese noch weitere Steuerelemente berücksichtigt, die wir noch nicht erstellt haben. In Bild 3 sehen Sie einen Zwischenstand.

Namen der Schaltfläche ändern

Wenn Sie den nun automatisch aus der Beschriftung der Schaltfläche abgeleiteten Namen ändern möchten, können Sie dies problemlos tun. Beachten Sie nur, dass dieser bei erneuter Änderung der Beschriftung wieder von der Beschriftung abgeleitet wird. Sie können bei Bedarf eine Option einbauen, welche die nachträgliche Änderung des Namens verhindert, wenn dieser einmal festgelegt wurde. Wir wollen es an dieser Stelle allerdings nicht zu kompliziert machen.

Bild auswählen

Bei der Bereitstellung der Bilder wollen wir eine eingebaute Funktion von Access nutzen. Dabei können Sie in der Entwurfsansicht eines Formulars den Ribbon-Befehl **Entwurf|Steuerelemente|Bild einfügen|Durchsuchen...** nutzen, um Bilder zur Datenbank hinzuzufügen (siehe Bild 4).

Diese können wir beim Programmieren der **.accda**-Datenbank zunächst nutzen, um ein paar als Icon zu verwendende Bilddateien in der Datenbank zu speichern. Die Bilder landen dann in einem Anlagefeld in der Tabelle **MSysResources**.

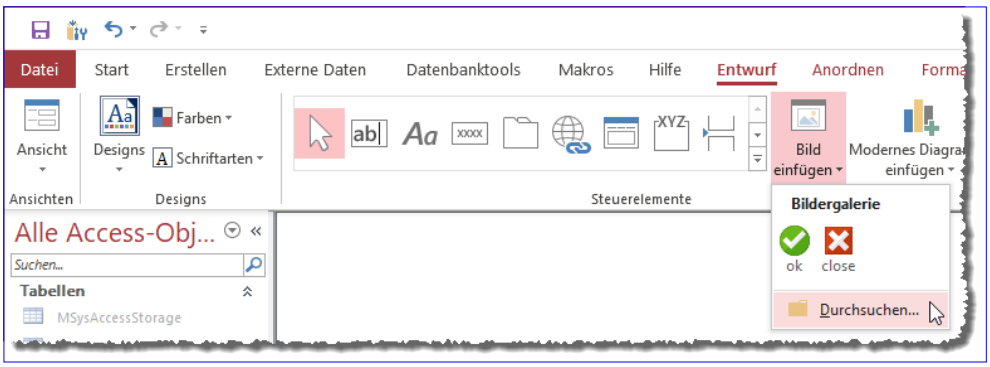


Bild 4: Auswählen von Bildern für die aktuelle Datenbank

Diese Tabelle verwenden wir in der Abfrage für die Datensatzherkunft des Kombinationsfeldes **cboBilder** als Datenquelle. Und wir setzen noch einen drauf, denn wir wollen auch einen Eintrag mit dem Text **<Kein Bild>** anzeigen. Daher benötigen wir eine **UNION**-Abfrage, deren SQL-Anweisung wie folgt aussieht:

```
SELECT 0 AS ID, "<Kein Bild>" AS Name, "" AS Type
FROM MSysResources
UNION
SELECT Id, Name, Type
FROM MSysResources WHERE Type="img"
```

Dies liefert ein Ergebnis wie in Bild 5.

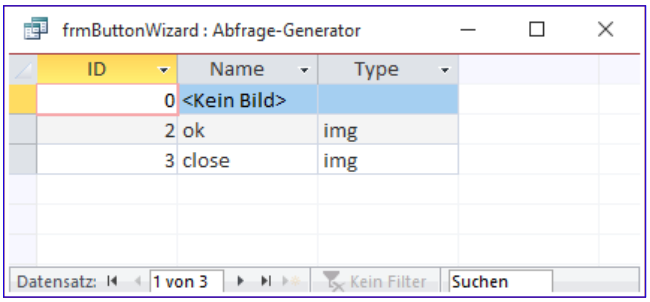


Bild 5: Datensatzherkunft für das Kombinationsfeld **cboBilder**

Für das Kombinationsfeld **cboBilder** stellen wir die Eigenschaften **Spaltenanzahl** auf **2** und **Spaltenbreiten** auf **0cm** ein. Danach können Sie mit dem Kombinationsfeld die Bilder wie in Bild 6 auswählen. Das funktioniert für den Moment, später werden wir hier noch nachjustieren müssen.

Das Aktualisieren des Kombinationsfeldes löst das Ereignis **Nach Aktualisierung** mit der folgenden Ereignisprozedur aus. Diese ruft zwei Prozeduren auf: **BildEinstellen**, um das neue Bild festzulegen, und die bereits oben einmal aufgerufene Prozedur **GroesseOptimieren**. Diese soll die Größe an den geänderten Inhalt anpassen:

```
Private Sub cboBilder_AfterUpdate()
    BildEinstellen
    GroesseOptimieren
End Sub
```

Die Prozedur **BildEinstellen** prüft, ob das Kombinationsfeld **cboBilder** den Wert **0** hat, was dem Text **<Kein Bild>** entspricht. In diesem Fall stellt es die Eigenschaft **Picture** der Beispielschaltfläche auf eine leere Zeichenkette ein. In allen anderen Fällen weist die Prozedur der Eigenschaft **Picture** den Namen des Bildes aus der Tabelle **MSysResources** zu, also beispielsweise **ok** oder **close**:

```
Private Sub BildEinstellen()
    If Me!cboBilder = 0 Then
        Me!cmdBeispiel.Picture = ""
    Else
        Me!cmdBeispiel.Picture = Me!cboBilder.Column(1)
    End If
End Sub
```

Hintergrund transparent schalten

Das Kontrollkästchen **chkTransparent** soll die Eigenschaft **BackStyle** entweder auf den Wert **0** einstellen (transparent) oder auf den Wert **1** (ausgefüllt).

Dies wird direkt nach dem Wählen eines neuen Wertes auf die Beispielschaltfläche angewendet:

```
Private Sub chkTransparent_AfterUpdate()
    If Me!chkTransparent Then
        Me!cmdBeispiel.BackStyle = 0
    Else
```

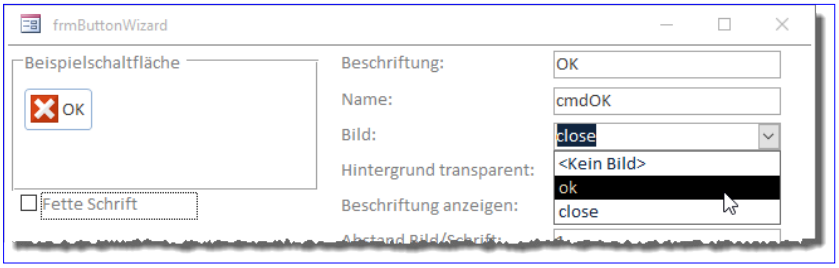


Bild 6: Auswahl von Bildern mit dem Kombinationsfeld **cboBilder**

```
        Me!cmdBeispiel.BackStyle = 1
    End If
End Sub
```

Beschriftungsanzeige einstellen

Das Kombinationsfeld **cboBeschriftungAnzeigen** erlaubt es, einen der sechs vorgesehenen Werte für die Eigenschaft **Anordnung der Bildbeschriftung** vorzunehmen. Die sechs Werte tragen wir direkt in die Eigenschaft **Datensatzherkunft** des Kombinationsfeldes ein:

```
"0";"Keine Beschriftung";"1";"Standard";2;"Beschriftung oben";3;"Beschriftung unten";4;"Beschriftung links";5;"Beschriftung rechts"
```

Damit nur die Texte und nicht die Zahlen angezeigt werden und die Liste überhaupt korrekt verarbeitet wird, legen Sie die Eigenschaft **Herkunftstyp** auf **Wertliste**, **Spaltenanzahl** auf **2** und **Spaltenbreiten** auf **0cm** fest.

Nach der Auswahl eines der Werte wird der passende Zahlenwert der VBA-Eigenschaft **PictureCaptionArrangement** zugewiesen. Anschließend erfolgt der obligatorische Aufruf der Prozedur **GroesseOptimieren**:

```
Private Sub cboBeschriftungAnzeigen_AfterUpdate()
    Me!cmdBeispiel.PictureCaptionArrangement = _
                                Me!cboBeschriftungAnzeigen
    GroesseOptimieren
End Sub
```

Abstand zwischen Bild und Schrift einstellen

Wenn Sie einer Schaltfläche ein Icon und einen Text zuweisen, kann es sein, dass der Platz zwischen beiden Elementen etwas zu klein ist. Dann bietet es sich an, dem Text ein oder mehrere Leerzeichen voranzustellen. Das erledigen wir mit dem Kombinationsfeld **cboAbstandBildSchrift**, das die Werte **0;1;2;3** als Wertliste anzeigt. Nach der Auswahl eines Wertes mit diesem Feld feuert das Ereignis **Nach Aktualisierung** und löst die folgende Ereignisprozedur aus:

```
Private Sub cboAbstandBildSchrift_AfterUpdate()
    Dim strLeerzeichen As String
    strLeerzeichen = Space(Me!cboAbstandBildSchrift)
    Me!cmdBeispiel.Caption = _
                        strLeerzeichen & strBeschriftung
    GroesseOptimieren
End Sub
```

Die Prozedur speichert eine Zeichenkette mit der dem Zahlenwert entsprechen Menge Leerzeichen in der Variablen **strLeerzeichen**. Dann fügt sie diese vor der Beschriftung aus **strBeschriftung** ein und weist das Ergebnis der Eigenschaft **Caption** der Beispielschaltfläche zu. Schließlich ruft auch diese Prozedur die Routine **GroesseOptimieren** auf.

Bildgröße eingeben

Wir könnten theoretisch die zu verwendenden Bilder analysieren und die Größe per Code ermitteln. Das ist uns aber an dieser Stelle zu aufwendig, weshalb wir dem Benutzer selbst die Möglichkeit geben, die Bildabmessungen in die beiden Textfelder **txtBildbreite** und **txtBildhoehe** einzugeben.

Die Eingabe löst die folgenden Prozeduren aus, die jeweils die Routine **GroesseOptimieren** aufrufen:

```
Private Sub txtBildbreite_AfterUpdate()
    GroesseOptimieren
End Sub

Private Sub txtBildhoehe_AfterUpdate()
    GroesseOptimieren
End Sub
```

Fett dargestellte Beschriftung

Das Kontrollkästchen **chkFetteSchrift** gibt an, ob die Beschriftung der Schaltfläche fett angezeigt werden soll. Die Änderung des Wertes des Kontrollkästchens löst diese Prozedur aus, die das Format der Beschriftung der Beispielschaltfläche anpasst:

```
Private Sub chkFetteSchrift_AfterUpdate()
    Me!cmdBeispiel.FontBold = Me!chkFet-
teSchrift
    GroesseOptimieren
End Sub
```

Ereignisprozedur auswählen

Schließlich wollen wir dem Benutzer noch die Möglichkeit geben, den Inhalt der Ereignisprozedur, die durch das Ereignis **Beim Klicken** ausgelöst wird, festzulegen. Dazu haben wir eine Tabelle vorgesehen, um verschiedene Anweisungen zu speichern.

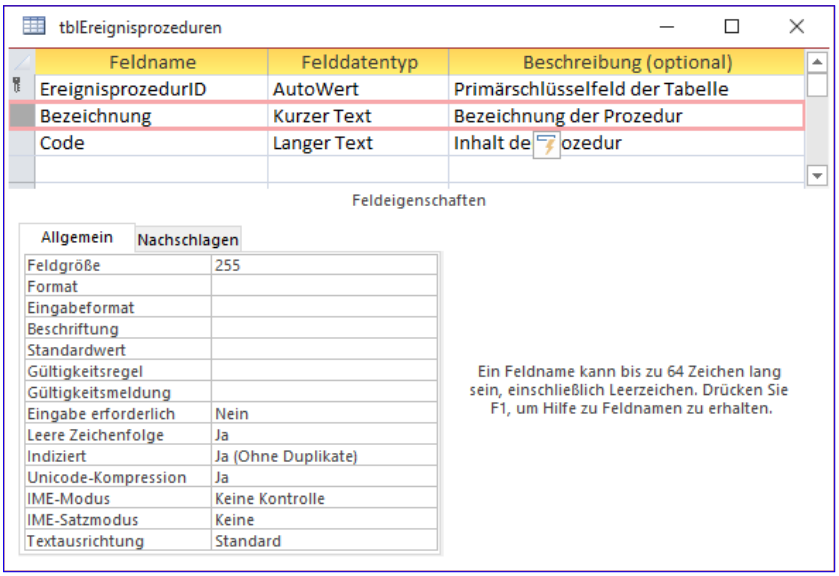


Bild 7: Entwurf der Tabelle zum Speichern der Prozedurinhalte

Diese Tabelle enthält die Felder **EreignisprozedurID**, **Bezeichnung** und **Code** und sieht im Entwurf wie in Bild 7 aus. Für das Feld **Bezeichnung** haben wir einen eindeutigen Index festgelegt, da jede Bezeichnung nur einmal vorkommen darf.

Für das Kombinationsfeld **cboEreignisprozeduren** legen wir die folgende Abfrage auf Basis der Tabelle **tblEreignisprozeduren** als Datensatzherkunft fest:

```
SELECT 0 AS EreignisprozedurID,
'<Keine Ereignisprozedur>' AS Bezeichnung
FROM tblEreignisprozeduren
UNION
SELECT EreignisprozedurID, Bezeichnung
FROM tblEreignisprozeduren
ORDER BY Bezeichnung;
```

```
Private Sub cmdEreignisprozedurenBearbeiten_Click()
    Dim lngEreignisprozedurID As Long
    If Nz(Me!cboEreignisprozeduren, 0) = 0 Then
        DoCmd.OpenForm "frmEreignisprozeduren", WindowMode:=acDialog, datamode:=acFormAdd
    Else
        DoCmd.OpenForm "frmEreignisprozeduren", WindowMode:=acDialog, WhereCondition:="EreignisprozedurID = " _
            & Me!cboEreignisprozeduren, datamode:=acFormEdit
    End If
    If IstFormularGeoeffnet("frmEreignisprozeduren") Then
        lngEreignisprozedurID = Nz(Forms!frmEreignisprozeduren!EreignisprozedurID, 0)
        If Not lngEreignisprozedurID = 0 Then
            Me!cboEreignisprozeduren = lngEreignisprozedurID
        End If
        DoCmd.Close acForm, "frmEreignisprozeduren"
    End If
End Sub
```

Listing 2: Ereignisprozedur zum Anzeigen des Dialogs für die Verwaltung von Ereignisprozeduren

Auch hier nutzen wir eine **UNION**-Abfrage, um einen zusätzlichen Eintrag zum Kombinationsfeld hinzuzufügen. Diese enthält den Wert **0** und den Text **<Keine Ereignisprozedur>**.

Und wieder stellen wir die beiden Eigenschaften **Spaltenanzahl** und **Spaltenbreiten** auf die Werte **2** und **0cm** ein. Wenn der Benutzer einen Eintrag im Kombinationsfeld auswählt, geschieht ausnahmsweise einmal nichts – die Ereignisprozedur wird erst beim Abschließen des Assistenten zur neu erstellten Schaltfläche hinzugefügt.

Mit der Schaltfläche **cmdEreignisprozedurenBearbeiten** soll der Benutzer ein Formular zum Verwalten der Ereignisprozeduren öffnen können. Es löst die Prozedur aus Listing 2 aus.

Die Prozedur prüft zunächst, ob der Eintrag **<Keine Ereignisprozedur>** ausgewählt ist, also der Wert **0**. Falls ja, wird ein Formular namens **frmEreignisprozeduren** als modaler Dialog im Modus zum Hinzufügen eines Datensatzes geöffnet. Anderenfalls erfolgt das Öffnen mit **DoCmd.OpenForm** im Bearbeitungsmodus und eine **WhereCondition** übergibt den Primärschlüsselwert des aktuell im Kombinationsfeld **cboEreignisprozeduren** gewählten Eintrags als Kriterium an das Formular.

In beiden Fällen läuft die Prozedur durch das Öffnen des Formulars **frmEreignisprozeduren** als modaler Dialog erst dann weiter, wenn das Formular geschlossen oder unsichtbar gemacht wird. Dann prüft die Prozedur, ob das Formular noch geöffnet ist. Falls ja, übernimmt die Prozedur den aktuell im Formular angezeigten Datensatz in das Kombinationsfeld **cboEreignisprozeduren**. Falls kein Datensatz ausgewählt ist, geschieht nichts. Das Formular wird dann einfach geschlossen.

Wenn das Formular schon geschlossen ist, geht die aufrufende Prozedur davon aus, dass der Benutzer das Formular **frmEreignisprozeduren** mit der **Abbrechen**-Schaltfläche geschlossen hat. Das Kombinationsfeld wird dann ebenfalls nicht aktualisiert.

Ereignisprozeduren verwalten

Zum Verwalten der Ereignisprozeduren legen wir ein Formular namens **frmEreignisprozeduren** an (siehe Bild 8). Dieses binden wir an die Tabelle **tblEreignisprozeduren** und ziehen die beiden Felder **Bezeichnung** und **Code** in den Detailbereich der Entwurfsansicht. Außerdem fügen wir ein Kombinationsfeld namens **cboSchnellauswahl** hinzu. Dieses verwendet ebenfalls die Tabelle **tblEreignisprozeduren** als Basis für die Datensatzherkunft. In diesem Fall stellen wir eine Abfrage zusammen, welche die beiden Felder **EreignisprozedurID** und **Bezeichnung** enthält, wobei die Einträge aufsteigend nach der Bezeichnung sortiert werden.

Da das Kombinationsfeld nur die Bezeichnungen der Ereignisprozeduren anzeigen soll und nicht die Primärschlüsselwerte, stellen wir noch die Eigenschaften **Spaltenanzahl** auf **2** und **Spaltenbreiten** auf **0cm** ein.

Damit das Formular beim Öffnen vom Assistenten aus mittig angezeigt wird, stellen wir die Eigenschaft **Automatisch zentrieren** auf den Wert **Ja** ein.

Die aktuell angezeigte Ereignisprozedur soll beim Schließen des Formulars in das Kombinationsfeld **cboEreigni-**

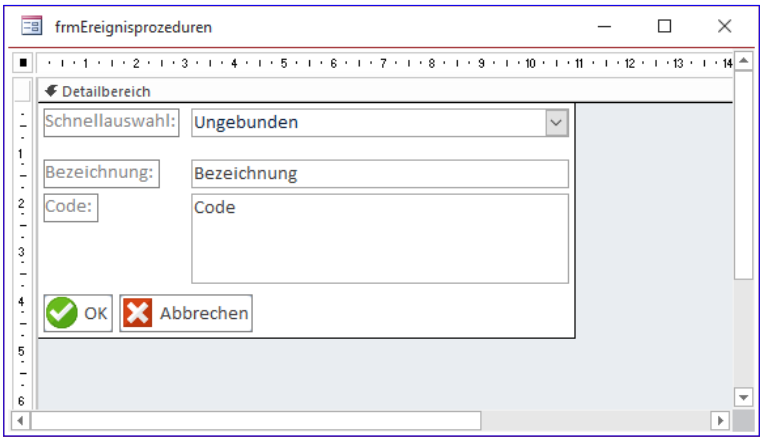


Bild 8: Entwurf des Formulars **frmEreignisprozeduren**