AUSGABE 05/2023

VISUAL BASIC ENTWICKLER

MAGAZIN FÜR DIE PROGRAMMIERUNG VON MICROSOFT OFFICE UND ANDEREN ANWENDUNGEN MIT VB.NET, VBA UND TWINBASIC

IN DIESEM HEFT:

André Minhorst Verlag

Windows-Apps mit twinBASIC

VB6 wird nicht mehr weiterentwickelt und es wird auch keine 64-Bit-Version mehr geben. Wechseln wir jetzt endgültig zu .NET oder anderen Alternativen? Oder schauen wir uns eine leichtgewichtige Alternative an, die nicht nur Kompatibilität zu VB6 verspricht, sondern auch noch 64-Bit-fähig ist und sogar Erweiterungen zum klassischen Visual Basic offeriert? Mit der wir nicht nur Desktop-Apps, sondern auch ausführbare Konsolenanwendungen sowie COM-DLLs und COM-Add-Ins zur Erweiterung unserer Office-Anwendungen programmieren können? Genau das liefert twinBASIC, das sich zwar noch in der Entwicklung befindet, aber dennoch schon vieles leistet.

Deshalb steigen wir vom Visual Basic Entwickler tiefer in die Entwicklungsumgebung und Programmiersprache twinBASIC ein, die außerdem sogar noch kostenlose Entwicklung von 32-Bit-Programmen enthält. Erst bei der Veröffentlichung von 64-Bit-Programmen müsste man in die Tasche greifen, um sich der sonst erscheinenden Splash-Screens zu entledigen. Dafür erhält man allerdings auch noch andere Features der Professional-Version.

Einige Möglichkeiten wie die zum Erstellen von COM-DLLs und COM-Add-Ins haben wir in vorhergehenden Ausgaben bereits betrachtet. In dieser Ausgabe steigen wir in die Entwicklung von Windows-Anwendungen ein, die eine eigene Benutzeroberfläche bieten und schon vieles von dem bieten, was auch unter Visual Studio für Visual Basic 6 möglich war – plus den in der Einleitung erwähnten Erweiterungen.

Im Artikel **twinBASIC: Grundlagen zur App-Entwicklung** stellen wir ab Seite 4 die Funktionen zum Erstellen von Windows-Anwendungen mit twinBASIC vor. Hier legen wir erste Anwendungen und Fenster alias Forms an, starten Projekte, legen Projekteigenschaften fest, lernen den Form-Entwurf mit den Möglichkeiten zum Anlegen und Anpassen von Steuerelementen kennen, stellen Form- und Steuerelementeigenschaften ein und sehen uns in den verschiedenen Menüs um.

Unter dem Titel **twinBASIC: Überblick der Controls und Eigenschaften** lernst Du ab Seite 17 die Steuerelemente und ihre Eigenschaften kennen, bevor wir in den nächsten Ausgaben genauer auf die einzelnen Controls eingehen.

Der Artikel **twinBASIC: Fenster öffnen, schließen und mehr** zeigt ab Seite 26, wie man Fenster beziehungsweise Form-Elemente öffnet und schließt – sowohl beim Start der Anwendung als auch über entsprechende Schaltflächen. Außerdem sehen wir uns hier an, wie man Fenster platziert und welche Modi es beim Öffnen gibt. Natürlich müssen wir auch wissen, wie wir auf die Ereignisse von Fenstern reagieren können. Das zeigen wir im Artikel **twinBASIC: Ereigniseigenschaften von Fenstern** ab Seite 34. Welche Ereignisse gibt es? Wir programmiert man sie? All dies in diesem Artikel.

Wer Formulare öffnet, möchte auch mal Daten an das geöffnete Formular übergeben – und nachher die geänderten Daten auslesen. Dieses Thema behandelt der Artikel **twinBASIC: Daten von Form zu Form** ab Seite 43. Und damit wir die Möglichkeit zum Öffnen von Formularen und zum Aufrufen von Funktionen über die Benutzeroberfläche haben, lernen wir im Artikel **twinBASIC: Menüs erstellen** ab Seite 52 alles über das Erstellen von Menüs über die Benutzeroberfläche.

Schließlich widmen wir uns unter **COM-Add-Ins registrieren** ab Seite 59 noch den Problemen bei der Registrierung per Setup und wie wir diese beheben können.

Nun viel Spaß beim Lesen!

Dein André Minhorst

twinBASIC: Grundlagen zur App-Entwicklung

VB6-Anwendungen – braucht das noch jemand? Oh ja! Gerade wer mal eben schnell eine Windows-Anwendung mit einer Benutzeroberfläche (oder auch ohne) programmieren will, kann immer noch gut auf seine guten, alten Visual Basic 6-Kenntnisse zurückgreifen – und zwar mit dem Nachfolger twinBASIC. In diesem Magazin haben wir schon die eine oder andere Lösung damit programmiert, die wir als COM-Add-In oder als COM-DLL für die Integration in eine der Office-Anwendungen oder auch zur Erweiterung von VBA genutzt haben. Und in diesem Artikel gehen wir noch einen Schritt weiter: Wir zeigen die grundlegenden Werkzeuge von twinBASIC zur Entwicklung von ausführbare twinBASIC-Anwendung mit eigener Benutzeroberfläche. Mit twinBASIC können wir unsere .exe-Anwendung in Zukunft vielleicht sogar auf Nicht-Windows-Rechnern laufen lassen. Zumindest aber ist die Kompatibilität mit 64-Bit-Systemen gesichert.

VB6 im modernen Zeitalter: Ein Relikt mit Wert

Visual Basic 6 (VB6) mag auf den ersten Blick wie ein Überbleibsel aus der Technologie-Vergangenheit erscheinen, doch die Realität ist, dass diese robuste Programmiersprache immer noch ihren festen Platz in der Software-Entwicklung hat. Erstens ist die Migration von alten VB6-Anwendungen auf neuere Technologien oft ein zeitaufwändiger und kostspieliger Prozess, den viele Unternehmen scheuen. Daher bleiben VB6-Programme in Betrieb und erfordern weiterhin Wartung und Support.

Zweitens bietet VB6 eine vertraute Umgebung für Entwickler, die über Jahrzehnte hinweg mit der Sprache gearbeitet haben, und ermöglicht eine schnelle und effiziente Anwendungsentwicklung. Auch wenn VB6 nicht die neuesten Funktionen oder Sprachkonstrukte bietet, bleibt seine Einfachheit und Effizienz in vielen Kontexten unübertroffen. Es ist ein klares Beispiel dafür, dass es nicht immer das Neueste und Beste sein muss, um relevant und wertvoll in der IT-Landschaft zu bleiben.

twinBASIC als Entwicklungsumgebung

Neu sind der Compiler und die Entwicklungsumgebung, die wir in diesem Fall nutzen. Leser unseres Magazins kennen twinBASIC bereits aus verschiedenen Veröffentlichungen, daher empfiehlt sich für den Einstieg ein

Bild 1: Erstellen einer Standard-Exe

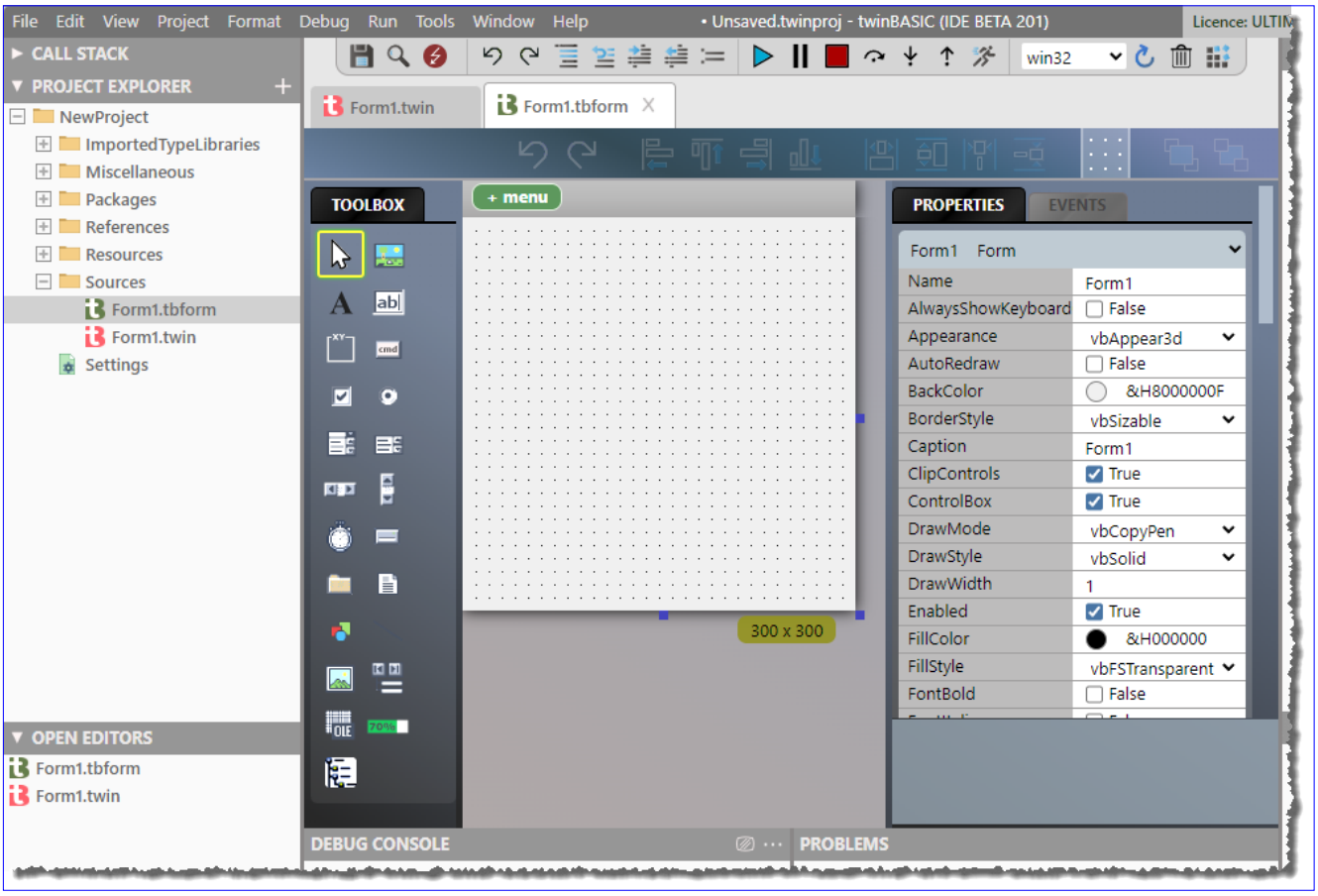

Bild 2: Ein neues Projekt des Typs **Standard Exe**

Blick in das Onlinearchiv von **https://www.vbentwickler.de** mit einer Suche nach dem Begriff **twinBASIC**. twinBASIC ist zumindest für die Entwicklung mit der 32-Bit-Variante kostenlos, bei Entwicklung von 64-Bit-Apps werden bei Verwendung der kostenlosen Variante Splash-Screens eingeblendet. Die 64-Bit-Version ohne Splash-Screens erfordert den Kauf der Professional-Lizenz (siehe **https://shop.minhorst.com/access-tools/364/twinbasic-professional-edition-jahreslizenz?c=78**).

Neues Projekt erstellen

Nach dem Start von twinBASIC wählen wir aus den Vorlagen den Eintrag **Standard Exe** aus (siehe Bild 1). Das Ergebnis ist ein neues Projekt, das lediglich ein leeres Fenster enthält. Dieses besteht wiederum aus zwei Elementen: der eigentlichen Benutzeroberfläche (**Forms1.tbform**) und dem Code behind-Klassenmodul (**Form1.twin**).

Bild 3: Das noch leere Hauptfenster der Anwendung

Beides sehen wir im **Project Explorer** von twinBASIC, in der Mitte wird direkt der Entwurf des neuen Fensters dargestellt (siehe Bild 2).

Erster Start des Projekts

Mit einem Klick auf die Ausführen-Schaltfläche in der Symbolleiste von twinBASIC

können wir das Projekt direkt ausprobieren. Damit zeigen wir das leere Fenster des Projekts an (siehe Bild 3).

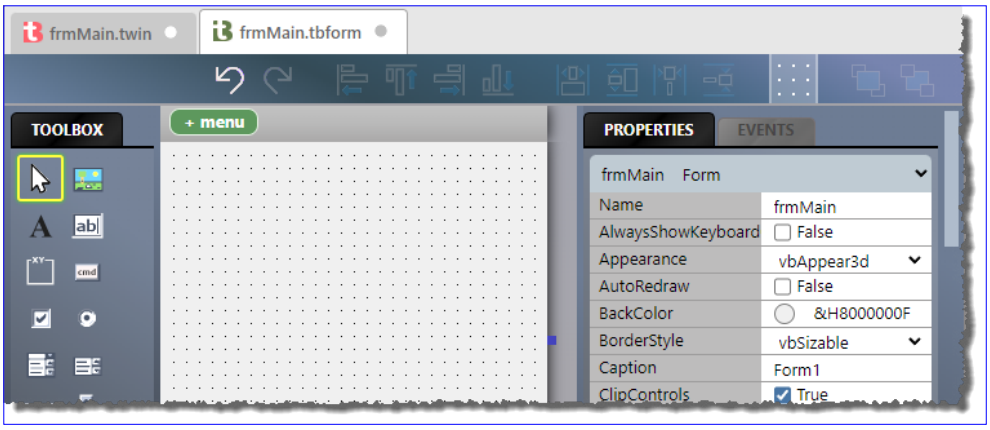

Bild 4: Umbenennen der **Form**-Klasse

Umbenennen der Objekte

Bevor wir das Projekt erstmals speichern, wollen wir die bisher vorhanden Objekte umbenennen. So soll das einzige bisher vorhandene **Form**-Objekt in **frmMain** umbenannt werden. Dazu schließen wir dieses Objekt und auch die dazugehörige Code behind-Klasse, sofern diese aktuell geöffnet sind. Zum Umbenennen verwenden wir den Kontextmenü-Eintrag **Rename** und legen die Bezeichnungen **frmMain.tbform** und **frmMain.twin** fest. Dies waren jedoch nur die Dateinamen. Außerdem ändern wir noch die entsprechenden Klassennamen. Die für das Fenster ändern wir im Eigenschaftsbereich. Hier stellen wir die Eigenschaft **Name** auf **frmMain** ein (siehe Bild 4).

Für die Code behind-Klasse nehmen wir die Änderungen direkt im Code vor (siehe Bild 5).

Woher weiß die Code behind-Klasse, dass sie zu dem entsprechenden **Form**-Element gehört? Dies wird durch eine GUID festgelegt, die wir in beiden Fällen in der Eigenschaft **FormDesignerId** finden.

Umbenennen, schnelle Version

Dieser Umbenennungsprozess ist nur bei der standardmäßig vorhandenen Form nötig. Wenn wir über den Kontextmenüeintrag **Add|Add Form Windows Form** des Eintrags **Sources** im Projektexplorer ein neues **Form**-Objekt hinzufügen, haben wir direkt nach dem Anlegen die Möglichkeit, dieses umzubenennen (siehe Bild 6).

Geben wir hier einen Namen wie im Beispiel **frmSample** an, wird auch gleich die Code behind-Klasse umbenannt. Außerdem wird beim **Form**-Objekt der Objektname auf **frmSample** eingestellt und auch der Klassenname in der Datei **frmSample.twin**.

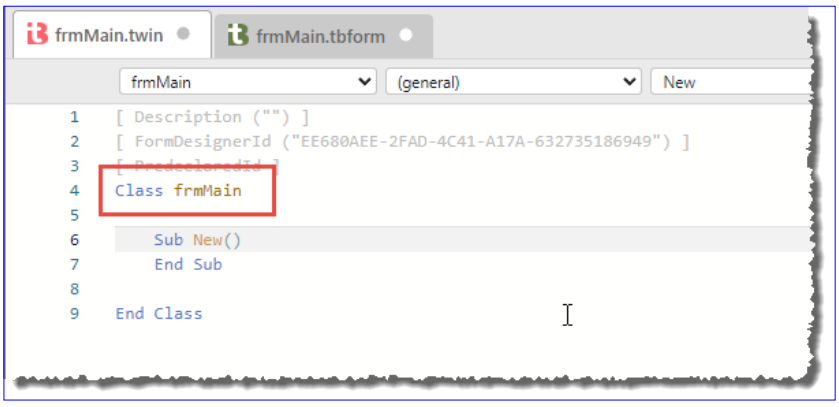

Bild 5: Umbenennen der Code behind-Klasse

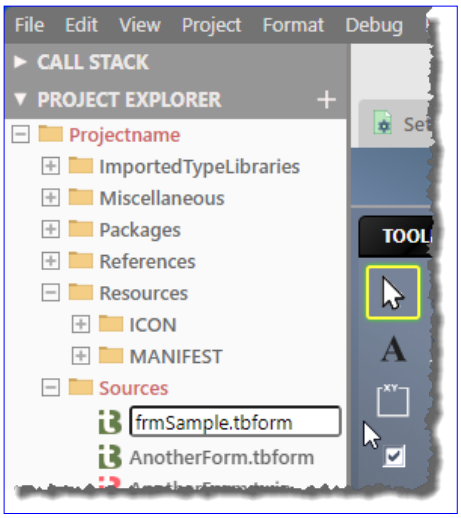

Bild 6: Umbenennen des Form-Elements

Wenn Du also beim Start das vorhandene **Form**-Element **Form1** umbenennen möchtest, löscht Du es am besten direkt und erstellst ein neues **Form**-Element mit dem gewünschten Namen.

Startup-Form festlegen

Damit haben wir nun Änderungen durchgeführt, die sich auf den Start des Projekts auswirken. Beim ersten Test wurde nämlich das **Form**-Objekt **Form1** geöffnet. Dieses ist nun nicht mehr vorhanden, also müssen wir angegeben, dass das nun in **frmMain** umbenannte Fenster geöffnet werden soll. Das erledigen wir in den **Settings**, die wir über den gleichnamigen Eintrag im Projektexplorer öffnen.

Hier finden wir etwas weiter unten den Bereich **Project: Startup Object**. Hier wählen wir wie in Bild 7 den Eintrag **frmMain** aus.

Alternativ können wir auch **Sub Main** auswählen. In diesem Fall müssten wir eine Prozedur namens **Main** bereitstellen, die beim Start aufgerufen wird. Dies eröffnet die Möglichkeit, vor dem Anzeigen des ersten Fensters bereits einige Anweisungen auszuführen. Wie das gelingt, zeigen wir im Artikel **twinBASIC: Forms öffnen, schließen und mehr** (**www.vbentwickler.de/389**).

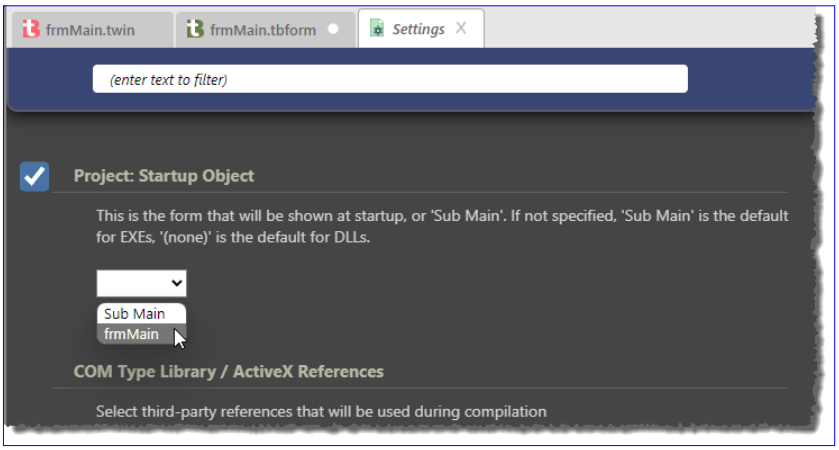

Bild 7: Einstellen des bei Start anzuzeigenden **Form**-Objekts

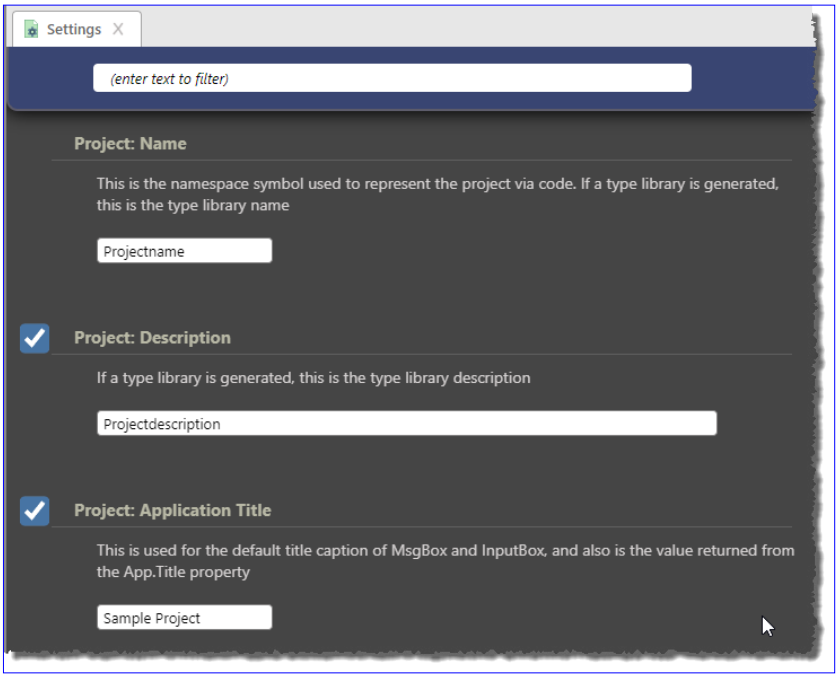

Bild 8: Einstellen weiterer Projekteigenschaften

Weitere Projekteinstellungen

Im gleichen Dialog finden wir weiter oben noch weitere wichtige Einstellungen, zum Beispiel den Projektnamen, eine Beschreibung und den Anwendungstitel

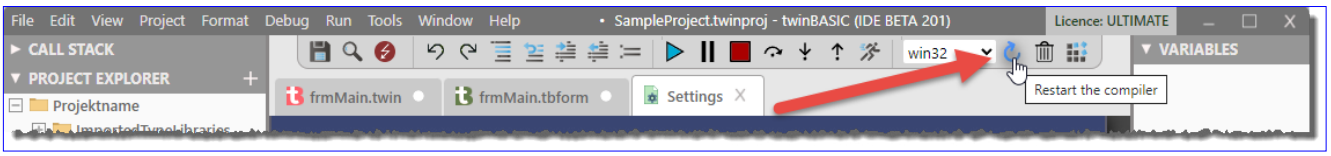

Bild 9: Schaltfläche zum Neustarten des Compilers

Bild 10: Die Schaltflächen zum Starten von Programmen und mehr

(siehe Bild 8). Der Wert unter **Project:Name** wird nicht angezeigt, sondern dient zum Beispiel folgenden Zwecken:

- Namensraum: Der Projektname dient als Teil des Namensraums für die in unserem Projekt erstellten Klassen und Formulare. Dies hilft dabei, Konflikte mit gleichnamigen Klassen oder Formularen in anderen Projekten zu vermeiden.

- Referenzen: Der Projektname wird in Referenzen und Verweisen auf Projekte und Bibliotheken innerhalb unseres VB6-Projekts verwendet. Wenn wir auf externe COM-Komponenten oder DLLs zugreifen, können wir den Projektname verwenden, um diese zu identifizieren und auf sie zuzugreifen.

Der Wert unter **Project: Description** wird angezeigt, wenn wir eine Typbibliothek, also eine DLL, auf Basis dieses Projekts erstellen und dieses referenzieren.

Da wir das nicht tun, können wir hier einen beliebigen Wert eintragen.

Der Wert unter **Project: Application Titel** hat folgende Funktionen:

- Standardbeschriftung für **MsgBox**- und **InputBox**-Objekte, für die wir keinen eigenen Titel festgelegt haben.

- Wert, der für die Eigenschaft **App.Title** zurückgegeben wird.

Die hier durchgeführten Änderungen wirken sich erst aus, wenn wir das Projekt gespeichert haben (was wir gleich im Anschluss erledigen) und den Compiler neu gestartet haben. Letzteres können wir durchführen, indem wir wie in Bild 9 auf die Schaltfläche **Restart the compiler** klicken.

Speichern des Projekts

Damit haben wir einen ersten Zwischenstand erreicht und speichern diesen mit **File|Save project** unter dem Namen **Sampleproject.twinproj**. Hier sehen wir direkt, dass genau eine einzige Datei für ein twinBASIC-Projekt angelegt wird.

Starten des Projekts

Damit können wir das Projekt nun erstmals starten. Die dazu notwendigen Schaltflächen finden wir in der Symbolleiste im mittleren Bereich (siehe Bild 10).

Für uns sind erst einmal die folgenden Schaltflächen wichtig:

- **Start/Continue (F5)**: Startet das Projekt.

- **Stop**: Stoppt das laufende Projekt. Das ist allerdings nicht der Fall, wenn noch ein **Form**-Objekt geöffnet ist – dieses muss von Hand geschlossen werden.

Klicken wir auf **Start/Continue (F5)** oder betätigen die Taste **F5**, wird das Projekt abhängig von der Einstellung für **Project: Startup Object** entweder mit dem dort angegebenen **Form**-Element gestartet oder mit einer Prozedur namens **Main**, die wiederum die Benutzeroberfläche öffnen sollte.

Wichtig: Dies erzeugt noch keine **.exe**-Datei, die wir dann beispielsweise weitergeben oder direkt aufrufen können.

Bild 11: Erstellen des Projekts

Projekt erstellen

Dies geschieht erst, wenn wir das Projekt tatsächlich erstellen. Dazu nutzen wir die Schaltflächen rechts in der Symbolleiste (siehe Bild 11).

In diesem Bereich ist vor allem die **Build**-Schaltfläche rechts wichtig. Diese startet den Build-Vorgang, dessen Fortschritt und Ergebnis wir im Bereich **DEBUG CONSOLE** einsehen können. Hier sehen wir, das die Datei **Projectname_win32.exe** erfolgreich erstellt wurde (siehe Bild 12).

win32.exe wurde hier an den Projektnamen angehängt, weil wir im Auswahlfeld links im markierten Bereich den Wert **win32** selektiert haben. Wir können 32-Bit- und 64-Bit-Versionen unserer Projekte erzeugen.

Die Schaltfläche **Clean (deregister & delete build)** löscht die erstellt **.exe**-Datei wieder und hebt eventuell angelegte Registry-Einträge wieder auf. Registry-Einträge werden allerdings eher bei COM-Add-Ins erstellt. Die beiden Texte **Launch EXE** und **Open Folder** im Bereich **DEBUG CONSOLE** können angeklickt werden.

Der erste startet die **.exe**-Datei und der zweite zeigt den Windows-Explorer mit dem **Build**-Verzeichnis an.

Hier finden wir die erstellte Datei, in diesem Fall **Projectname_win32.exe**, vor. Diese können wir nun schon so weitergeben und auf anderen Rechnern ausführen oder sie in ein Setup einbinden.

Mit dem Form-Entwurf arbeiten

Der Entwurf für ein **Form**-Element sieht ähnlich aus wie in Visual Studio 6 (siehe Bild 13). Wir sehen in der Mitte den Entwurf des Fensters selbst. Links erscheint die Toolbox mit allen verfügbaren Steuerelementen. Rechts sehen wir die Eigenschaften, aufgeteilt in die beiden Bereiche **PROPERTIES** und **EVENTS**.

Um ein Fenster im Entwurf anzuzeigen, klicken wir einfach auf seinen Namen im Projektexplorer. Ist ein Fenster bereits geöffnet, können wir es auch durch einen Klick auf den entsprechenden Registerreiter über dem Entwurfsbereich aktivieren. Recht unscheinbar ist die Befehlsleiste für den Entwurf, zumindest so lange nicht mindestens ein zu bearbeitendes Steuerelement markiert ist. Dann ist nur die Schaltfläche **Toggle the grid indicators ON/OFF** markiert (siehe Bild 14). Sie bietet verschiedene Befehle, zum Beispiel:

- **Show Grid**: Gibt an, ob das Grid angezeigt werden soll.

- **Align Controls To Grid**: Gibt an, ob Steuerelemente an den Punkten des Grid eingerastet werden sollen. Eine Platzierung ist dann mit der linken, oberen Ecke nur an einem der Punkte des Grids möglich. Auch beim Verändern der Größe wird der rechte,

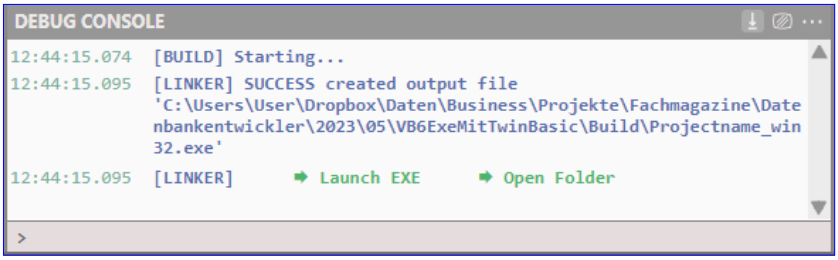

Bild 12: Ablauf der Erstellung eines Projekts

twinBASIC: Überblick der Controls und Eigenschaften

twinBASIC bietet genau wie Visual Basic 6 einige Steuerelemente an – neben den vielfach verwendete Elementen wie Button, TextBox und Label haben wir die Auswahl zwischen vielen weiteren Steuerelementen, mit denen sich die wesentlichen Anforderungen an Desktop-Anwendungen umsetzen lassen. In diesem Artikel schauen wir uns die Steuerelemente einmal im Überblick an und betrachten einige der Eigenschaften, welche die meisten Steuerelemente gemeinsam haben. In weiteren Artikeln gehen wir dann im Detail auf die einzelnen Steuerelemente ein und zeigen, wie wir diese programmieren und nutzen können.

Steuerelemente unter twinBASIC

twinBASIC bietet so weit die gleichen Steuerelemente, die wir auch in VB6 finden. Sobald wir eine Form in der Entwurfsansicht öffnen, erscheint links daneben der Bereich **TOOLBOX** mit den Steuerelementen (siehe Bild 1). Im oberen Bereich finden wir die eingebauten Steuerelemente von twinBASIC beziehungsweise VB6. Darunter werden weitere Steuerelemente angezeigt, die wir uns ebenfalls ansehen werden.

Hier sind die Standard-Steuerelemente:

- **PictureBox**: Ein Steuerelement, das dazu dient, Grafiken, Bilder oder sogar Zeichnungen anzuzeigen. Es kann auch als Container für andere Steuerelemente dienen.

- **Label**: Ein Steuerelement, das dem Benutzer Text anzeigt. Es wird häufig verwendet, um anderen Steuerelementen auf einem Formular einen Kontext oder eine Beschriftung zu geben. Das **Label**-Steuerelement kann keinen Fokus erhalten und der Benutzer kann seinen Inhalt nicht direkt ändern.

- **TextBox**: Ein Steuerelement, das dem Benutzer ermöglicht, Text einzugeben oder anzuzeigen. Es kann auch dazu verwendet werden, Passwörter oder andere verdeckte Eingaben zu akzeptieren.

- **Frame**: Ein Container-Steuerelement, das dazu dient, andere Steuerelemente zu gruppieren und zu organisieren, oft in Kombination mit **OptionButton**-Elementen, um eine Gruppe zu bilden.

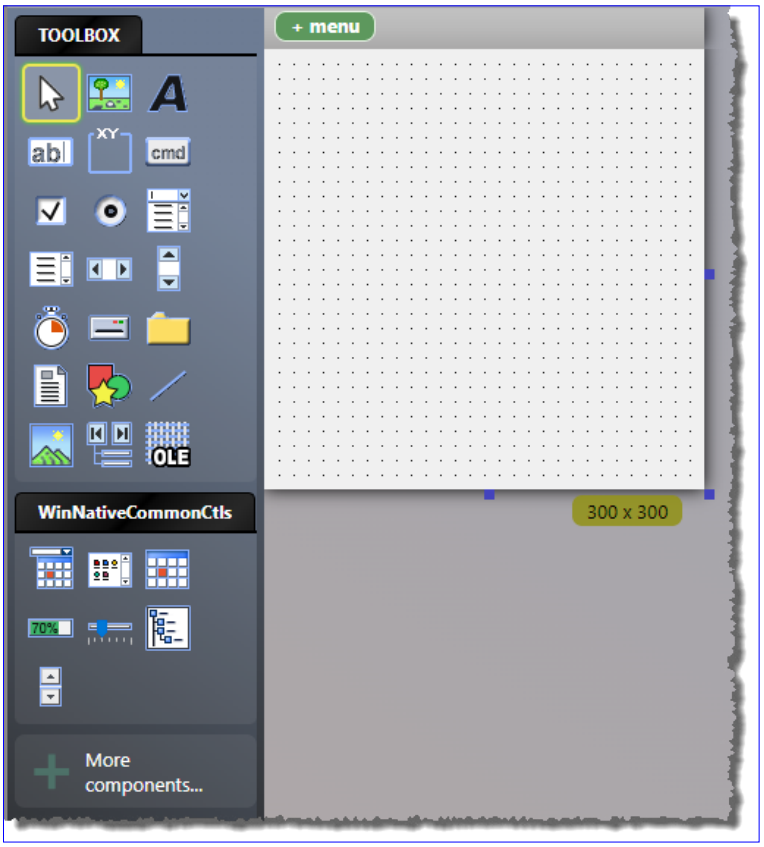

Bild 1: Die Toolbox mit den Steuerelementen

- **CommandButton**: Ein Button, den der Benutzer anklicken kann, um eine Aktion auszulösen. In anderen Umgebungen wird es oft einfach als **Button** bezeichnet.

- **CheckBox**: Ein Kästchen, das der Benutzer aktivieren (ankreuzen) oder deaktivieren (nicht ankreuzen) kann. Es wird oft verwendet, um Optionen anzugeben, die aktiviert oder deaktiviert sind.

- **OptionButton**: Auch als »RadioButton« bekannt. Erlaubt dem Benutzer, eine Option aus einer Gruppe von Optionen auszuwählen. Innerhalb einer Gruppe kann immer nur ein OptionButton aktiviert sein.

- **ComboBox**: Ein Dropdown-Steuerelement, das eine Liste von Einträgen anzeigt, aus denen der Benutzer auswählen kann. Es kann auch so konfiguriert werden, dass es benutzerdefinierte Texteingaben akzeptiert.

- **ListBox**: Ein Steuerelement, das eine Liste von Einträgen anzeigt. Der Benutzer kann einen oder mehrere Einträge aus dieser Liste auswählen.

- **HScrollBar** und **VScrollBar**: Horizontale oder vertikale Schieberegler, die dem Benutzer ermöglichen, durch einen Bereich von Werten zu scrollen oder eine große Ansicht in einem kleineren Bereich anzuzeigen.

- **Timer**: Ein nicht sichtbares Steuerelement, das Ereignisse in regelmäßigen Abständen auslöst. Es wird oft verwendet, um Aktionen periodisch auszuführen, wie zum Beispiel das Aktualisieren von Daten oder das Animieren von Benutzeroberflächen.

- **DriveListBox**: Ermöglicht die Auswahl eines Laufwerks auf dem Computer. Es zeigt eine Liste der verfügbaren Laufwerke (Festplatten, CD-ROM-Laufwerke und so weiter) an und ermöglicht es, ein Laufwerk auszuwählen.

- **DirListBox**: Ermöglicht die Auswahl eines Verzeichnisses oder Ordners auf dem Computer. Es zeigt die Verzeichnisstruktur an und ermöglicht es, ein Verzeichnis auszuwählen.

- **FileListBox**: Zeigt eine Liste von Dateien in einem ausgewählten Verzeichnis oder Ordner an und ermöglicht es, Dateien auszuwählen und auf sie zuzugreifen.

- **Shape**: Grundlegendes Zeichenwerkzeug, mit dem wir einfache Formen wie Rechtecke, Kreise und Linien auf einem Formular erstellen können. Es kann für die Darstellung von Grafiken oder zur Hervorhebung von Bereichen in einem Formular verwendet werden.

- **Line**: Eine einfache Zeichenlinie oder ein Strich, der auf einem Formular angezeigt wird. Es kann verwendet werden, um Linien oder Trennlinien in der Benutzeroberfläche zu erstellen.

- **Image**: Dient zur Anzeige von Bildern oder Grafiken auf einem Formular. Es kann Bilder aus Dateien oder Ressourcen laden und in verschiedenen Formaten angezeigt werden.

- **OLE**: Ermöglicht die Integration von OLE-fähigen Objekten in eine VB6-Anwendung. Es kann verwendet werden, um Dokumente, Tabellenkalkulationen, Diagramme und andere OLE-Objekte in einem Formular anzuzeigen oder zu bearbeiten.

Ein paar dieser Steuerelemente schauen wir uns in diesem einführenden Artikel so weit zu Beispielzwecken nötig an. Wir gehen aber in einzelnen Artikeln im Detail auf die verschiedenen Steuerelemente ein.

Weitere Steuerelemente

Im Bereich **WinNativeCommonCtls** finden wir weitere Steuerelemente:

- **DTPicker** (Datumsauswahlfeld): Ermöglicht die Auswahl von Datum und Uhrzeit in einem benutzerfreundlichen Kalender. Benutzer können ein Datum auswählen, indem sie auf den Kalender klicken, und Datum und Uhrzeitwerte eingeben.

- **ListView** (Listenansicht): Bietet eine tabellarische Ansicht von Daten, die in Spalten und Zeilen organisiert sind. Es unterstützt verschiedene Ansichtsmodi, einschließlich Symbolansicht, Miniaturansicht und Detailansicht. Entwickler können benutzerdefinierte Daten in einem ListView anzeigen und Bearbeitungsfunktionen bereitstellen.

- **MonthView** (Monatsansicht): Stellt einen Kalender in Monatsansicht dar. Benutzer können auf verschiedene Monate klicken, um Datumsinformationen anzuzeigen. Es ist hilfreich, wenn Sie Datumsauswahl in einem größeren Kontext anzeigen möchten.

- **ProgressBar** (Fortschrittsbalken): Visualisiert den Fortschritt einer Aufgabe oder eines Prozesses, indem es einen Balken anzeigt, der sich von links nach rechts füllt. Es wird oft verwendet, um Benutzern den Fortschritt einer längeren Aufgabe anzuzeigen.

- **Slider** (Schieberegler): Ermöglicht die Auswahl eines Wertebereichs, durch das Ziehen eines Schiebereglers. Es wird verwendet, um Werte innerhalb eines bestimmten Bereichs auszuwählen oder einzustellen, beispielsweise Lautstärke oder Helligkeit.

- **TreeView** (Baumansicht): Ermöglicht die Darstellung von Daten in einer hierarchischen Baumstruktur. Benutzer können Knoten erweitern oder zusammenklappen, um Unterelemente anzuzeigen oder zu verbergen. Es wird oft für die Darstellung von Kategorien oder Verzeichnisstrukturen verwendet.

- **UpDown** (Hoch-Runter-Steuerung): Besteht aus Aufwärts- und Abwärtspfeilen und wird oft mit Textfeldern oder anderen Steuerelementen kombiniert. Benutzer können es verwenden, um Werte zu erhöhen oder zu verringern, beispielsweise zur Einstellung von Zahlenwerten.

Steuerelemente hinzufügen

Wie wir Steuerelemente zum Entwurf einer Form hinzufügen, haben wir uns bereits im Artikel **twinBASIC: Grundlagen zur App-Entwicklung** (**www.vbentwickler.de/388**) angesehen. Hier haben wir auch gesehen, wie man Steuerelemente anordnet und ihre Größe anpasst. Deshalb schauen wir uns hier nun direkt die Eigenschaften an, welche die meisten Steuerelemente gemeinsam haben.

Allgemeine Steuerelementeigenschaften

Steuerelemente haben viele verschiedene Eigenschaften und einige davon sind speziell für bestimmte Steuerelemente gedacht. Diese wollen wir uns in den jeweiligen Artikeln ansehen, in denen wir die einzelnen Steuerelemente beschreiben. An dieser Stelle betrachten wir die Eigenschaften, die alle oder zumindest viele Steuerelemente aufweisen – wie zum Beispiel **Name**.

Die Eigenschaften haben eines gemein: Sie können sowohl über den Eigenschaftsbereich in der Benutzeroberfläche eingestellt werden als auch per VBA. Und in den folgenden Abschnitten kommen auch schon die Eigenschaften, die fast alle Standard-Steuerelemente aufweisen.

Steuerelement benennen mit der Name-Eigenschaft

Die Eigenschaft **Name** definiert den Namen eines Steuerelements. Dieser Name dient dazu, das Steuer-

element im Code zu identifizieren und auf dieses zuzugreifen.

Hier verwenden wir in der Regel ein aus drei Buchstaben bestehendes Präfix wie zum Beispiel **txt** für **TextBox**, **cmd** für **CommandButton**, **lbl** für **Label** und so weiter und eine Bezeichnung, die den Inhalt des Steuerelements charakterisiert. Bei einem Textfeld zum Abfragen eines Dateipfades also beispielsweise **txtFilepath**.

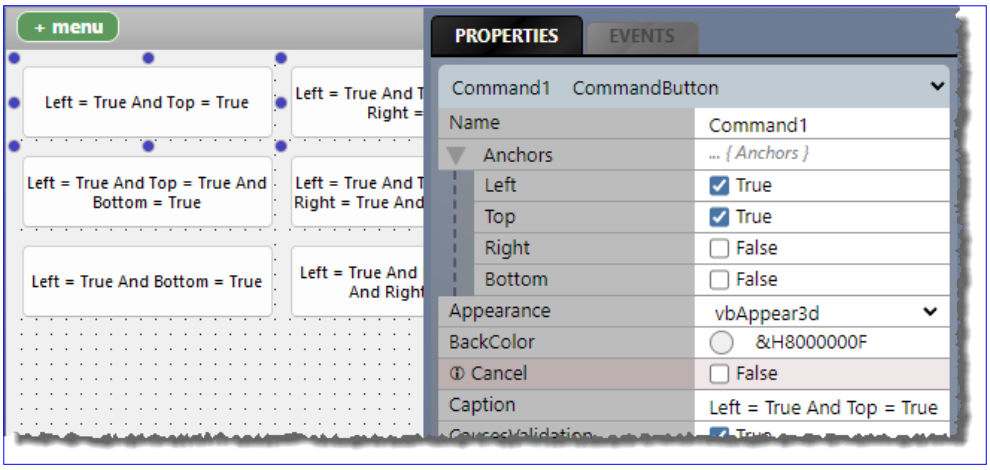

Bild 2: Einstellungen für die Eigenschaft **Anchors**

Verankern mit Anchors

Die **Anchors**-Eigenschaft ist eine echte Neuerung in twinBASIC gegenüber VB6. Sie bestimmt, wie ein Steuerelement in einem Formular verankert ist.

Wer bereits mit Access gearbeitet hat, kennt vielleicht die Eigenschaften **Horizontaler Anker** und **Vertikaler Anker** – diese arbeiten ähnlich.

Anchors ist eigentlich eine Klasse, die vier weitere Eigenschaften aufweist, die jeweils den Wert **True** oder **False** erwarten:

- **Left**: Gibt an, ob das Steuerelement am linken Rand verankert werden soll.
- **Top**: Gibt an, ob das Steuerelement am oberen Anker verankert werden soll.
- **Right**: Gibt an, ob das Steuerelement am rechten Rand verankert werden soll.
- **Bottom**: Gibt an, ob das Steuerelement am unteren Rand verankert werden soll.

Die Eigenschaften sehen wir, wenn wir im Eigenschaftsbereich den Eintrag **Anchors** aufklappen (siehe Bild 2).

Bild 3: Beispiele für verschiedene Einstellungen der Eigenschaft **Anchors**

Wenn wir alle vier Eigenschaften auf **True** einstellen, dann wird das Steuerelement beim Vergrößern des umgebenden **Form**-Elements entsprechend vergrößert.

Wenn wir nur die Eigenschaften **Left** und **Top** auf **True** einstellen, behält das Steuerelement Größe und

twinBASIC: Fenster öffnen, schließen und mehr

Form-Objekte waren in VB6 und sind in twinBASIC das Element der Wahl, wenn es um die Abbildung von Benutzeroberflächen geht. In diesem Artikel zeigen wir die grundlegenden Techniken rund um die Verwendung und Programmierung von Forms. Dabei zeigen wir, wie Du ein Form-Element öffnest, es wieder schließt und welche Öffnungsmodi es gibt. Dabei beschreiben wir auch, wie Du die Position eines Formulars nach dem Öffnen festlegen kannst. Außerdem schauen wir uns schon einige der grundlegenden Ereigniseigenschaften von Forms an und zeigen, wie wir diese in Form von Ereignisprozeduren implementieren können.

Fenster oder Form?

In den folgenden Abschnitten werden wir synonym von Fenstern und Forms sprechen. Allgemein ist zwar nicht jedes Fenster ein **Form**-Objekt, aber im Kontext des vorliegenden Artikels schon.

Forms öffnen

Wenn wir mit einer neuen twinBASIC-Anwendung starten, haben wir zwei Möglichkeiten, das erste Fenster zu öffnen:

- Durch Angabe des Fensters für die Eigenschaft **Project: Startup Object** oder
- durch Aufrufen der Prozedur **Main**, der wir den Code zum Öffnen eines Fensters zuweisen.

Form als Startfenster öffnen

Wir schauen uns zuerst die erste Möglichkeit an, wo wir ein Fenster als Startfenster festlegen. Damit wir in unserem neuen Projekt namens **Forms_OpenCloseAndCo** nicht die ganze Zeit mit einem Startform namens **Form1** arbeiten müssen, löschen wir dieses und fügen über den Kontextmenü-Eintrag **Add|Add Form** ein neues **Form**-Element hinzu, das wir gleich nach dem Anlegen **frmMain** nennen.

Achtung: Nur wenn wir dies erledigen, solange der Bearbeitungsmodus des neuen **Form**-Elements im Projektexplorer aktiviert ist, werden alle Elemente des Form-Objekts umbenannt – also auch die Eigenschaft **Name** des **Form**-Elements, der Name der Code behind-Klassendatei und auch der Name der Klasse selbst. Hast Du diesen Zeitpunkt verpasst, kein Problem – lösche das Element einfach wieder und lege es erneut an.

Haben wir dies erledigt, brauchen wir nur noch die Optionen des Projekts aufzurufen und hier den Eintrag **frmMain** auszuwählen (siehe Bild 1).

Starten wir das Projekt nun, wird die Form **frmMain** angezeigt.

Startform über die Prozedur Main öffnen

Die zweite Möglichkeit ist das Öffnen des Fensters per Code. Wenn wir das Startformular **frmMain** per Code öffnen wollen, statt über die Eigenschaft, entfernen wir

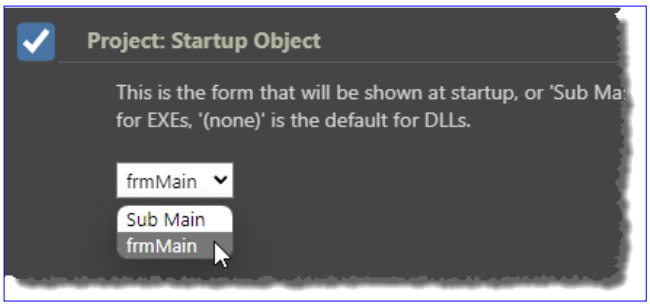

Bild 1: Einstellen der Startform **frmMain**

es zunächst aus der Eigenschaft **Project: Startup Object** und wählen dort den Wert **Sub Main** aus (siehe Bild 3).

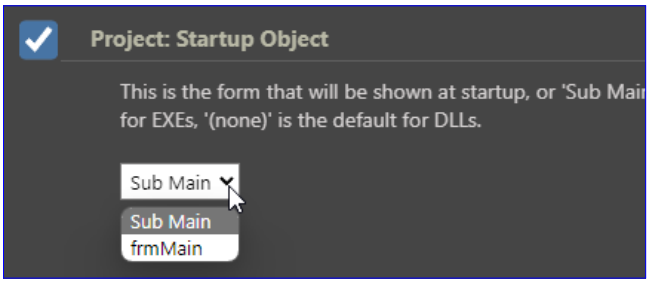

Bild 3: Einstellen der Startprozedur **Main**

Diese Prozedur müssen wir nun auch noch anlegen. Dazu fügen wir dem Projekt mit dem Kontextmenübefehl **Add|Add Module .TWIN supporting Unicode** ein neues Modul hinzu (siehe Bild 2). Das neue Modul benennen wir in **mdlMain** um.

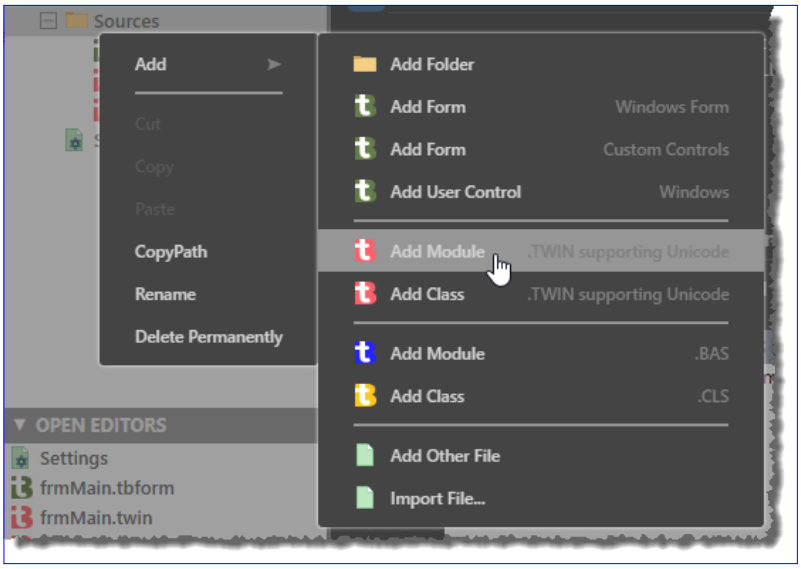

Bild 2: Hinzufügen eines Moduls zum Projekt

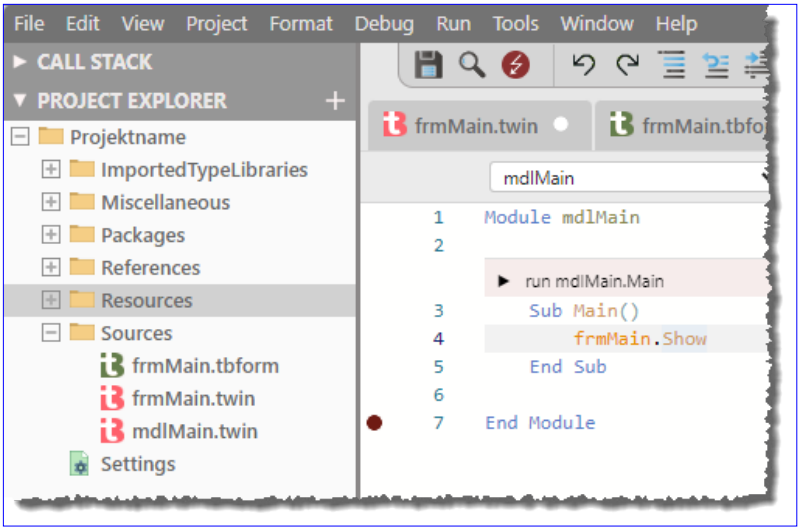

Bild 4: Anlegen der Startprozedur **Main**

Dieses öffnen wir nun und fügen die folgende Prozedur hinzu:

```
Sub Main()
    frmMain.Show
End Sub
```

Die einzige Anweisung der Prozedur besteht aus dem Objekt **frmMain** und der Methode **Show**. Jede Code behind-Klasse eines **Form**-Elements stellt die Eigenschaften und Methoden der **Form**-Klasse zur Verfügung, so zum Beispiel **Show**.

Um die Eigenschaften und Methoden dieser Klasse zu nutzen, müssen wir den Klassennamen angeben, der im Klassenmodul hinter dem Schlüsselwort **Class** angegeben ist. Sollte bei Dir die Klasse also einmal anders benannt sein als die Klassendatei (diese beiden sind nicht zu verwechseln), verwende den Klassennamen.

Das Codefenster sieht nun wie in Bild 4 aus. Starten wir die Anwendung nun erneut, erscheint das Fenster **frmMain** genauso, als wenn wir es als Startobjekt festgelegt hätten.

Damit wir sehen, dass tatsächlich die Prozedur das Fenster geöffnet hat, fügen wir dieser noch eine Meldung hinzu:

```
MsgBox "Vor dem Öffnen"
frmMain.Show
MsgBox "Nach dem Öffnen"
```

Die erste Meldung wird vor dem Anzeigen des Fensters eingeblendet, die zweite unmittelbar danach.

Forms schließen

Bisher sehen wir nur die X-Schaltfläche oben rechts als Möglichkeit, das Fenster zu schließen. Außerdem können wir durch einen linken Mausklick auf das Anwendungs-Icon das Systemmenü aus Bild 5 anzeigen. Dieses erscheint auch, wenn wir mit der rechten Maustaste auf eine beliebige Stelle der Titelleiste klicken.

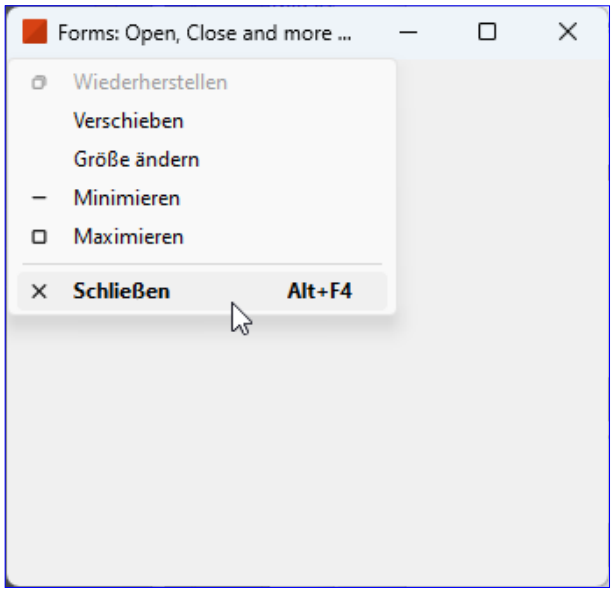

Bild 5: Systemmenü

Gegebenenfalls möchten wir aber eine eigene Schaltfläche zum Schließen des Fensters nutzen. Das erledigen wir, indem wir zunächst eine Schaltfläche zum Fenster hinzufügen.

Schaltfläche anlegen

Dazu aktivieren wir die Anzeige des Entwurfs des Fensters und klicken in der Toolbox einfach auf das Icon für die Schaltfläche. Dann ziehen wir im Entwurf einen Rahmen von der gewünschten Größe der anzulegenden Schaltfläche auf. Anschließend stellen wir im Eigenschaftsfenster den Namen auf **cmdUnload** und die Beschriftung auf **Unload** ein (siehe Bild 6).

Danach hinterlegen wir eine Ereignisprozedur für das Ereignis **Click** der Schaltfläche. Das gelingt ganz einfach, indem wir doppelt auf den Entwurf der Schaltfläche klicken. twinBASIC öffnet nun den Code-Editor mit der automatisch angelegten Prozedur. Dieser fügen wir gleich die Anweisung zum Schließen des aktuellen Fensters hinzu. Dabei gibt es allerdings verschiedene Varianten.

Fenster schließen beziehungsweise entladen

Um ein Fenster per Code zu schließen, gibt es mindestens die folgenden Möglichkeiten:

- **Unload Me**: Schließt das Formular und entfernt das Formular aus dem Speicher.
- **Me.Close**: Schließt das Formular, dieses bleibt jedoch im Speicher erhalten.

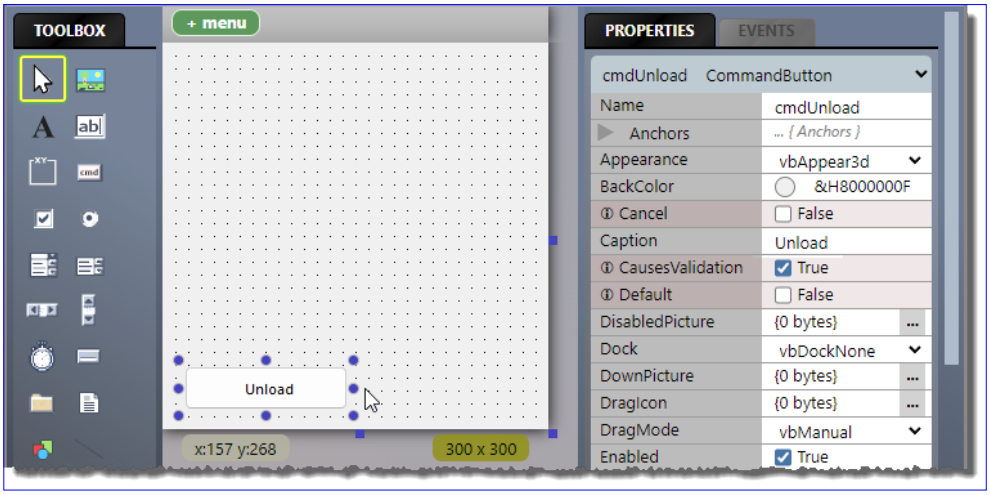

Bild 6: Anlegen einer Schaltfläche

- **SendMessage**: Ein API-Befehl, der das mit dem Handle referenzierte Fenster und alle enthaltenen Elemente zuverlässig terminiert.

Im Formular **frmMain** haben wir noch zwei weitere Schaltflächen untergebracht, um alle drei Methoden ausprobieren zu können. Die übrigen Schaltflächen heißen **cmdClose** und **cmdSendMessage** (siehe Bild 7).

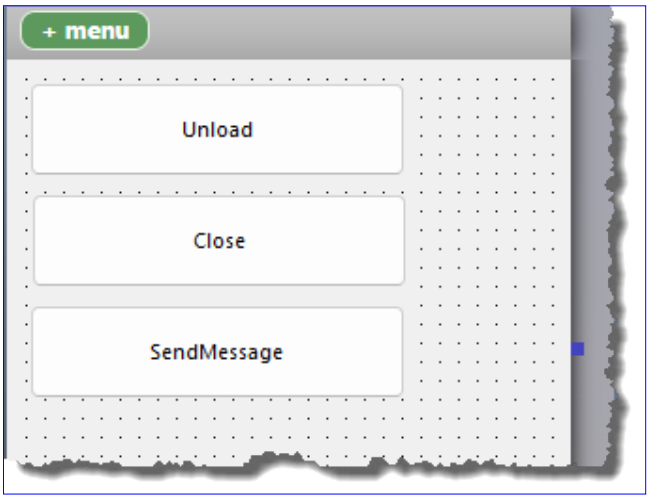

Bild 7: Schaltflächen mit verschiedenen Methoden zum Schließen des aktuellen Fensters

Die **Unload**-Prozedur sieht wie folgt aus:

```
Private Sub cmdUnload_Click()
    Unload Me
End Sub
```

Die **Close**-Prozedur sieht so aus:

```
Private Sub cmdClose_Click()
    Me.Close
End Sub
```

Und die **SendMessage**-Version enthält diesen Code:

```
Private Sub cmdSendMessage_Click()
    CloseForm Me.hWnd
End Sub
```

Die hier aufgerufene Prozedur **CloseForm** gibt es noch nicht, wir müssen sie noch definieren. Dazu hinterlegen wir ein neues Modul namens **mdlAPI.twin**, das den Code aus Listing 1 enthält. Das Modul enthält im ersten Teil die Deklaration einer API-Funktion namens **SendMessage**.

Diese rufen wir in der Prozedur **CloseForm** auf. Dabei nutzen wir den Parameter **hWnd**, welcher einen Pointer auf die Speicheradresse des zu schließenden **Form**-Objekts entgegennimmt. Dies ist der erste Parameter der **SendMessage**-Funktion. Der zweite gibt an, welche Nachricht an das Objekt gesendet werden soll – in diesem Fall **WM_CLOSE** zum Schließen des Objekts.

Diese Funktion können wir allen Projekten hinzufügen – es dient nicht nur zum Schließen des jeweils

```
Module mdlAPI
    Private Declare PtrSafe Function SendMessage Lib "user32" Alias "SendMessageA" (ByVal hwnd As LongPtr, _
        ByVal wMsg As Long, ByVal wParam As LongPtr, ByVal lParam As LongPtr) As LongPtr
    Private Const WM_CLOSE As Long = &H10

    Public Sub CloseForm(hWnd As LongPtr)
        SendMessage hWnd, WM_CLOSE, 0, 0
    End Sub
End Module
```

Listing 1: Die Prozedur **CloseForm** schließt das mit **hwnd** referenzierte **Form**-Objekt.

twinBASIC: Ereigniseigenschaften von Fenstern

Im Artikel twinBASIC: Forms öffnen, schließen und mehr (www.vbentwickler.de/389) haben wir bereits gezeigt, wie wir Fenster beziehungsweise Form-Elemente mit twinBASIC-Befehlen öffnen und schließen können und welche Varianten es dabei gibt. Über ein wichtiges Feature von Form-Elementen haben wir dabei noch nicht gesprochen: Die sogenannten Ereigniseigenschaften. Für diese können wir Prozeduren hinterlegen, die beim Auslösen bestimmter Ereignisse des Fensters ausgelöst werden – beispielsweise beim Öffnen, Schließen, Aktivieren, Deaktivieren oder auch beim Ändern der Größe oder der Position eines Fensters. Und auch wenn der Benutzer das Fenster an einer Stelle anklickt, die keine Steuerelemente enthält, wird ein Ereignis ausgelöst, auf das wir reagieren können. Wie wir solche Ereigniseigenschaften nutzen können und wir Du diese anlegst, zeigen wir in diesem Artikel.

Ereignisprozeduren anlegen

Im Artikel **twinBASIC: Forms öffnen, schließen und mehr** (**www.vbentwickler.de/389**) haben wir gelernt, wie wir **Form**-Elemente öffnen und schließen können, aber eine weitere, mächtige Möglichkeit haben wir uns noch nicht angesehen. Dabei handelt es sich um die Ereigniseigenschaften der **Form**-Klasse.

Was aber sind Ereignisse, Ereigniseigenschaften und Ereignisprozeduren überhaupt?

Ereignisse existieren erst einmal unabhängig von Dingen wie Ereigniseigenschaften oder Ereignisprozeduren. Ereignisse geschehen bei der Benutzung – ein **Form**-Element wird geöffnet, eine Schaltfläche wird angeklickt, ein Form-Element verliert den Fokus, weil ein anderes **Form**-Element diesen erhält, der Benutzer klickt mit rechten Maustaste auf einen Bereich, um ein Kontextmenü anzuzeigen und so weiter.

Hier setzen wir mit Ereigniseigenschaften und Ereignisprozeduren an, um uns diese Ereignisse zu Nutze zu machen. Dabei legen wir im Prinzip fest, dass beim Auftreten eines Ereignisses eine bestimmte Prozedur ausgelöst werden soll. Das geschieht durch das Erstellen einer Ereignisprozedur und durch das Einstellen einer Ereigniseigenschaft auf den Namen dieser Prozedur (zumindest in twinBASIC – in anderen Entwicklungsumgebungen gibt es andere Vorgehensweisen).

Dann geschieht Folgendes:

- Der Benutzer führt eine Aktion durch, für die es eine Ereigniseigenschaft gibt, beispielsweise das Anklicken des **Form**-Elements.
- Die Anwendung schaut, ob für die entsprechende Ereigniseigenschaft eine Ereignisprozedur hinterlegt ist.
- Falls ja, wird diese Ereignisprozedur aufgerufen.

Da sich dies sehr theoretisch anhört, schauen wir uns nun in der Praxis an, wie es funktioniert.

Beispiel für eine Ereignisprozedur

Wir schauen uns eines der ersten Ereignisse an, das beim Öffnen eines Formulars ausgelöst wird, und das durch die **Load**-Ereigniseigenschaft repräsentiert wird.

Um eine Ereignisprozedur anzulegen, die durch dieses Ereignis ausgelöst wird, wechseln wir in der Entwurfs-

ansicht des **Form**-Elements, hier das Fenster **frmMain** des Beispielprojekts, zur zweiten Registerseite des Eigenschaftbereichs. Hier finden wir weiter unten den Eintrag **Load** (siehe Bild 1).

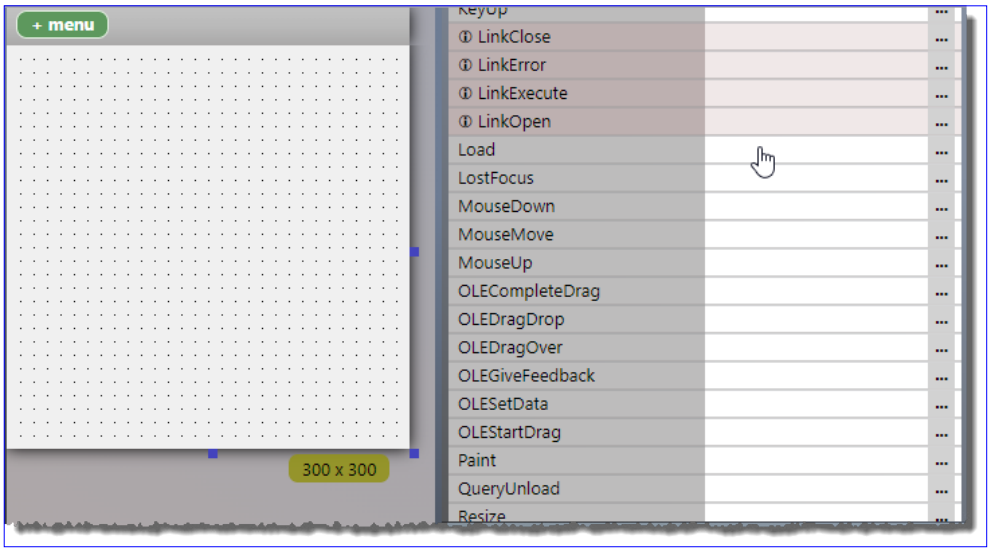

Bild 1: Ereignisprozedur anlegen

Wir brauchen nur in das Textfeld der Eigenschaft zu klicken, damit twinBASIC automatisch den Code-Editor für das Klassenmodul des **Form**-Elements öffnet und dort eine leere Ereignisprozedur für dieses Ereignis hinterlegt. Dieses sieht anschließend wie in Bild 2 aus.

Benennung von Ereignisprozeduren

Der Name der Ereignisprozedur setzt sich standardmäßig aus dem Namen des Elements, für welches das Ereignis ausgelöst wird, einem Unterstrich und dem Namen des Ereignisses zusammen. Im Falle des **Form**-Elements **frmMain** ist das jedoch anders – hier verwendet twinBASIC standardmäßig nicht beispielsweise **frmMain_Load**, sondern **Form_Load**. Das ist eine Vereinfachung, weil wir so in allen Klassenformularen von **Form**-Elementen die gleiche Benennung für solche Ereignisprozeduren haben. Auf diese Weise können wir die Ereignisprozeduren auch leicht kopieren und in anderen **Form**-Klassenmodulen verwenden. Bei anderen Elementen wie zum Beispiel Schaltflächen ist dies anders. Wollen wir eine Ereignisprozedur anlegen, die beim Anklicken beispielsweise einer Schaltfläche namens **cmdOK** ausgelöst wird, erhält diese tatsächlich den Namen **cmdOK_Click**. Warum verwendet twinBASIC hier nicht einfach **CommandButton_Click**? Ganz einfach: Beim **Form**-Element ist es meist so, dass wir nur das **Form**-Element im Code referenzieren, zu dem das Klassenmodul auch gehört. **Form**-Elemente enthalten aber meist mehr als ein Element des Typs

Bild 2: Die neue Ereignisprozedur

CommandButton, **TextBox** et cetera. Daher vergibt twinBASIC direkt eine Bezeichnung für die Ereignisprozedur, die den Namen des Steuerelements enthält.

Nach dem Anlagen der Ereignisprozedur trägt twinBASIC ihren Namen für die jeweilige Ereigniseigenschaft ein (siehe Bild 3).

Wenn wir die Ereignisprozedur vom **Form**-Entwurf aus bearbeiten wollen, können wir direkt auf die bereits gefüllte Eigenschaft klicken.

twinBASIC zeigt die entsprechende Ereignisprozedur dann im Codefenster an.

Wichtige Ereigniseigenschaften des Form-Elements

Schließlich wollen wir uns noch die grundlegenden Ereigniseigenschaften des **Form**-Elements ansehen und betrachten, wozu wir diese nutzen können. Hier ist die Übersicht der vorgestellten Ereignisse:

- **Form_Load**
- **Form_Unload**
- **Form_Resize**
- **Form_Click**
- **Form_KeyPress**
- **Form_Activate**
- **Form_Deactivate**
- **Form_GotFocus**
- **Form_LostFocus**
- **Form_QueryUnload**

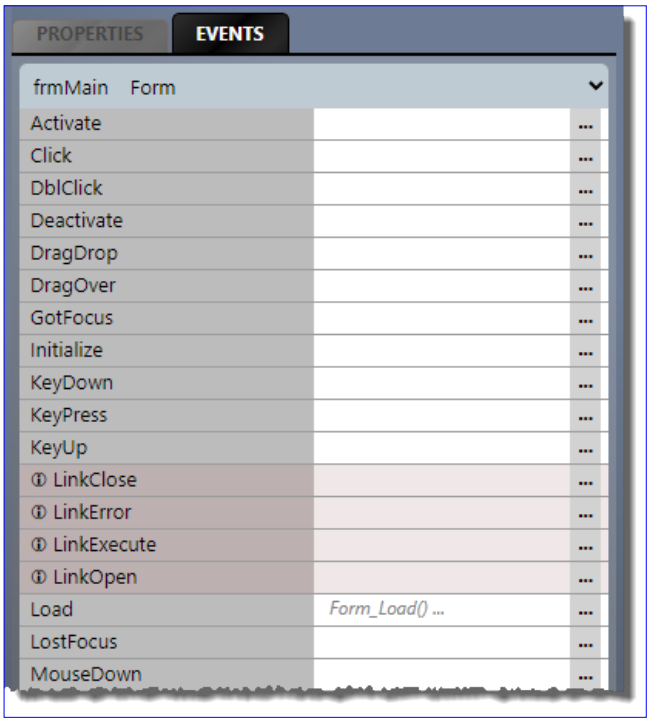

Bild 3: Ereignisprozedur im Eigenschaftsbereich

Reihenfolge der Ereignisse beim Öffnen eines Form-Elements

Wenn man die Ereignisse eines **Form**-Elements programmieren möchte, ist es wichtig, die Reihenfolge zu kennen, in der die Ereignisse ausgelöst werden.

Wie aber können wir die Reihenfolge herausfinden? Das ist zum Beispiel auf die folgenden beiden Arten möglich. Beide erfordern, dass wir für alle Ereigniseigenschaften jeweils eine Ereignisprozedur anlegen. Dann gibt es folgende Möglichkeiten:

- Wir fügen jeder Ereignisprozedur einen Haltepunkt für die erste Zeile hinzu.
- Wir fügen jeder Ereignisprozedur eine Debug.Print-Anweisung hinzu, die den Namen der Ereignisprozedur im Bereich **DEBUG CONSOLE** ausgibt.

In Bild 4 haben wir beides gleichzeitig implementiert.

```
(component)  (general)  (declaration)
 1  [ Description ("") ]
 2  [ FormDesignerId ("2A521B6A-7FE7-4C6C-ABB4-32D750286CA1") ]
 3  [ PredeclaredId ]
 4  Class frmMain
 5
 6      Sub New()
 7      End Sub
 8
 9      Private Sub Form_Activate()
10          Debug.Print "Form_Activate"
11      End Sub
12
13      Private Sub Form_Click()
14          Debug.Print "Form_Click"
15      End Sub
16
17      Private Sub Form_DblClick()
18          Debug.Print "Form_DblClick"
19      End Sub
20
21      Private Sub Form_Deactivate()
22          Debug.Print "Form_Deactivate"
23      End Sub
24
25      Private Sub Form_DragDrop(Source As Control, X As Single, Y As Single)
26          Debug.Print "Form_DragDrop"
27      End Sub
28
29      Private Sub Form_DragOver(Source As Control, X As Single, Y As Single, State As Integer)
30          Debug.Print "Form_DragOver"
31      End Sub
32
33      Private Sub Form_GotFocus()
34          Debug.Print "Form_GotFocus"
35      End Sub
```

Bild 4: Test der Abfolge des Aufrufs der einzelnen Ereignisprozeduren für verschiedene Aktionen

Auf die gleiche Weise lassen sich auch die durch andere Aktionen ausgelösten Ereignisprozeduren und die Reihenfolge des Aufrufs ermitteln – beispielsweise beim Aktivieren, beim Aktivieren eines anderen Elements er Benutzeroberfläche, beim Anklicken oder auch beim Überfahren mit dem Mauszeiger.

Und auch Aktionen wie das Betätigen der Tastatur wird zurückgemeldet.

Die Kenntnis der ausgelösten Ereignisse und ihre Reihenfolge ist wichtig, wenn man diese programmieren möchte.

Wenn wir nun die Anwendung starten und das **Form**-Element **frmMain** wird direkt beim Start geladen, sehen wir beispielsweise das Ergebnis aus Bild 5 in der **DEBUG CONSOLE**.

Schließen wir das Formular wieder, erhalten wir das Ergebnis aus Bild 6.

Hier sehen wir noch den Eintrag **Form_Activate** – dieser entsteht dadurch, dass wir das **Form**-Element zum Schließen anklicken und somit aktivieren müssen (zumindest im Debugging-Modus, wenn zuvor die twinBASIC-Entwicklungsumgebung den Focus hatte).

```
DEBUG CONSOLE
15:14:42.498  [DEBUGGER] global variables have been cleared
15:14:42.514  Executing '[frmMain].Show()'...
15:14:42.520  Form_Initialize
15:14:42.523  Form_Load
15:14:42.529  Form_Activate
15:14:42.532  Form_Resize
15:14:42.533  [DEBUGGER] Waiting for remaining forms to close...
15:14:42.533  Form_GotFocus
15:14:42.533  Form_Paint
>
```

Bild 5: Ausgelöste Ereignisse beim Öffnen eines **Form**-Elements

```
DEBUG CONSOLE
15:17:30.446  Form_Activate
15:17:30.506  Form_QueryUnload
15:17:30.507  Form_Unload
15:17:30.510  Form_Deactivate
15:17:30.515  (time taken: 0.0177521s)
```

Bild 6: Ausgelöste Ereignisse beim Schließen eines **Form**-Elements

twinBASIC: Daten von Form zu Form

Wenn man mit SDI-Forms arbeiten, also mit solchen Formularen, die als einzelne Fenster geöffnet werden, möchte man manchmal eine der folgenden beiden Aufgaben erledigen – oder auch beide: Das aufrufende Formular soll Daten an das aufgerufene Formular übergeben, beispielsweise um den Primärschlüssel eines im Detailformular anzuzeigenden Datensatzes zu übergeben. Oder man möchte ein Formular zum Abfragen von Daten öffnen und diese dann vom aufrufenden Formular aus aufrufen. Wie beides gelingt und welche unterschiedlichen Wege es dazu gibt, schauen wir uns in diesem Artikel an. Außerdem betrachten wir, welche Arten von Daten man grundsätzlich übertragen können sollte.

Es wird immer wieder Anlässe geben, bei denen wir einen Button auf einer Form verwenden, um eine andere Form damit zu öffnen. In manchen Fällen reicht dies aus – wir verwenden die andere Form dann für einen Zweck, der in sich abgeschlossen ist. In vielen Fällen wird es jedoch so sein, dass wir Daten von einem zum anderen Fenster übergeben wollen – und vielleicht auch wieder zurück. Ein Beispiel sind Forms, die an Daten gebunden sind. Wenn wir etwa ein Form-Objekt nutzen, um darauf eine Liste von Produkten anzuzeigen, wollen wir meist auch ein weiteres Form verwenden, um die Details eines einzelnen Produkts zu liefern – beispielsweise um diese bearbeiten zu können. Wenn wir dann einen der Einträge in der Liste markieren und auf eine Schaltfläche klicken, um die Form mit den Details anzuzeigen, müssen wir dieser Form auf irgendeine Weise die Information übergeben, zu welchem Produkt es die Detaildaten anzeigen soll.

Andersherum wollen wir nach dem Bearbeiten des Produkts oder auch nach dem Anlegen eines neuen Produkts und dem Schließen des Detailfensters Informationen an das aufrufende Fenster zurückgeben – zum Beispiel, weil wir den soeben bearbeiteten oder angelegten Datensatz in der Liste markieren wollen oder weil die Änderungen in den Daten direkt in den dort angezeigten Eintrag übernommen werden sollen.

Es gibt auch noch einfachere Beispiele, die es bereits gibt – zum Beispiel das mit der Funktion **MsgBox** angezeigte Meldungsfenster oder das mit **InputBox** angezeigte Eingabefenster für einen einfachen Text.

Wenn wir einmal Daten vom Benutzer abfragen wollen, die zu umfangreich sind als jene, die wir mit **MsgBox** und **InputBox** ermitteln können, müssen wir eigene Forms dafür entwickeln – und die dort ermittelten Daten wollen wir auf irgendeine Weise für die Weiterverarbeitung auslesen.

Weitere Beispiele für das Übertragen von Daten

Beispiele für das Übertragen von Daten zwischen zwei Form-Objekten sind die folgenden:

- Anmeldeformular
- Suchformular
- Einstellungen und Konfigurationen
- Übertragung der ID eines ausgewählten Datensatzes von einer Hauptform zur Bearbeitungsform.
- Workflows über mehrere Forms

Möglichkeiten zur Übertragung von Daten an die zu öffnende Form

Im Folgenden schauen wir uns verschiedene Möglichkeiten an, wie wir Daten vom aufrufenden **Form**-Ob-

jekt zum aufgerufenen **Form**-Objekt und zurück übergeben können:

- Zwischenspeichern in einer globalen Variablen, die von der aufrufenden Form aus gefüllt und von der aufgerufenen Form aus ausgelesen wird und umgekehrt

- Wenn der zu übergebende Wert in einem Steuerelement der aufrufenden Form enthalten ist, können wir dieses von der aufgerufenen Form aus auslesen.

- Übergeben der Informationen von der aufrufenden Form an eine öffentliche Eigenschaft der aufgerufenen Form. Diese kann von der aufgerufenen Form auch geändert werden, damit wir den Wert wieder in die aufrufende Form zurückholen können.

Von Form zu Form mit einer globalen Variablen

Die Verwendung einer globalen Variablen ist ein Weg, mit dem wir Daten zwar nicht direkt von einer Form zur nächsten übertragen, aber wir können so einen Wert von der ersten Form aus so zur Verfügung stellen, dass die aufgerufene Form diesen auch nutzen kann. Der Nachteil ist, dass dieser Wert nicht nur von den beiden Forms gelesen und geschrieben werden kann, die ihn nutzen, sondern auch noch von anderen Stellen aus. Hier ist also sicherzustellen, dass diese Variable nur für diesen speziellen Zweck verwendet wird – beispielsweise durch einen eindeutigen Namen.

Für das Beispiel benötigen wir als Erstes eine globale Variable, in der wir den zu übergebenden Wert speichern können. Diese legen wir in einem neuen Modul namens **mdlGlobal** unter dem Namen **strGlobal** an (siehe Bild 1). Da wir nur Texte von einem Textfeld zum nächsten übertragen wollen, reicht der Datentyp String aus.

Im Formular **frmMain** haben wir ein Textfeld namens **txtValue** erstellt, in das wir den zu übertragenden

Bild 1: Globale Variable

Wert eintragen. Für die Schaltfläche **cmdSendViaGlobalVariable** haben wir den folgenden Code hinterlegt:

```
Private Sub cmdSendViaGlobalVariable_Click()
    If Not Len(txtValue) = 0 Then
        strGlobal = txtValue
        frmGlobalVariable.Show
    Else
        MsgBox "Textfeld ist leer."
    End If
End Sub
```

Dieser prüft, ob überhaupt ein Text enthalten ist – falls nicht, erscheint eine entsprechende Meldung. Anderenfalls schreibt die Prozedur den Inhalt des Textfeldes in die Variable **strGlobal** und ruft die Form **frmGlobalVariable** auf. Diese Form enthält ebenfalls ein Textfeld namens **txtValue** sowie eine Schaltfläche namens **cmdOK**. Das Textfeld **txtValue** soll mit dem Wert aus dem gleichnamigen Textfeld des aufrufenden Formulars gefüllt werden.

Dazu legen wir in der aufgerufenen Form eine Ereignisprozedur für das Ereignis **Load** an. Dieses sieht wie folgt aus:

```
Private Sub Form_Load()
    Me.txtValue = strGlobal
End Sub
```

Damit erhalten wir das Ergebnis aus Bild 2.

Auslesen des zu übergebenden Wertes aus einem Steuerelement

Die nächste Möglichkeit setzt voraus, dass der zu übergebende Wert in einem Steuerelement im aufrufenden **Form**-Element angezeigt wird – wie in unserem Beispiel im Textfeld **txtValue**. Wir rufen dann die zweite Form auf und können dann von dort aus dem Wert des Textfeldes auslesen und in das Textfeld der Zielform eintragen. Dazu fügen wir **frmMain** eine zweite Schaltfläche namens **cmdReadValueFromControl** zu, die folgende Prozedur aufruft:

```
Private Sub cmdReadValueFromControl_Click()
    If Not Len(txtValue) = 0 Then
        frmReadValueFromControl.Show
    Else
        MsgBox "Textfeld ist leer."
    End If
End Sub
```

Sie soll also einfach die Form **frmReadValueFromControl** aufrufen. Für die **Load**-Ereigniseigenschaft dieser Form könnten wir einfach die folgende Ereignisprozedur hinterlegen:

```
Private Sub Form_Load()
    Me.txtValue.Text = frmMain.txtValue
End Sub
```

Das funktioniert, sofern die aufrufende Form **frmMain** aktuell geöffnet ist. Wenn die aufgerufene Form **frmReadValueFromControl** allerdings von einem anderen **Form**-Objekt geöffnet wird und **frmMain** nicht geöffnet ist, erhalten wir einen Fehler.

Diesen können wir allerdings umgehen, indem wir prüfen, ob die Form geöffnet ist. Dazu erweitern wir die Prozedur wie folgt:

```
Private Sub Form_Load()
    If IsFormOpen("frmMain") Then
        Me.txtValue.Text = frmMain.txtValue
    End If
End Sub
```

Die Funktion **IsFormOpen** ist allerdings keine eingebaute Funktion, wir müssen diese selbst programmieren. Die Funktion erwartet den Namen des zu untersuchenden Formulars und gibt den Wert **True** zurück, wenn es geöffnet ist. Sie durchläuft in einer **For...Next**-Schleife alle Zahlen von **0** bis zur Anzahl der geöffneten Formulare minus **1**. In dieser Schleife vergleicht sie den Namen des aktuell durchlaufenen Formulars mit dem zu untersuchenden. Sind beide gleich, ist das Formular geöffnet und die Funktion kann den Wert **True** zurückgeben, bevor sie beendet wird.

```
Public Function IsFormOpen(strForm As String) As Boolean
    Dim i As Integer
    For i = 0 To Forms.Count - 1
        If Forms(i).Name = strForm Then
            Return True
            Exit Function
        End If
    Next i
End Function
```

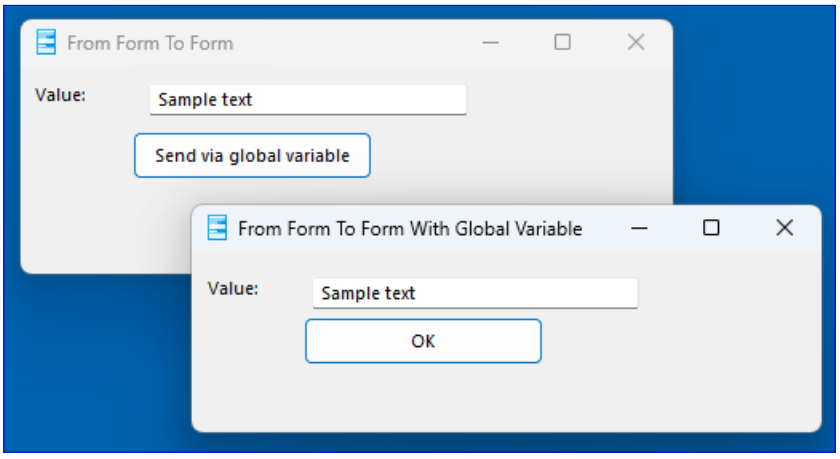

Bild 2: Text per globaler Variable zum aufgerufenen Formular übertragen

Nutzen wir diese Funktion, liest das aufgerufene **Form**-Element also nur das Textfeld des aufrufenden **Form**-Elements aus, wenn dieses aktuell geladen ist.

Das Ergebnis ist das Gleiche wie im vorherigen Beispiel – der Inhalt des Textfeldes **txtValue** der aufrufenden Form landet im gleichnamigen Textfeld der aufgerufenen Form.

Von Form zu Form mit einer öffentlichen Eigenschaft im Zielform

Die nächste Variante verwendet eine öffentliche Eigenschaft, die wir im Code behind-Klassenmodul der Zielform definieren. Die Definition lautet wie folgt:

```
Public Property Let Stringvalue(strValue As String)
    Me.txtValue = strValue
End Property
```

Die Eigenschaft nimmt den zugewiesenen Wert entgegen und trägt diesen in das Textfeld **txtValue** ein (siehe Bild 3).

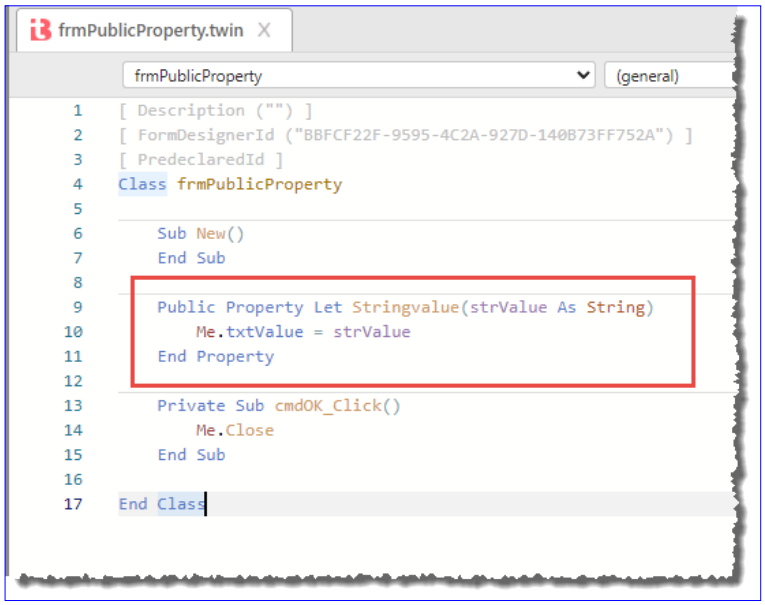

Bild 3: Wert per öffentlicher Eigenschaft übergeben

Im Klassenmodul des aufrufenden **Form**-Elements **frmMain** können wir nun für die Schaltfläche **cmdSetPublicPropertyInTargetform** eine Ereignisprozedur hinterlegen, welche die Eigenschaft **Stringvalue** der Form **frmPublicProperty** mit dem gewünschten Wert füllt. Die Eigenschaft ist nun per IntelliSense auswählbar (siehe Bild 4). Danach zeigen wir **frmPublicProperty** mit der **Show**-Methode an.

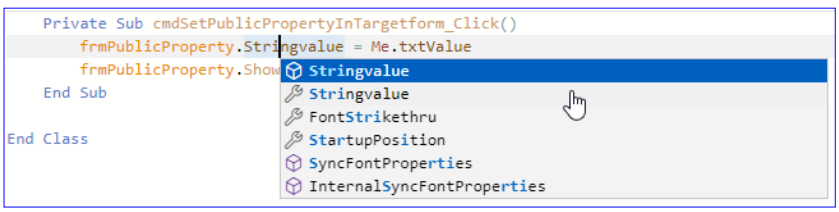

Bild 4: Öffentliche Eigenschaften sind per IntelliSense verfügbar

Werte vom aufgerufenen Form-Element auslesen

Damit kommen wir zur entgegengesetzten Aufgabe: Wir haben ein **Form**-Element aufgerufen, mit dem wir Daten abfragen wollen. Der Aufbau sieht genauso aus wie in den vorherigen Beispielen.

Allerdings wollen wir nun keinen Wert vom aufrufenden Formular an das aufgerufene Formular übergeben, sondern einen im aufgerufenen Formular eingegebenen Wert vom aufrufenden Formular aus auslesen oder zurückbekommen. Hier gibt es wiederum verschiedene Möglichkeiten:

- Speichern des zu ermittelnden Wertes in eine globale Variable, die vom aufrufenden Formular ausgelesen werden kann
- Wenn der Wert sich im aufgerufenen Formular in einem Steuerelement befindet: Auslesen des Inhalts des Steuerelements vom aufrufenden Formular aus

twinBASIC: Menüs erstellen

Während man sich beim Programmieren von Office-Anwendungen wie Word, Excel, Outlook oder Access mit XML-Definitionen für das Ribbon beschäftigen muss, kann man unter twinBASIC zumindest für die Fenstermenüs auf eine einfache und praktische Benutzeroberfläche zurückgreifen. Okay, für Kontextmenüs ist dann doch wieder VB-Code gefragt, aber das ist auch in den meisten Office-Anwendungen noch die gängige Vorgehensweise (nicht in Outlook, dort sind die Kontextmenüs bereits in die Ribbondefinition integriert). In diesem Artikel schauen wir uns erst einmal die Möglichkeiten an, twinBASIC-Anwendungen über die Benutzeroberfläche mit Menüs auszustatten und ihre Eigenschaften per Code anzupassen.

Hauptmenü anlegen

Bereits wenn wir ein Formular in twinBASIC öffnen, ist die Möglichkeit zum Hinzufügen eines Menüs nicht zu übersehen (siehe Bild 1).

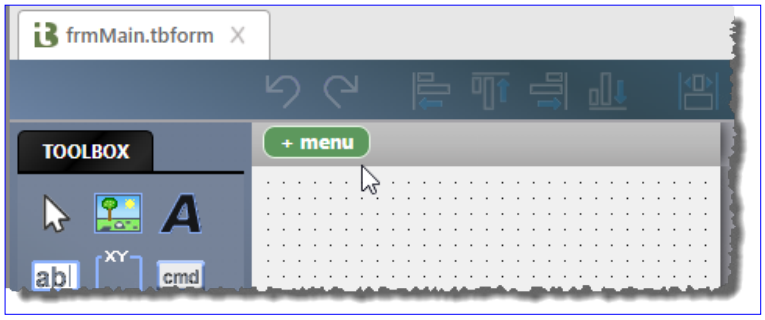

Bild 1: Button zum Anlegen eines Menüs

Klicken wir diese Schaltfläche an, erscheint direkt der neue Menüeintrag und bietet die Möglichkeit, den Namen anzupassen (siehe Bild 2).

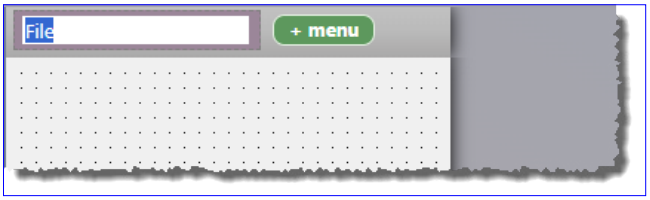

Bild 2: Ändern der Menübeschriftung

Schon jetzt können wir das Projekt starten und den Menüpunkt sehen (siehe Bild 3). Durch wiederholtes Betätigen der Schaltfläche können wir weitere Hauptmenüpunkte hinzufügen.

Menüpunkte hinzufügen

Klicken wir nun auf das ersten angelegten Menü, sehen wir eine weitere Schaltfläche (siehe Bild 4). Mit einem Klick fügen wir einen Menübefehl hinzu.

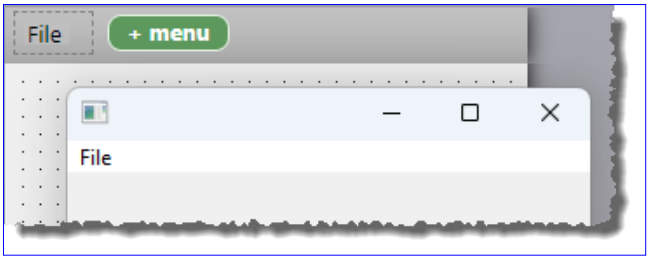

Bild 3: Der erste Menüpunkt im Einsatz

Dieser bietet auch direkt die Möglichkeit zum Umbenennen an (siehe Bild 5).

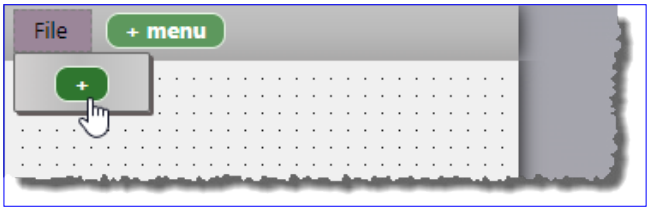

Bild 4: Hinzufügen eines Menüpunktes

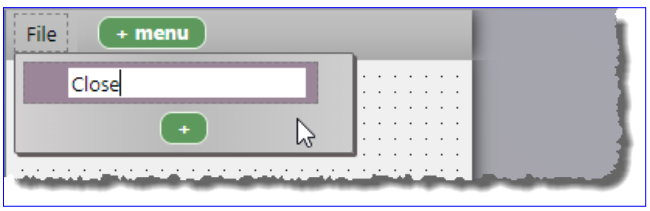

Bild 5: Der neue Menüpunkt

Anlegen von Untermenüs

Bevor wir uns die Anpassung der Menüs und Menüpunkte ansehen, schauen wir uns noch an, wie wir ein Untermenü anlegen. Dazu klicken wir einfach auf einen der Menüpunkte und fügen diesem wie in Bild 6 einen Unterpunkt hinzu.

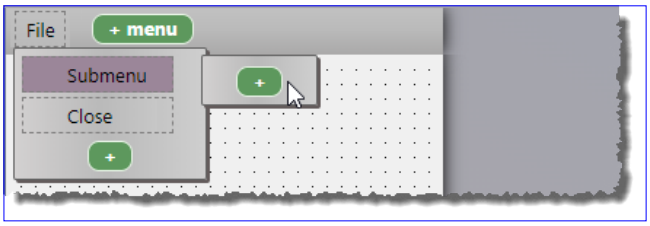

Bild 6: Anlegen eines Menüpunktes in einem Untermenü

Nach dem Start des Projekt steht unser erste Menü bereits (siehe Bild 7). Nun können wir uns um die Anpassung kümmern.

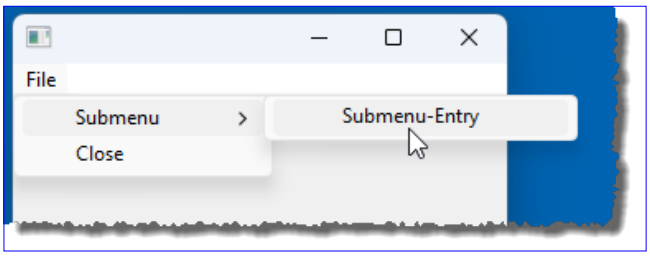

Bild 7: Das Menü in Aktion

Menüs anpassen

Die Haupteinträge, also die Einträge, die wir direkt in der Menüleiste sehen, können wir über ein Kontextmenü bearbeiten, das wie in Bild 8 aussieht.

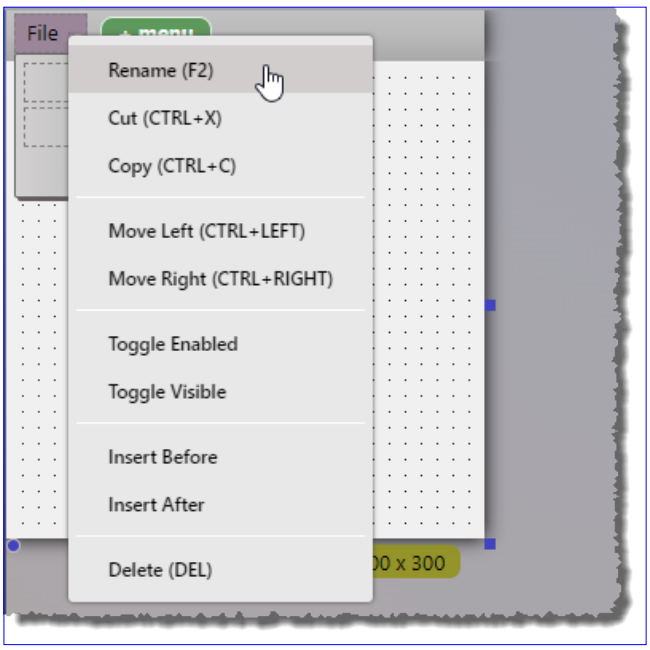

Bild 8: Menüs anpassen per Kontextmenü

Hier können wir Einträge umbenennen, ausschneiden oder kopieren und anschließend wieder einfügen. Dazu stehen (nicht im Bild zu sehen) gleich drei Kontextmenübefehle zur Verfügung:

- **Paste Before**: Fügt das Menü aus der Zwischenablage vor dem aktuellen Eintrag ein.
- **Paste After**: Fügt das Menü aus der Zwischenablage hinter dem aktuellen Eintrag ein.
- **Paste Into (CRTL + V)**: Fügt das Menü aus der Zwischenablage als Untermenü in das aktuelle Menü ein.

Außerdem können wir die Menüpunkte des Hauptmenüs mit den beiden Befehlen **Move Left (CTRL + LEFT)** und **Move Right (CTRL + RIGHT)** nach links oder rechts verschieben.

Mit **Toggle Enabled** stellen wir den Menüpunkt als deaktiviert ein. Wir können diesen dann später per Code aktivieren. Wie können Menüelemente aber auch per Code deaktivieren. **Toggle Visible** erledigt das Gleiche für die **Visible**-Eigenschaft.

Mit **Insert Before** und **Insert After** können wir neue Menüeinträge vor oder hinter dem aktuellen Menüeintrag hinzufügen. Schließlich können wir den aktuell markierten Eintrag mit **Delete (DEL)** wieder löschen.

Menüeinträge anpassen

Für Untermenüs und Menüpunkte liefert das Kontextmenü weitgehend die gleichen Befehle (siehe Bild 9).

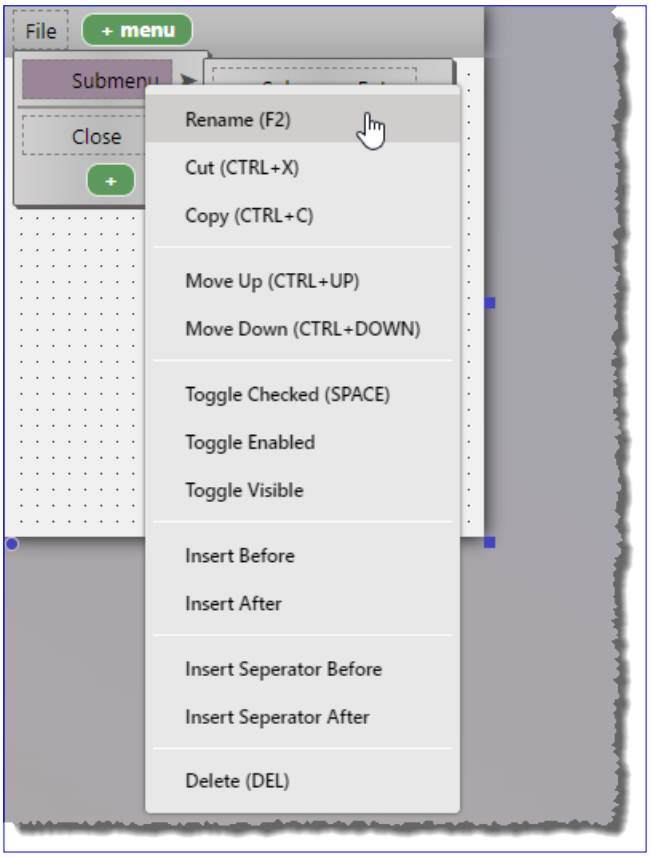

Bild 9: Menüpunkte anpassen per Kontextmenü

Einige kommen noch hinzu, zum Beispiel **Toggle Checked (SPACE)**. Damit können wir festlegen, ob der Eintrag bereits beim ersten Anzeigen mit einem Haken versehen sein soll.

Trennstriche einfügen

Außerdem können wir mit den Menübefehlen **Insert Separator Before** und **Insert Separator After** einen Trennstrich vor oder hinter dem aktuellen Eintrag hinzufügen. Dieser erscheint im Entwurf sowie zur Laufzeit als Trennstrich zwischen zwei Menüpunkten.

Eigenschaften von Menüs einstellen

Die Eigenschaften von Menüs sehen wir im Eigenschaftsbereich (siehe Bild 10). In der aktuellen Version von twinBASIC sind noch nicht alle Eigenschaften implementiert, die es unter VB6 gibt, jedoch die wesentlichen. Hier sind einige davon:

- **Name**: Name des Menüs
- **Caption**: Beschriftung des Menüs
- **Checked**: Gibt an, ob ein Haken angezeigt werden soll.
- **Enabled**: Gibt an, ob das Menü aktiviert angezeigt werden soll.
- **Picture**: Dient der Auswahl eines Icons für das Menü.
- **Tag**: Erlaubt das Angeben von individuellen Informationen.
- **Visible**: Gibt an, ob das Menü sichtbar ist.

Auf der zweiten Registerseite Events im Eigenschaftsbereich sehen wir nur eine Eigenschaft namens **Click**. Dies ist die Ereigniseigenschaft, für die wir eine Proze-

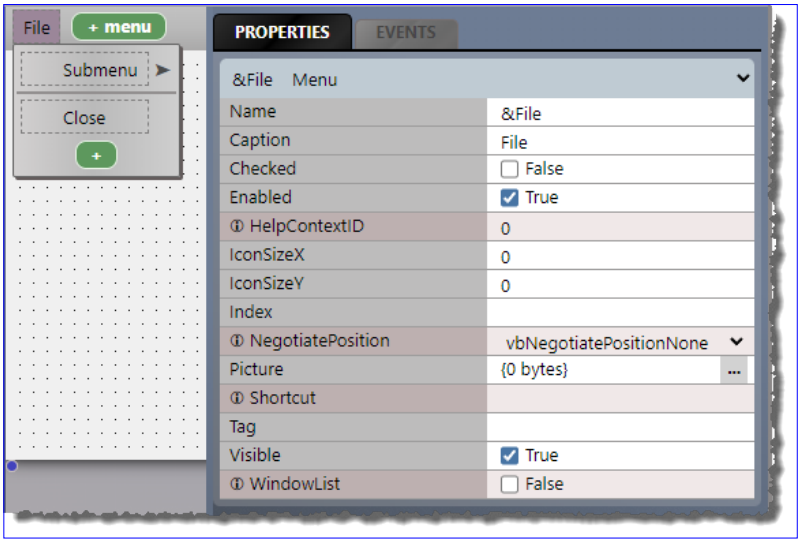

Bild 10: Eigenschaften von Menüs

COM-Add-Ins registrieren

Aus verschiedenen Gründen kann die Installation eines COM-Add-Ins für eine Office-Anwendung fehlschlagen – oder zumindest erreicht man damit nicht das gewünschte Ergebnis. Zum Beispiel könnte ein Benutzer das Setup mit dem COM-Add-In, das normalerweise für den aktuellen Benutzer installiert wird, im Kontext eines anderen Benutzers installieren – zum Beispiel als Administrator. Dann erscheint das COM-Add-In für den Benutzer jedoch nicht. In diesem Artikel zeigen wir zwei schnelle Wege, wie die Installation dennoch gelingt, sofern die DLL-Datei mit dem COM-Add-In vorliegt. Außerdem schauen wir uns an, wo in der Registry der Eintrag für ein COM-Add-In landet und was die verschiedenen Einträge bedeuten.

Setups haben meist den Zweck, die Installation einer Software für den Benutzer so angenehm wie möglich zu machen. Manchmal klappt es allerdings einfach nicht wie gewünscht. Das hängt zum Beispiel damit zusammen, dass die Registry des Benutzers anders aufgebaut ist, als es das Setup erwartet. Dann werden die für den reibungslosen Betrieb erforderlichen Einträge in der Registry aus Sicht des Setups zwar an der richtigen Stelle vorgenommen, aber sie werden vom System nicht berücksichtigt.

Es gibt nämlich bestimmte Registry-Bereiche, die Office-Anwendungen beim Start untersuchen, um zu erfahren, ob COM-Add-Ins oder andere Anwendungen vorliegen, die mit der Office-Anwendung gestartet werden sollen.

In diesem Magazin haben wir bereits einige COM-Add-Ins vorgestellt, die selbst festlegen, an welche Stellen der Registry die notwendigen Einträge geschrieben werden sollen. In dem COM-Add-In aus dem Artikel **COM-Add-In für Word: PDF-Export** (**www.vbentwickler.de/383**) legen wir das beispielsweise wie in Bild 1 fest. Die Konstante **RootRegistryFolder_WORD** nimmt den Basispfad auf, die Funktion **DllRegisterServer** legt die Einträge beim Registrieren der Anwendung in der Registry an.

Wie landen die Einträge in der Registry?

Wenn wir wie in dem Artikel gezeigt mit twinBASIC entwickeln und die Anwendung kompilieren, startet twinBASIC automatisch die Prozedur **DllRegisterServer** und trägt so die Einträge in die Registry ein.

```
Const RootRegistryFolder_WORD As String = "HKCU\SOFTWARE\Microsoft\Office\Word\Addins\" & AddinQualifiedClassName & "\"

▶ run DllRegistration.DllRegisterServer
Public Function DllRegisterServer() As Boolean
    On Error GoTo RegError
    Dim wscript As Object = CreateObject("wscript.shell")
    wscript.RegWrite RootRegistryFolder_WORD & "FriendlyName", AddinProjectName, "REG_SZ"
    wscript.RegWrite RootRegistryFolder_WORD & "Description", AddinProjectName, "REG_SZ"
    wscript.RegWrite RootRegistryFolder_WORD & "LoadBehavior", 3, "REG_DWORD"
    Return True
RegError:
```

Bild 1: Anweisungen zum Anlegen der Registry-Einträge

Wenn wir ein Setup mit InnoSetup erstellen, wie wir es in **Installation mit Inno Setup: Die Basics** (**www.vbentwickler.de/382**) beschreiben, fügen wir dem Eintrag für die DLL-Datei noch den Zusatz **regserver** unter Flags hinzu.

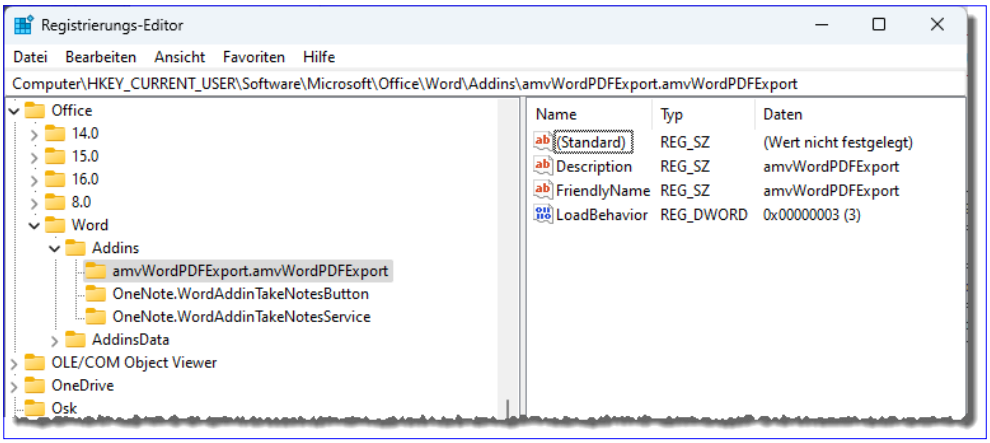

Bild 2: Die resultierenden Registry-Einträge

Dadurch wird die Funktion **DllRegisterServer** automatisch aufgerufen:

```
[Files]
Source: "Build\amWordPDFExport_win32.dll"; DestDir:
"{app}"; Flags: regserver ignoreversion; Check:Is32Bit
```

Hier kann das in der Einleitung beschriebene Szenario auftreten. Wenn die Funktion **DllRegisterServer** in einen Bereich der Registry schreibt, der dem ausführenden Benutzer gehört, landen die Einträge auch in dem Bereich der Registry für diesen Benutzer.

Andere Benutzer auf dem gleichen Rechner können das COM-Add-In somit nicht nutzen.

Das Ergebnis sieht übrigens wie in Bild 2 aus.

Wenn die DLL jedoch schon auf dem Rechner ist, haben wir folgende Möglichkeiten:

- Der Benutzer führt das Setup einfach im Kontext seines eigenen Benutzerkontos aus.
- Der Benutzer nutzt den Befehl **regsvr32.exe** zum Registrieren.
- Der Benutzer registriert das COM-Add-In über die Benutzeroberfläche des jeweiligen Office-Produkts.

Den ersten Punkt brauchen wir nicht weiter zu erläutern. Die anderen beiden schauen wir uns im Anschluss an.

Registrieren mit regsvr32.exe

Um dies zu erledigen, benötigen wir den Pfad zu der zu installierenden COM-Add-In-Datei. Wenn zuvor bereits ein Setup gelaufen ist, wurde darin festgelegt, wo die Datei gelandet ist – meist in einem Ordner unterhalb von **Programme** oder **Programme (x86)**.

Wenn wir den Pfad der **.dll**-Datei ermittelt haben, können wir die Eingabeaufforderung öffnen, am Besten im Administratormodus. Dazu geben wir cmd in das Windows-Suchfeld ein und klicken mit der rechten Maustaste auf den nun erscheinenden Befehl **Eingabeaufforderung**. Aus dem Kontextmenü wählen wir dann den Eintrag **Als Administrator ausführen** aus.

Es erscheint die Eingabeaufforderung. Wir verschaffen uns zunächst einen Überblick über die Optionen des Befehls **regsvr32**, indem wir diesen Befehl ohne Parameter eingeben. Die Parameter werden dann in einem Meldungsfenster angezeigt (siehe Bild 3).

Für uns ist zunächst nur ein Parameter wichtig: die Eingabe der zu registrierenden **.dll**-Datei. Später können wir gegebenenfalls noch den Parameter **/u** nutzen.

Damit geben wir an, dass die Registrierung wieder aufgehoben werden soll.

Rufen wir nun beispielsweise den folgenden Befehl auf:

```
regsvr32 [Pfad]\amvWordPDFExport.
dll
```

Damit erhalten wir das Ergebnis aus Bild 4 und die COM-DLL ist nun im Kontext des aktuellen Benutzers registriert.

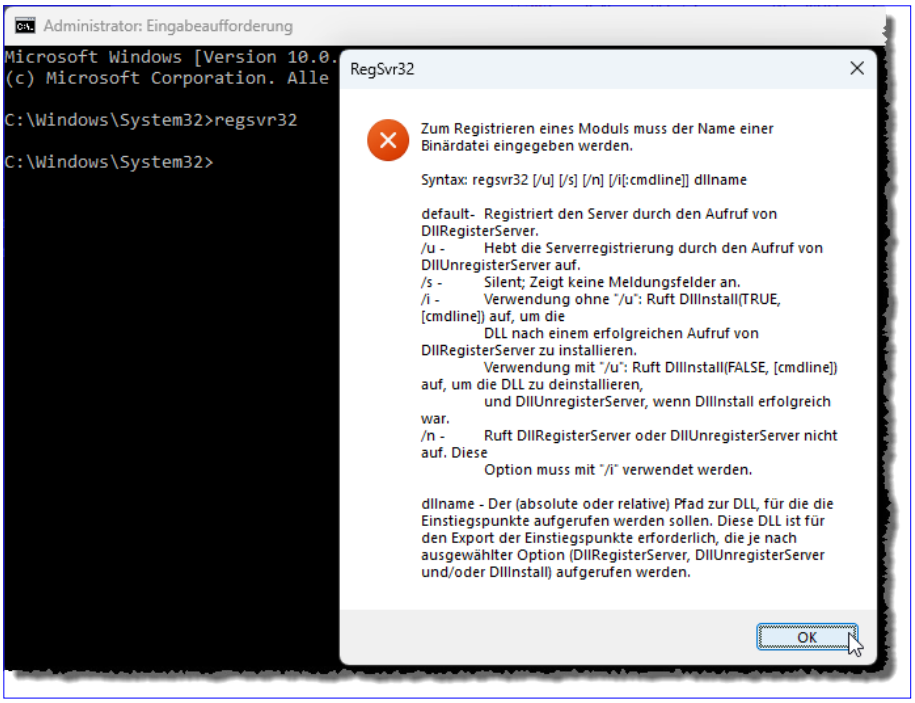

Bild 3: Optionen des Befehls **regsvr32.exe**

Registrieren über die Benutzeroberfläche

Eingangs haben wir auch noch davon gesprochen, dass das Registrieren über die Benutzeroberfläche erfolgen kann – also selbst ohne Eingabe von Befehlen in die Eingabeaufforderung. Um dies auszuprobieren, deregistrieren wir das COM-Add-In erst einmal wieder. Dazu verwenden wir nun den folgenden Befehl:

```
regsvr32 "[Pfad]\amvWordPDFExport_win32.dll" /u
```

Auch hier erhalten wir als Ergebnis eine Meldung (siehe Bild 5). Schauen wir uns nun die entsprechende Stelle der Registry an, stellen wir fest, dass der Eintrag für **amvWordPDFExport** verschwunden ist.

Damit steigen wir in die Registrierung direkt über die jeweilige Anwendung ein. Dazu öffnen wir in diesem

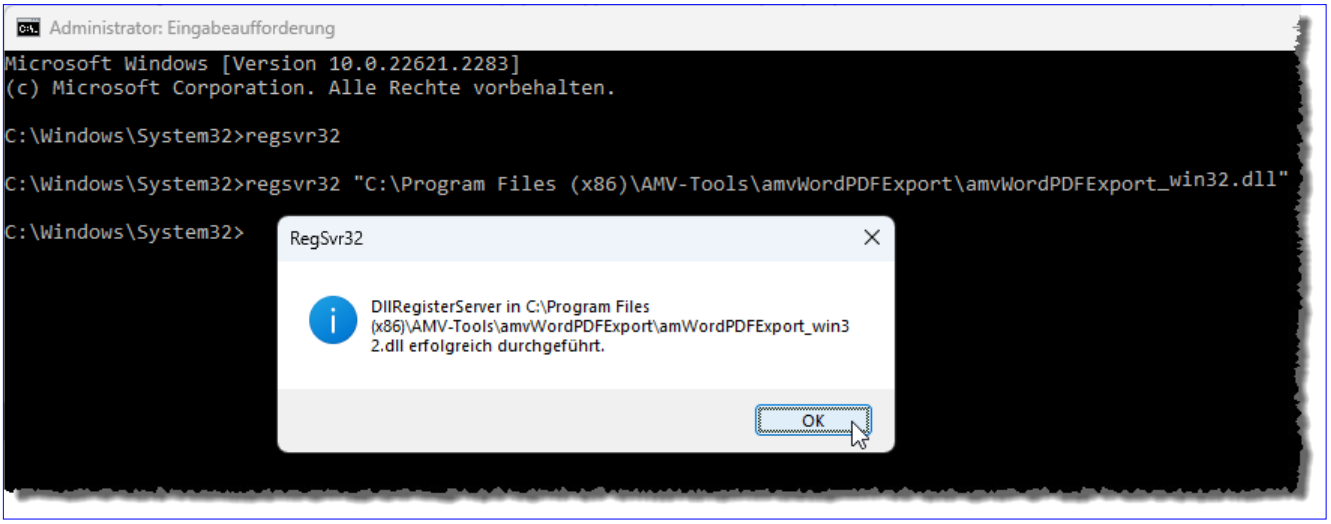

Bild 4: Erfolgreiche Registrierung des COM-Add-Ins