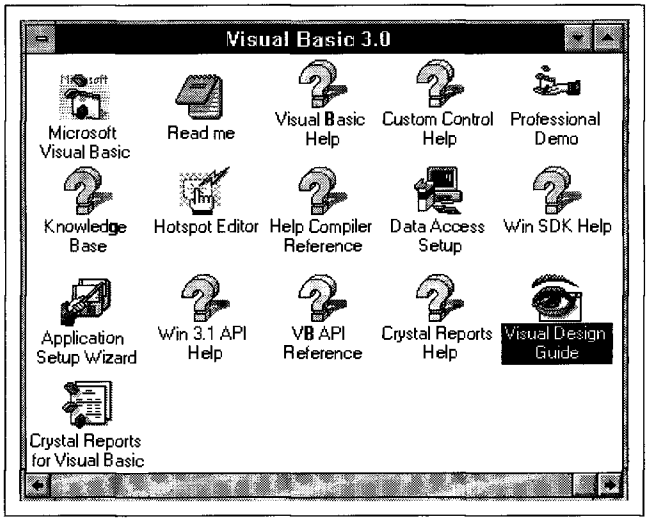


Programmieren für Juristen Visual Basic – Lektion V

Maximilian Herberger

"Nachlieferung" zu Folge IV

Aufmerksame Leser haben es schnell gemerkt, andere haben es für eine Denksportaufgabe gehalten, es war aber "nur" eine Panne:
In der letzten Folge wurden die ersten drei Abbildungen nicht richtig einmontiert. Deshalb hier zuerst – zugleich als Einstieg in die heutige Folge, da am selben Formular weitergearbeitet wird – die Abbildungen 1 bis 3 aus Folge IV:

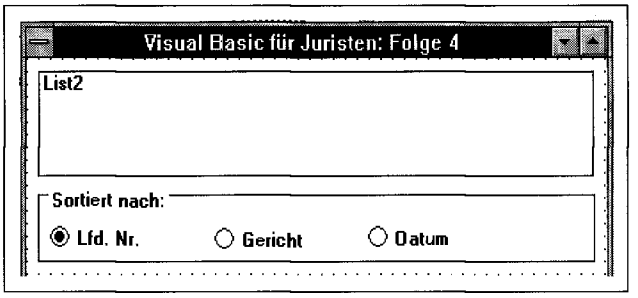


Abb. 1: Formular mit umsortierbarer Liste

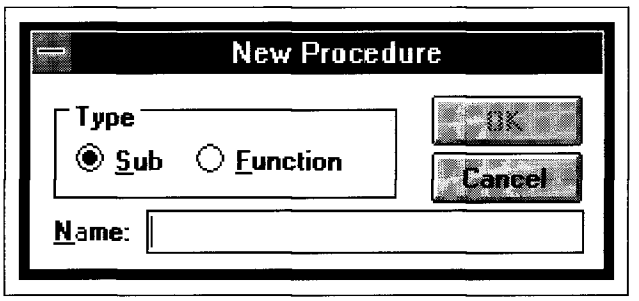


Abb. 2: "Definitionsfenster" für eine eigene Prozedur

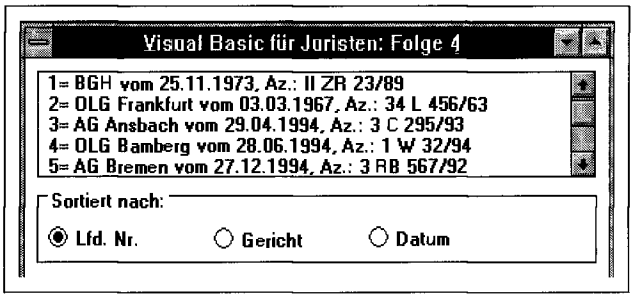
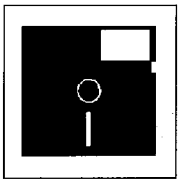
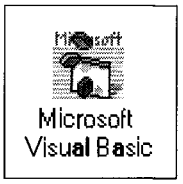


Abb. 3: "Gefüllte" Liste nach Programmstart





Arbeiten mit Index-Dateien: Neue Datensätze hinzufügen

Bei unserem Arbeiten mit Indexdateien steht eine Probe auf's Exempel noch aus: Das Hinzufügen und das Löschen von Datensätzen. In beiden Fällen müssen alle Indexdateien aktualisiert werden. Beginnen wir mit dem Hinzufügen von Datensätzen.

Um Datensätze hinzufügen zu können, benötigen wir eine Erfassungsmaske. Diese sollte allerdings nicht immer sichtbar sein, weil dadurch das Formular bei sonstigen Arbeiten mit der Datenbank als überladen erscheinen würde. Die Erfassungsmaske soll also nur für den Fall des Erfassens erscheinen. Um das zu erreichen, wird zunächst die Erfassungsmaske angelegt und dann teils unsichtbar, teils sichtbar ausgestaltet.

Die Erfassungsmaske

Die Erfassungsmaske wird in einer "picture box" untergebracht. Das erlaubt es dann, die gesamte Maske – da in einer "picture box" enthalten – auf einmal zu- und wegzuschalten. Ermöglicht wird dieses Verfahren dadurch, daß die "picture box" ein sogenanntes "container-Element" ist, d. h. sie kann andere Objekte in sich aufnehmen (hier die Felder für die Datenerfassung). Es ist also zunächst eine "picture box" auf dem Formular zu plazieren und mit den Textfeldern für die Datenerfassung (samt Labels für die Feldbezeichner) aufzufüllen. Das Ergebnis (es beschränkt sich auf die Felder "Laufende Nummer", "Gericht", "Datum" und "Aktenzeichen") ist aus Abb. 5.1 ersichtlich.

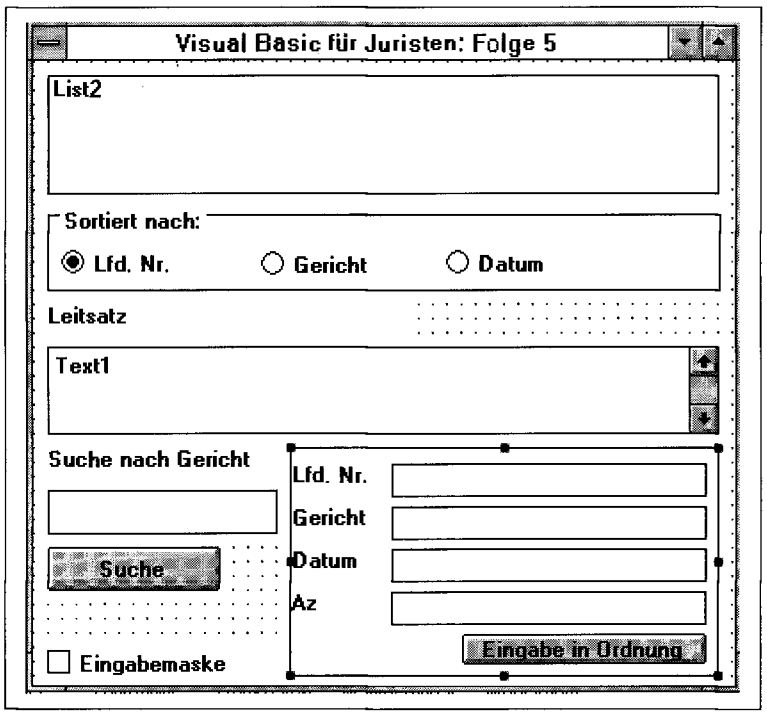


Abb. 5.1: Formular mit "picture box" für Dateneingabe

Die "check box" mit dem Eintrag "Eingabemaske" schaltet die "picture box" mit den Eingabefeldern zu- oder weg. Bewirkt wird das durch folgenden einfachen, sich selbst erklärenden Code:

```
REM Aus sichtbar wird unsichtbar ...
If picture1.Visible = True Then
    picture1.Visible = False
    Else
REM ====
REM ... und umgekehrt
    picture1.Visible = True
REM =================
    Text3.Setfocus
End If
```

Text3.Setfocus plaziert nach dem Aufbau des Erfassungsformulars den Cursor im Feld "Text3" (dem für die laufende Nummer). Mit *Setfocus* kann man also kontextabhängig bestimmten Elementen den "Focus" geben, d.h. sie für die Eingabe oder Auslösung anbieten. Erfaßt wird der neue Datensatz durch die Ausführung der Befehle, die hinter dem Befehlsknopf "Eingabe in Ordnung" liegen. Sie sehen folgendermaßen aus:

```
Table1.AddNew
Table1("lfdnr") = Text3.Text
Table1("gericht") = text4.Text
Table1("datum") = text5.Text
Table1("az") = text6.Text
Table1.Update
Liste_Auffrischen
picture1.Visible = False
```

Das ist nur wenig erläuterungsbedürftig: *Table1.Addnew* fügt der Tabelle einen neuen leeren Datensatz hinzu. Die folgenden Zeilen füllen die Felder des Datensatzes mit den Werten auf, die in die Textfelder der Eingabemaske eingegeben wurden. *Table1.Update* schließlich speichert den neuen Datensatz ab.

Arbeiten mit Index-Dateien: Datensätze löschen

Um Datensätze löschen zu können, fügen wir dem Formular einen Befehlsknopf "Aktuellen Datensatz löschen" hinzu. Als aktueller Datensatz wird der Datensatz betrachtet, der in der Liste durch Click aktiviert wurde und (im Normalfall) als blau hinterlegt erscheint. An dieser Stelle ist aber Vorsicht geboten: Der Click auf die Liste bewegt den Datensatzzeiger nicht. Würde man also einfach löschen, so würde der Datensatz gelöscht, bei dem der Datensatzzeiger auf Grund der vorherigen Operation stehengeblieben ist. Das ist nicht erwünscht. Man muß deshalb vor dem Löschen den Datensatzzeiger zu dem Datensatz bewegen, der der hervorgehobenen Zeile in der Liste entspricht. Zu diesem Zweck kann man denselben Code einsetzen, der für die Leitsatzanzeige verwendet wurde *(vgl Folge IV, S. 3245 unten)*. Da es nur wenige Zeilen sind, sei es gestattet, ihn einfach zu duplizieren. Wer wirklich kunstgerecht arbeiten will, könnte daraus eine Prozedur machen *(vgl. Folge IV, S. 3244 f.)*. Somit erhält der "Lösch-Code" folgendes Aussehen:

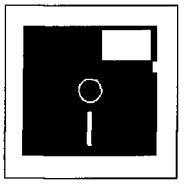
```
REM Bewegung des Datensatzzeigers zum aktuellen
REM (= "angeclickten") Datensatz
gleich = InStr(list2.Text, "=")
ziel = Val(Mid$(list2.Text, 1, gleich - 1))
Table1.Index = "lfdnr"
Table1.Seek "=", ziel
REM ================
REM Löschen des Datensatzes
Table1.Delete
REM ======================
REM Bewegung an den Anfang der Datenbank Table1.MoveFirst
REM ================
REM Auffrischen der Listenanzeige Liste_Auffrischen
```

Die nötigen Kommentare sind in den Code eingearbeitet. Der Löschbefehl lautet *Table1.Delete*. Er entfernt die Datensätze nicht physikalisch aus der Datenbank, sondern merkt sie - wie von dBASE her gewohnt - nur zur Löschung bei einer nachfolgenden Reorganisation vor. Will man die Datenbank reorganisieren, muß man dies hier selbst programmieren. Aus der Liste werden derartige "gelöschte" Datensätze aber ohne besonderes Zutun ausgeblendet, der Befehl *"Table1.Movenext"* überspringt sie einfach. Enthält die Datenbank sehr viele zur Löschung vorgemerkte Datensätze, wird das zeitkritisch und eine Reorganisation unumgänglich.

Menü-Gestaltung

Nachdem die wesentlichen Elemente der Datenbankprogrammierung vorgestellt worden sind, sei ein kurzer Abschnitt einer wesentlichen Frage der äußeren Gestaltung von Programmen gewidmet. Bisher wurden alle Aktionen durch "Knöpfe" oder andere Elemente im Formular ausgelöst. Zum Standard gehört aber, daß auch Menus verfügbar sind. Diese einzubauen, ist in Visual Basic dermaßen leicht, daß es nur weniger Worte der Erläuterung bedarf.

Man ruft das entsprechende Werkzeug über "Window" und "Menu Design" oder direkt mit Ctrl+M auf *(vgl. Abb. 5.2 auf der gegenüberliegenden Seite)*.



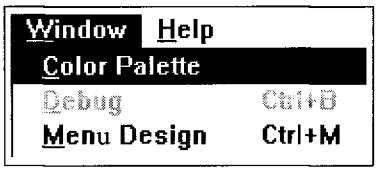


Abb. 5.2:
Aufruf des Menu-"Designers"

Es erscheint dann das "Menu Design Window" *(vgl. Abb. 5.3)*. Es soll hier nicht näher erläutert werden. Es ist weitgehend selbsterklärend, und wo dies nicht der Fall ist, hilft die kontextsensitive Hilfe schnell weiter. Im übrigen kann man ohne Gefahr verschiedene Einstellungen erproben - das Ergebnis ist sofort sichtbar. Nur soviel sei gesagt: Jeder Menu-Punkt *muß* einen Namen haben. Und die Hervorhebung eines Buchstabens für die Direktanwahl erreicht man durch & vor dem betreffenden Buchstaben.

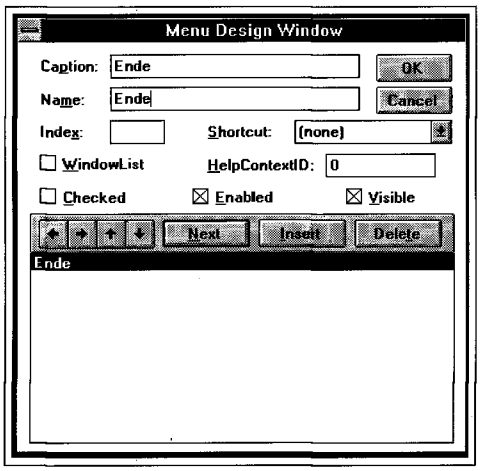


Abb. 5.3:
Das Werkzeug zur Menu-Gestaltung

Als Grundlage für die weiteren Bemerkungen wurde ein Menu-Punkt "Ende" eingefügt, der das Programm beendet. Das zugehörige Code-Fenster erreicht man durch Doppelclick auf den Menu-Punkt. Es ist immer angeraten, vor dem Verlassen eines Programms die verwendete Datenbank zu schließen. Folglich fügt man ein:

```
REM Datenbank schließen
Table1.Close
REM ========
REM Programm verlassen
End
```

Vielfach sind Funktionen sowohl im Menu als auch mit Elementen im Formular repräsentiert. Es ist dann in einfacher Weise möglich, im Menu eine auch anderweitig im Formular vorhandene Funktion aufzurufen. Demonstrieren wir das am Beispiel des Löschens eines Datensatzes. Dies geschieht in unserem Fall durch Click auf das Element *Command1*. Die entsprechende Unterroutine heißt demgemäß *Command1_Click*. Wer diese Unterroutine aus einem Menu-Punkt heraus aufrufen will schreibt dort einfach *Command1_Click* - das ist alles.

Ein interaktives Hilfesystem

Nachdem das Programm eine Menu-Leiste erhalten hat, bleibt noch ein anderes Umgebungselement zu gestalten, das mittlerweile zum Standard gehört: Das Hilfesystem. Dazu sind folgende Schritte notwendig:

- Schreiben einer Hilfe-Quelldatei.
- Kompilieren der Hilfe-Quelldatei zu einer .HLP-Datei.
- Verbinden des Anwendungsprogramms mit der Hilfedatei.

Diese Schritte werden im folgenden dargestellt, allerdings nur bis zu dem Punkt, an dem man Verfeinerungen auf der Grundlage der Dokumentation unschwer selbst einbringen kann *(vgl. Microsoft Visual Basic, Professional Features Book 1, Help Compiler Guide)*. Es entsteht also nur ein minimales Hilfesystem zur Demonstration der Grundprinzipien.

Schreiben einer Hilfe-Quelldatei

Eine Hilfe-Quelldatei muß als RTF-Datei geschrieben werden. Dafür kommt jeder Editor in Frage, der RTF-Dateien erzeugen kann. Im Beispiel wird Word für Windows verwandt, um die aus Abb. 5.4 (auf der nächsten Seite) ersichtliche Hilfe-Quelldatei zu schreiben.
Wie ersichtlich beinhaltet der Hilfetext ein Inhaltsverzeichnis und zwei Unterpunkte. Jede dieser drei Einheiten (sog. "topics") muß mit einem Seitenende abschließen.
Die weiteren Strukturinformationen sind in benutzerdefinierten Fußnoten enthalten, die mit Einfügen-Fußnote-benutzerdefiniert erstellt werden.

Die #-Fußnote: Name

Die #-Fußnote gibt dem einzelnen Topic einen individuellen Namen. Diese Namen müssen singulär sein, sie dürfen nicht mehrfach vorkommen.

Die $-Fußnote: Titel

Die $-Fußnote weist dem Topic einen Titel zu. Dieser Titel wird in der Ergebnisliste von Suchen angezeigt, die auf die Schlüsselworte (dazu sogleich bei der Erläuterung der K-Fußnote) zugreifen.

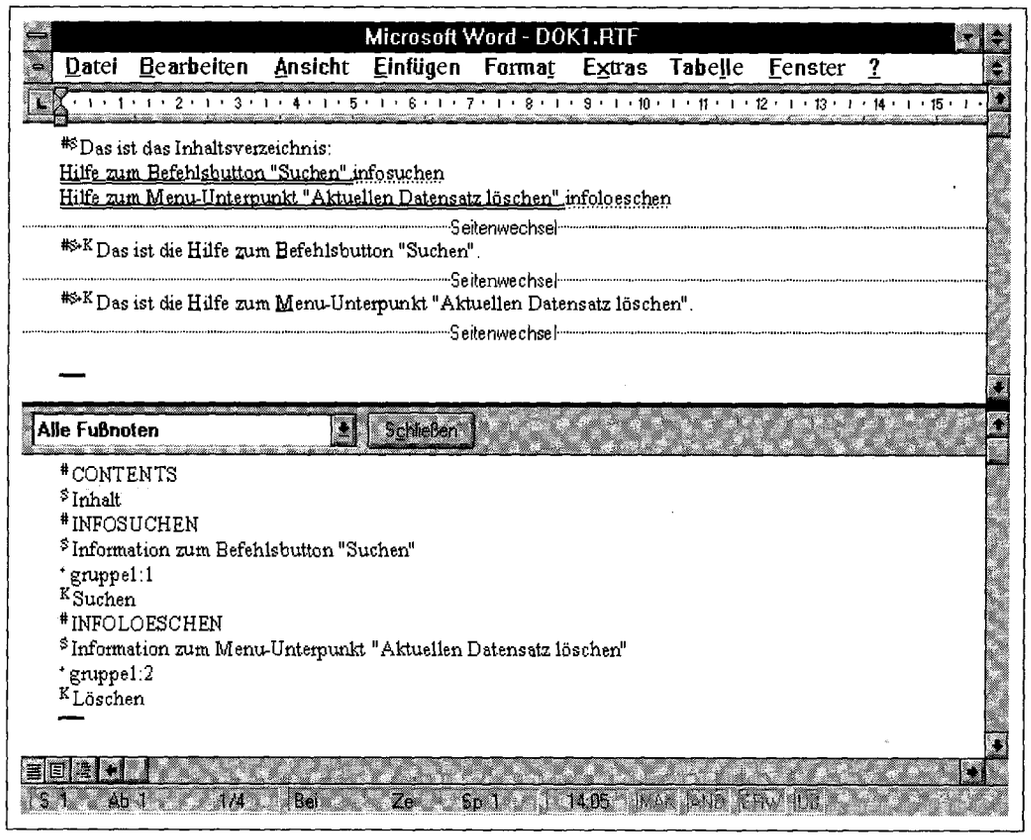


Abb. 5.4:
Die Miniatur-Hilfe-Quelldatei

Die +-Fußnote: Reihenfolge
Die +-Fußnote legt die Blätter-Reihenfolge *(browse sequence)* fest. Im Beispielsfall gehören der zweite und dritte Topic zu einem "Gruppe1" genannten Zusammenhang und haben dort die Position 1 und 2. Der Gruppenname steht vor dem Doppelpunkt, die Position in der Gruppe nach dem Doppelpunkt.

Die K-Fußnote: Schlüsselwort
Die K-Fußnote ordnet einem Topic Schlüsselworte zu, auf die später bei der Suche zugegriffen werden kann. Es erscheinen als Ergebnis der Suche in einer Ergebnisliste die Titel *(vgl. oben zur $-Fußnote)* der Dokumente mit dem betreffenden Schlüsselwort.

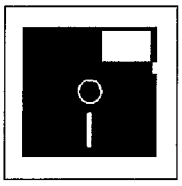
Hypertext
Ein wenig Hypertext enthält die Quelldatei auch: Aus dem Inhaltsverzeichnis kann zu den beiden Ziel-Topics "gesprungen" werden.
Jeder Hypertext besteht aus Start und Ziel. Der Startpunkt wird dadurch kenntlich gemacht, daß man ihn als doppelt unterstrichen formatiert. Der Zielpunkt folgt direkt danach, und zwar als verborgener Text formatiert. Um den Zielpunkt individuell festzulegen, wird der Name des Topics verwandt, wie er in der #-Fußnote bestimmt ist.

Das Kompilieren der Hilfe-Quelldatei
Um die Hilfe-Quelldatei zu kompilieren, benötigt man den Windows-Hilfe-Kompiler, das Programm HC31.EXE. Es wird mit Visual Basic Professional ausgeliefert und befindet sich, hat man es mit installiert, in einem Unterverzeichnis \HC (für Help Compiler).
Verlauf und Ergebnisse des Kompilierens werden durch eine .HPJ-Datei definiert, die sog. Projektdatei. Es handelt sich um eine ASCII-Datei. Sie sieht für das Minimal-Hilfesystem wie folgt aus:

```
[options]
Root=D:\msoffice\winword
title=Hilfe

[baggage]

[config]
browsebuttons()
```




```
[files]
dokl.rtf

[map]
CONTENTS 1
INFOSUCHEN 2
INFOLOESCHEN 3

[windows]
main = "Hilfe",(1,1,512,912),0
```

[options]
Unter [options] sind das Verzeichnis, in dem die Quelldatei gesucht wird, und der Titel des Hilfesystems angegeben.

[baggage]
Unter [baggage] könnte man weitere vom Hilfesystem verwandte Dateien (etwa Bild-Dateien) angeben. Diese werden dann in die .HLP-Datei (das Ergebnis des Kompilierens) übernommen.

[config]
[config]-Einträge entscheiden über verschiedene Konfigurationsdetails. Wir wollen blättern (aus diesem Grunde wurde mit der +-Fußnote die Blätter-Reihenfolge festgelegt) und deswegen benötigen wir die Buttons für das Blättern. *browsebuttons()* fügt diese hinzu.

[files]
Unter [files] steht der Name der Quelldatei, die im mit [root] spezifizierten Verzeichnis gesucht wird. Es können auch mehrere Quelldateien eingetragen werden.

[map]
Für die Verknüpfung des Hilfesystems mit dem Anwendungsprogramm sind die Eintragungen unter [map] entscheidend. Hier erscheinen zunächst die Namen der Hilfstopics, so wie sie in der #-Fußnote festgelegt sind. Die Zahlen dahinter (auch sie dürfen nur je einmal vorkommen) werden vom Visual Basic-Programm benutzt, um bei Betätigen der F1-Taste den entsprechenden Topic anzuspringen.
Eingetragen wird die Zahl unter *HelpContextID*. Für den Menu-Punkt "Aktuellen Datensatz löschen" ist die 3 einzutragen, wie aus Abb. 5.5 ersichtlich: Der Topic heißt "INFOLOESCHEN", per [map] wurde ihm die 3 als Kennziffer zugewiesen.

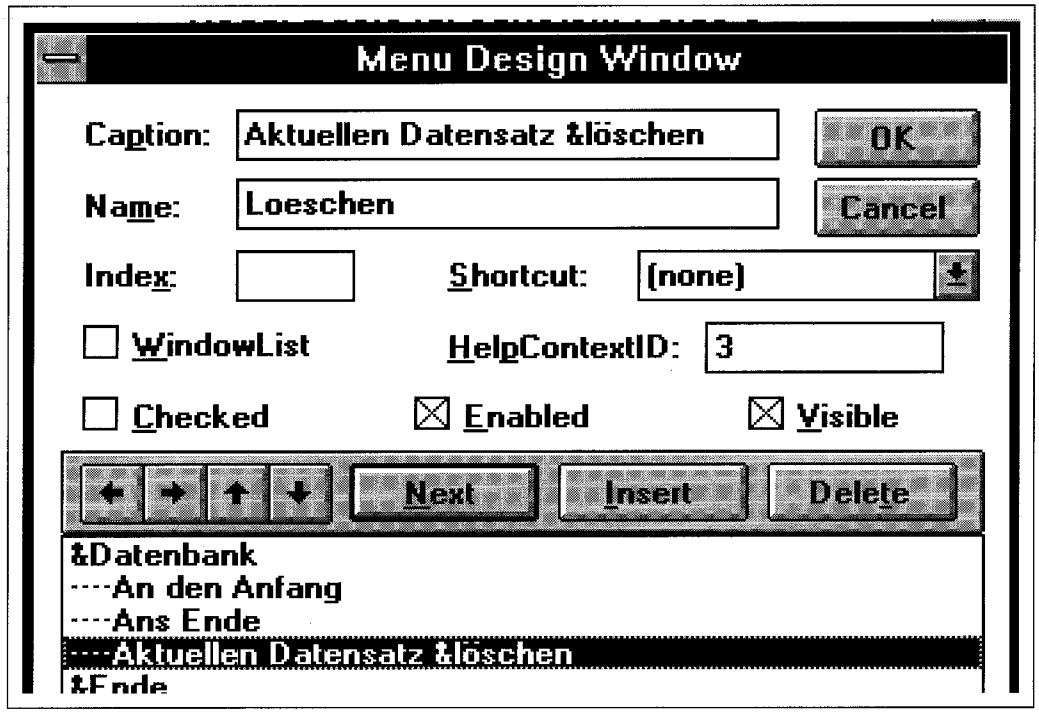


Abb. 5.5:
HelpContextID für Menu-Punkt

Bei auf dem Formular plazierten Elementen wird *HelpContextID* im Properties-Fenster festgelegt *(vgl. Abb. 5.6 für den Befehlsknopf "Suche")*.

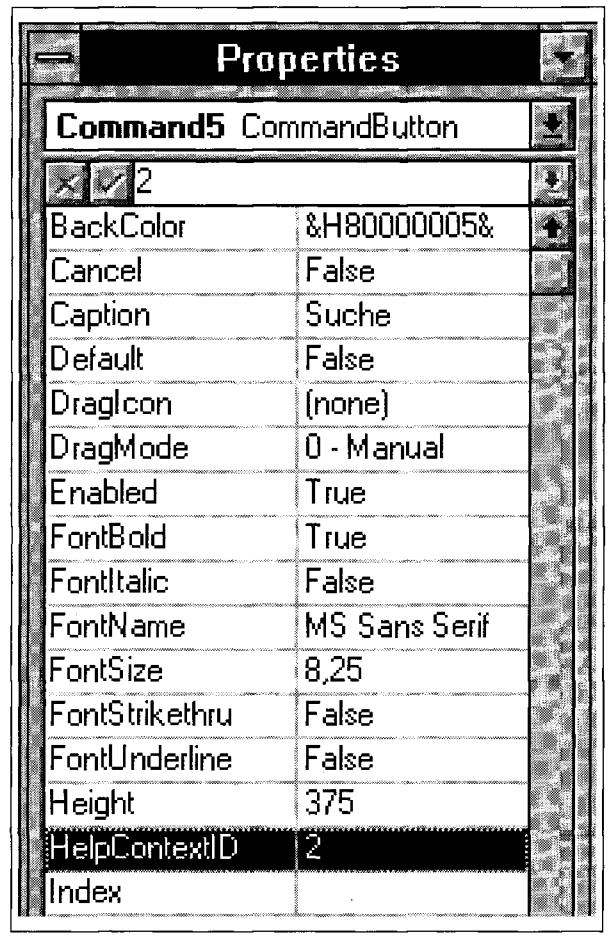


Abb. 5.6: HelpContextID für Bedienelement

[windows]
[windows] entscheidet über Fenster-Einstellungen und -modalitäten.

Das Kompilieren
"After you've created the topic files and project file, the job of building the Help file is simple" (Microsoft Visual Basic, Professional Features Book 1, Help Compiler Guide, S. 125), beruhigt uns das Handbuch. Für unser einfaches Hilfesystem stimmt das. Wird die Angelegenheit etwas komplexer, wird man mit den Fehlermeldungen des Kompilers manchmal unliebsame Bekanntschaft machen. Wir aber kompilieren mit

```
hc31 dok1.hpj
```

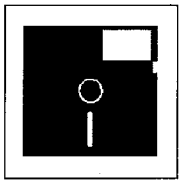
und erhalten als Ergebnis eine .HLP-Datei, die man etwa durch Doppelclick aus dem Dateimanager bereits selbständig aufrufen kann.

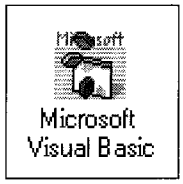
Verbinden des Anwendungsprogramms mit der Hilfedatei
Der Aufruf der Hilfe-Datei funktioniert aus der Visual Basic-Anwendung heraus erst, wenn dort unter *Options-Project* noch Verzeichnis und Name der Hilfe-Datei angegeben wurden. Setzt man dann beispielsweise den Focus auf den Knopf "Suche" (mit TAB, bis der Knopf "hervorgehoben" erscheint, was durch eine gepunktete Linie um "Suche" angezeigt wird) und betätigt F1, so erscheint der entsprechende Hilfe-Eintrag.

Zum Schluß: Etwas (über) Windows-Kommunikation
Der Zugriff auf die Hilfe-Datei erfolgt durch den Aufruf des Windowsprogramms WINHELP.EXE. WINHELP.EXE wird danach nicht nur aktiv, sondern bleibt auch aktiv. Man kann sich leicht davon überzeugen, indem man das Visual Basic-Programm verläßt und dann in die Task-Liste schaut (mit Ctrl+ESC): Der Task "Hilfe" (so war das Hilfesystem genannt worden) ist noch aktiv. Das legt den Wunsch nahe, dem Programm WINHELP.EXE beim Verlassen des Anwendungsprogramms "mitzuteilen", daß die Hilfedatei nicht mehr benötigt wird und geschlossen werden soll. Man kann das als einen einfachen Fall von Inter-Prozeß-Kommunikation ansehen, für die Visual Basic vorbereitet ist.
Um mit anderen Windows-Anwendungen Verbindung aufzunehmen, muß man sich des Windows-API (für *Application Programming Interface*) bedienen. Dieses API enthält Funktionen. Um sie nutzen zu können, muß man sie in einer .BAS-Datei (einer ASCII-Datei), die ansonsten denselben Namen wie die Applikation trägt, deklarieren und diese .BAS-Datei über *File, Add File* mit dem Projekt verbinden. Wie eine solche Deklaration auszusehen hat, ist jeweils in der Dokumentation beschrieben. In unserem Fall lautet sie (entgegen dem Layout hier als eine lange Zeile geschrieben):

Die Funktion "Winhelp": Deklaration

```
Declare Function winhelp Lib "User" (ByVal hWnd As Integer, ByVal
lpHelpFile As String, ByVal wCommand As Integer, dwData As Any) As
Integer
```





Entnommen wird die Funktion der Windows-Bibliothek "USER" ("Lib" steht für "Library"). Für die Detail-Erläuterung muß auf das Handbuch verwiesen werden *(vgl. Microsoft Visual Basic, Professional Features Book 1, Help Compiler Guide, S. 160 ff.)*. Nur soviel muß man an dieser Stelle wissen: Die Funktion hat Parameter (hier: vier). Jeder einzelne davon bekommt einen Namen (hier der Reihenfolge nach: hWnd, IpHelpFile, wCommand und dwData). Zugleich wird für jeden Parameter ein Typ festgelegt (hier der Reihenfolge nach: Integer, String, Integer, Any). Im folgenden soll lediglich die anwendungspraktische Handhabung für zwei Zielsetzungen beschrieben werden.

Ziel Nr. 1:
Schließen einer offenen Hilfe-Datei bei Programmende
Die erste Zielsetzung ist die bereits genannte: Beim Verlassen des Anwendungsprogramms soll eine etwa noch offene, von diesem Anwendungsprogramm angestoßene Hilfe-Datei geschlossen werden. Zu diesem Zweck fügen wir folgende Zeile in den Beendigungscode (Menu-Option "Ende") ein:

Die Funktion "Winhelp": Aufruf
```
R = winhelp(hwnd, dummy$, HELP_QUIT, 0)
```

Eine Funktion ist dadurch gekennzeichnet, daß sie einen Wert "zurückliefert". Deswegen muß der Aufruf sich als "Gleichung" darstellen. Links vom Gleichheitszeichen steht der Name der Variablen, die die "Rückmeldung" aufnimmt. Die "Rückmeldung" hat informativen Wert (jeweils der Dokumentation zu entnehmen), weswegen man im weiteren Verlauf mit dieser Variablen arbeiten kann. Kommt etwa 0 zurück, so war der Aufruf erfolglos, und man muß eine Fehlerbehandlungsroutine anschließen.
Sodann hat eine Funktion, wie bereits bei der Deklaration behandelt, eine bestimmte Stellenanzahl (Parameteranzahl), vier im Falle von WINHELP. Der Aufruf muß alle diese Stellen mit Werten belegen.

Die Funktion "Winhelp": Die Parameter
hWnd ist ein Parameter, der erkennen läßt, von welchem Fenster der "Hilferuf" ausging. Visual Basic teilt diesen Wert mit, da jedes Formular einen solchen Wert mit sich führt, hWnd kann also einfach stehenbleiben. *hWnd*
Der zweite Funktionsparameter (der String *lpzFileName*) ist der Name der aufzurufenden Hilfedatei (samt Pfadangabe). Da es hier nur um das Schließen einer offenen Hilfeanwendung geht, reicht die Angabe eines leeren "Dummies" (= dummy$). *lpzFileName*
Der dritte Funktionsparameter (wCmd) übermittelt ein Kommando an das Programm WINHELP. In unserem Falle lautet es, da die Hilfe geschlossen werden soll, HELP_QUIT. Nun wird allerdings der kritische Leser sofort bemerken, daß dieser dritte Parameter (wCmd nämlich) als Integer deklariert ist, wohingegen "HELP_QUIT" nun sicher ein String und kein Integer ist. Damit der Aufruf trotzdem funktioniert, muß der .BAS-Datei noch folgende Zeile hinzugefügt werden: *wCmd*
```
Global Const HELP_QUIT = &H2
```
Auf diese Weise wird (global für alle Module des Programms) ein Zahlenwert mit HELP_QUIT kombiniert, der es erlaubt, im folgenden HELP_QUIT an einer Stelle zu schreiben, an der ein Integer-Wert verlangt wird. Diese zuerst scheinbar als etwas "um die Ecke" daherkommende Verfahrensweise erlaubt es, statt (mit schlecht merkbaren) numerischen Werten mit (einprägsamen) Wort-Bezeichnungen zu operieren. Man findet eine Zusammenstellung der globalen Konstanten im Visual Basic-Verzeichnis in der Datei CONSTANT.TXT.
Der vierte Funktionsparameter (dwData nämlich) bezeichnet den Topic, der Ziel des "Hilferufs" ist. Da hier die Hilfe geschlossen wird, es also nicht um den Aufruf eines Topics geht, ist der Eintrag beliebig (und folgenlos). *dwData*

Ziel Nr. 2:
Aufruf der Hilfe-Datei WINHELP.HLP
Die Winhelp-Funktion eröffnet die Möglichkeit, außer der für das Projekt (unter Options, Project) definierten Hilfedatei noch weitere Hilfedateien aufzurufen. Eine solche weitere Hilfedatei, die man möglicherweise innerhalb der eigenen Applikation den Benutzern zur Verfügung stellen will, ist WINHELP.HLP, die Datei, die allgemein über die Verwendung der Windows-Hilfe informiert. Um diesen Aufruf zu demonstrieren, kreieren wir einen

neuen Menu-Punkt "Windows-Hilfe" und hinterlegen den folgenden Code:

```
t = winhelp(hwnd, "winhelp.hlp", help_index, 0)
```

Nach der vorstehenden Erläuterung der Funktionsparameter sind nur wenige Zusatzbemerkungen erforderlich. Damit an dritter Stelle "help_index" funktioniert und die Hauptseite aufruft, ist wieder der Weg über eine globale Konstante in der eben erläuterten Form notwendig. Die entsprechende Zeile für die .BAS-Datei lautet:

```
Global Const HELP_INDEX = &H3
```

Der vierte Parameter ist beliebig, wenn der dritte Parameter "help_index" war. Man kann also (als Integer) eintragen, was man will.

Visual Basic-"Connectivity"

Der kurze Blick auf die Winhelp-Funktion ist geeignet, eine der zentralen Stärken von Visual Basic ansatzweise zu veranschaulichen: Der Zugriff auf die Windows-Funktionen über das Windows-API (und auf andere Funktionen in DLL's, die dokumentiert sind) erlaubt eine "Vernetzung" der eigenen Applikation mit der gesamten Windows-Umgebung, die eine Integration (im echten Sinne) ermöglicht. Die Reichweite der eigenen Programmiermöglichkeiten wird damit außerordentlich gesteigert. Hinzuzufügen ist: Wenn die betreffenden Funktionen dokumentiert sind. Die von Schulman angestoßene Debatte um die nicht-dokumentierten Windows-Funktionen belegt, daß die Windows-Entwicklung nicht frei von Abschottungstendenzen ist. Es leuchtet ein, daß der prinzipiell gute integrative Ansatz durch derartige Strategien konterkariert wird. Womit zuletzt der programmierende Jurist wieder im Heimathafen des Wettbewerbs- und Kartellrechts angekommen wäre: Dort sind alle Funktionen deklariert (und müssen es auch sein), obwohl, wenn man genauer nachdenkt ...