Excel VBA

PEARSON Education

ADDISON-WESLEY

MASTER CLASS

Michael Schwimmer

Excel VBA

Einstieg für Anspruchsvolle

ADDISON-WESLEY [in Kooperation mit] PEARSON Studium

Bibliografische Information Der Deutschen Bibliothek

Die Deutsche Bibliothek verzeichnet diese Publikation in der Deutschen Nationalbibliografie; detaillierte bibliografische Daten sind im Internet über *http://dnb.ddb.de* abrufbar.

10 9 8 7 6 5 4 3 2 1

08 07 06 05

ISBN 3-8273-2183-2

ein Imprint der Pearson Education Deutschland GmbH,
Martin-Kollar-Straße 10–12, D-81829 München/Germany

www.pearson-studium.de
Lektorat: Frank Eller, feller@pearson.de
Einbandgestaltung: Marco Lindenbeck, webwo GmbH (mlindenbeck@webwo.de)
Herstellung: Elisabeth Prümm, epruemm@pearson.de
Satz: mediaService, Siegen (www.media-service.tv)
Druck und Verarbeitung: Kösel, Krugzell (www.KoeselBuch.de)

Printed in Germany

Inhaltsverzeichnis

Vorwort 11

Kapitel 1 Grundlagen 15

1.1 Was Sie in diesem Kapitel erwartet . . . 15
1.2 Kommentare . . . 15
1.3 Geschwindigkeit von VBA . . . 16
1.4 Variablen . . . 17
1.4.1 Variablennamen . . . 17
1.4.2 Namenskonventionen . . . 20
1.4.3 Datentypen . . . 23
1.4.4 Variablendeklaration . . . 26
1.4.5 DefType . . . 30
1.4.6 Gültigkeitsbereich . . . 32
1.4.7 Lebensdauer . . . 33
1.5 Konstanten . . . 34
1.5.1 Normale Konstanten . . . 34
1.5.2 Enum-Typen . . . 35
1.6 DoEvents . . . 36
1.7 Benutzerdefinierte Tabellenfunktionen . . . 38
1.8 Parameterübergabe ByVal versus ByRef . . . 41
1.9 Performance steigern . . . 42
1.9.1 Select Case versus If Then . . . 42
1.9.2 Logische Operatoren . . . 46
1.9.3 Zeitvorteil durch GoTo . . . 50
1.9.4 Referenzierung und Objektvariablen . . . 52
1.9.5 Bildschirmaktualisierungen . . . 54
1.9.6 Berechnungen ausschalten . . . 56
1.9.7 Weitere Optimierungsmöglichkeiten . . . 57
1.10 Rekursionen . . . 61
1.11 Sortieren und Mischen . . . 64
1.11.1 Bubblesort . . . 66
1.11.2 Quicksort . . . 67
1.11.3 Mischen . . . 70
1.12 Bedingte Kompilierung . . . 71

Kapitel 2 Auflistungen und Collections 73

2.1 Was Sie in diesem Kapitel erwartet . . . 73
2.2 Auflistungen . . . 73
2.2.1 Durchlaufen von Auflistungen . . . 74
2.2.2 Löschen aus Auflistungen . . . 76
2.3 Collections . . . 79
2.3.1 Vorteile von Collections . . . 79
2.3.2 Nachteile von Collections . . . 81
2.3.3 Collections anlegen . . . 81
2.3.4 Bingo mit Collection . . . 83
2.3.5 Bingo mit Datenpool . . . 85
2.3.6 Bingo mit Datenfeld . . . 86
2.3.7 Beispiel Änderungen markieren . . . 87
2.3.8 Fazit . . . 90

Kapitel 3 Klassen 91

3.1 Was Sie in diesem Kapitel erwartet . . . 91
3.2 Allgemeines . . . 91
3.3 Instanzierung . . . 93
3.4 Eigenschaften und Methoden . . . 93
3.5 Überladen von Parametern . . . 94
3.6 Erstellen von Klassen . . . 95
3.6.1 Initialisierungsdateien . . . 95
3.6.2 Das Benutzen der Klasse clsIni . . . 96
3.6.3 Die Klasse clsIni . . . 99
3.7 Ereignisprozeduren WithEvents . . . 103
3.7.1 Kontrollkästchen und Tabellenblatt . . . 103
3.7.2 CommandButton und UserForms . . . 106

Kapitel 4 Datenbanken 111

4.1 Was Sie in diesem Kapitel erwartet . . . 111
4.2 Excel ist keine Datenbank . . . 111
4.3 ADOX (ActiveX Data Objects Extension) . . . 113
4.4 ADO (ActiveX Data Objects) . . . 115
4.4.1 Das Connection-Objekt . . . 116
4.4.2 Das Recordset-Objekt . . . 118
4.5 SQL . . . 127
4.6 Excel-Tabellen . . . 128
4.7 Access-Datenbanken . . . 133
4.8 dBase . . . 136
4.9 Oracle-Datenbanken . . . 138

Kapitel 5 API-Grundlagen 141

5.1 Was Sie in diesem Kapitel erwartet . . . 141
5.2 Was ist überhaupt die API? . . . 142
5.3 Datenspeicher. . . . 142
5.4 Stack . . . 143
5.5 Parameterübergabe. . . . 145
5.6 Die Declare-Anweisung . . . 146
5.7 Datentypen. . . . 148
5.8 Strings . . . 155
5.8.1 Kommandozeile auslesen . . . 156
5.9 CopyMemory . . . 158
5.10 Little Endian, Big Endian . . . 160
5.11 Arrays, Puffer und Speicher . . . 162
5.11.1 Arrays allgemein . . . 162
5.11.2 Arrays im Speicher . . . 162
5.11.3 Puffer . . . 165
5.11.4 Safearrays. . . . 167
5.12 Fenster . . . 175
5.13 Koordinaten, Einheiten. . . . 177
5.14 Weiterführende Quellen . . . 179

Kapitel 6 Dialoge 181

6.1 Was Sie in diesem Kapitel erwartet . . . 181
6.2 Eingebaute Dialoge . . . 181
6.3 Dialoge zum Ändern der Systemeinstellungen . . . 183
6.4 Ausgabe von Meldungen . . . 183
6.4.1 MsgBox, MessageBoxA . . . 183
6.4.2 MsgBox Timeout . . . 186
6.5 Inputbox . . . 194
6.6 Farbauswahl. . . . 197
6.7 Schriftartdialog . . . 204
6.8 Dateiauswahl . . . 209
6.9 Verzeichnisauswahl . . . 213

Kapitel 7 Dateien und Verzeichnisse 219

7.1 Was Sie in diesem Kapitel erwartet . . . 219
7.2 Allgemeines . . . 220
7.2.1 VBA-Befehle und -Funktionen . . . 220
7.2.2 FileSearch-Objekt . . . 220
7.2.3 FileSystemObject . . . 220
7.2.4 API . . . 221
7.2.5 DOS-Befehle . . . 221

7.3 Dateien suchen 221
7.3.1 Dir 222
7.3.2 FileSearch-Objekt 225
7.3.3 Scripting.FileSystemObject 228
7.3.4 API 230
7.3.5 Fazit 238
7.4 Dateiattribute lesen und schreiben 239
7.4.1 GetAttr/SetAttr 239
7.4.2 FileSystemObject 240
7.4.3 API 241
7.5 Dateizeiten lesen und schreiben 244
7.5.1 FileDateTime 245
7.5.2 FileSystemObject 245
7.5.3 API 245
7.6 Komplette Pfade anlegen 254
7.7 Dateioperationen mit der API 255
7.7.1 Verschieben und Kopieren 258
7.7.2 Löschen 259
7.8 Lange und kurze Dateinamen 260
7.9 Sonderverzeichnisse 264
7.10 Dateien kürzen 266

Kapitel 8 Laufwerke 271

8.1 Was Sie in diesem Kapitel erwartet 271
8.2 Informationen über Laufwerke 271
8.2.1 FileSystemObject 272
8.2.2 API 273
8.3 Freies Netzlaufwerk ermitteln 280
8.4 Netzlaufwerke verbinden und trennen 280
8.4.1 Windows Script Host 281
8.4.2 WNetAddConnection2, WNetCancelConnection2 282

Kapitel 9 Datum und Zeit 285

9.1 Was Sie in diesem Kapitel erwartet 285
9.2 Ostern und Feiertage 285
9.2.1 Feiertage 287
9.2.2 Wochenende oder Feiertag 288
9.3 Weitere Zeitfunktionen 288
9.3.1 Kalenderwoche 288
9.3.2 Montag der Woche 289
9.3.3 Lebensalter 289
9.4 Datums- und Zeiteingabe 290
9.5 Sonnenauf- und Sonnenuntergang 292

Kapitel 10 Grafik 295

10.1 Was Sie in diesem Kapitel erwartet . . . 295
10.2 Bilderschau . . . 295
10.3 Icons extrahieren . . . 301
10.4 Bildschirmauflösung ändern . . . 312
10.4.1 Die UserForm . . . 313
10.4.2 Klasse clsWinEnd . . . 320
10.5 Spielkarten . . . 325
10.6 Statusleiste als Zeichenfläche . . . 333
10.6.1 Die Klasse clsStatusleiste . . . 334
10.6.2 Prozeduren zum Testen der Klasse . . . 347

Kapitel 11 Sound 351

11.1 Was Sie in diesem Kapitel erwartet . . . 351
11.2 Die Beep-Anweisung . . . 351
11.3 Die API-Funktion Beep . . . 352
11.4 Die API-Funktion sndPlaySound . . . 352
11.5 Die API-Funktion mciSendString . . . 353
11.6 Piepsen . . . 356
11.6.1 Benutzen der Klasse clsWave . . . 357
11.6.2 Die Klasse clsWave . . . 357
11.7 Töne mit MIDI . . . 370

Kapitel 12 UserForms 377

12.1 Was Sie in diesem Kapitel erwartet . . . 377
12.2 Min, Max, Resize . . . 377
12.3 Runde UserForms mit Loch . . . 384

Kapitel 13 Fremde Anwendungen 393

13.1 Was Sie in diesem Kapitel erwartet . . . 393
13.2 Allgemeines . . . 394
13.2.1 OLE . . . 394
13.2.2 Shell . . . 395
13.2.3 ShellExecute . . . 395
13.3 E-Mail mit VBA, Shell und ShellExecute versenden . . . 398
13.4 E-Mail mit Outlook versenden . . . 399
13.5 Anwendung starten und warten . . . 403
13.6 Fremde Programme abschießen . . . 405
13.7 Präsentation starten . . . 408
13.7.1 Shell . . . 408
13.7.2 ShellExecute . . . 411
13.7.3 OLE . . . 412
13.8 Wahlhilfe benutzen . . . 416

Kapitel 14 Netzwerke und Internet 419

14.1 Was Sie in diesem Kapitel erwartet 419
14.2 Netzwerkressourcen 419
14.2.1 Anstoßen der Suche 420
14.2.2 Die Suche nach Ressourcen 421
14.3 Tracert 428
14.3.1 Benutzen der Klasse clsTracert 429
14.3.2 Die Klasse clsTracert 431
14.4 FTP und HTTP 444
14.4.1 Die Klasse clsInternet 445
14.4.2 FTP-Übertragung 461
14.4.3 URL lesen 463
14.5 Net Send 464

Kapitel 15 Sonstiges 465

15.1 Was Sie in diesem Kapitel erwartet 465
15.2 OEM/Char 465
15.3 Drucker auflisten und auswählen 468
15.4 Beschreibung für eigene Funktionen 477
15.5 Freischaltcode erzeugen 480
15.6 GUID 484

Stichwortverzeichnis 487

Vorwort

Excel

Unter allen Anwendungsprogrammen ist Excel meiner Ansicht nach mit Abstand das flexibelste und als Tabellenkalkulation ist es nahezu unschlagbar. Ich kenne persönlich sogar Leute, die Excel als Textverarbeitung oder als Zeichenprogramm missbrauchen und damit Schalt- und Klemmenbelegungspläne zeichnen.

Neben solch eher zweifelhaften Einsatzgebieten bietet Excel als Kalkulationsprogramm in der Finanz- und Bürowelt so ziemlich alles, was man sich in diesem Bereich vorstellen kann. Zum Erledigen dieser Aufgaben stehen dem Benutzer jede Menge eingebaute Funktionen zur Verfügung. Man kann diese Funktionen zu Formeln zusammenfassen und ist damit in der Lage, auch kompliziertere Berechnungen in einer einzigen Zelle durchzuführen. Sogar Matrixformeln lassen sich erstellen, wodurch sich die Anzahl der Möglichkeiten noch mal erhöht.

Als besondere Highlights gelten zu Recht der Solver und die Pivottabellen. Sie sind Hilfsmittel, die viele zusätzliche Aufgabengebiete abdecken. Hochachtung vor dem, der von sich behaupten kann, mit all dem, was Excel standardmäßig so bietet, sicher umgehen zu können. Ich selbst gehöre leider nicht zu diesem Personenkreis.

Ich tröste mich aber immer damit, dass es sicherlich kaum jemanden gibt, der sich hundertprozentig mit allen Funktionen, Einstellungen und Möglichkeiten auskennt, die Excel bietet, selbst die Entwickler nicht. Für manche Tabellenfunktionen ist sogar ein Mathematikstudium von Vorteil.

VBA

Trotz aller eingebauten Funktionalitäten stößt man ab und zu auf Beschränkungen, die sich auch mit den raffiniertesten Matrixformeln nicht durchbrechen lassen. Spätestens dann kommt VBA zum Einsatz. Sie können sich eigene Funktionen schreiben und diese genau so in Formeln einsetzen, wie jede andere eingebaute Funktion auch. Als ein besonderer Leckerbissen gelten die Ereignisprozeduren. Angefangen beim Öffnen einer Mappe gibt es massig Ereignisse, die eine größtmögliche Automatisierung von Excel gestatten. Durch das Einsetzen von UserForms ist zudem ein komfortabler Dialog mit dem Benutzer möglich.

Neben dem Konzept von .NET erscheint VBA vielleicht etwas antiquiert, aber im Officebereich ist es nahezu unverzichtbar und ich sehe auch für die nähere Zukunft keine Alternative dazu. Dazu müsste Microsoft schon revolutionär neue Konzepte anbieten, die zudem noch abwärtskompatibel sein müssten.

Sicherlich verfügen die neueren Versionen der Microsoft-Bürosoftware über Funktionalitäten, die vor ein paar Jahren unvorstellbar waren. Aber mit dem Essen steigt bekanntlich auch der Appetit. Der Trend geht dabei immer mehr zur Automatisierung. Aufgaben, die früher mithilfe der Assistenten erledigt wurden, sollen heute weitgehend ohne Eingriff eines Benutzers ablaufen und genau dabei wird VBA eingesetzt.

Noch so ein Excel-VBA-Buch?

Es geht in diesem Buch zwar um VBA und Excel, aber ganz gewiss nicht darum, VBA oder Excel von Grund auf zu lernen. Dass Sie Excel gut kennen sowie VBA relativ sicher beherrschen, gilt als Grundvoraussetzung, um mit dem hier vorliegenden Werk überhaupt etwas anfangen zu können.

Bücher, die einem den Umgang mit Excel beibringen, oder den Leser in die Geheimnisse der VBA Programmierung einweihen wollen, gibt es bereits massig. Darunter sind auch einige hervorragende Werke, wie beispielsweise die Werke von Bernd Held und Michael Kofler über VB(A) und Excel, die zum Erlernen der Programmiersprache ein absolutes Muss sind. Wer VBA-Lösungen für alle möglichen Probleme des Alltags sucht, für den gibt es sicher nichts besseres als das Excel-VBA Codebook von Monika Weber und Melanie Breden.

Dieses Buch geht etwas weiter und richtet sich an den ambitionierten VBA-Aufsteiger, der schon die Grundlagen beherrscht und auch sonst nicht immer das Gleiche vorgesetzt bekommen will.

Dabei soll die Lernkurve zwar steil sein, darf aber auch nicht überfordern. Sicherlich ist das eine Gratwanderung, deshalb wird in diesem Buch auch besonderer Wert auf kommentierten Beispielcode gelegt, denn es ist auch für mich immer wieder frustrierend, Funktionalitäten erklärt zu bekommen, ohne auf ein funktionierendes Beispiel zurückgreifen zu können. Ein Beispiel mit Quelltext erklärt manchmal mehr als tausend Worte.

- In Kapitel 1 werden einige Grundkonzepte von VBA vorgestellt, die Ihnen helfen sollen, Ihren Code übersichtlicher, wartbarer und in vielen Fällen auch schneller zu machen. Außerdem werden anhand von Beispielen Rekursionen und einige Algorithmen zum Sortieren und Mischen vorgestellt.
- In Kapitel 2 werden die Vor- und Nachteile von Auflistungen vorgestellt und es wird gezeigt, wie Sie diese selbst in Form von Collections erzeugen können. Ein Thema dabei sind auch die zeitlichen Aspekte, die darüber mitentscheiden, ob und wann der Einsatz einer Collection überhaupt sinnvoll ist.
- Kapitel 3 beschäftigt sich mit Klassen. Anhand eines Beispieles, mit dem Sie Zugriff auf Initialisierungsdateien bekommen, werden die Schritte zum Anlegen und Benutzen einer einfachen Klasse vorgestellt. In zwei weiteren Beispielen sehen Sie, wie Sie Steuerelemente dynamisch einfügen und mithilfe von Klassen mit den Ereignissen dieser Steuerelemente umgehen.

- In Kapitel 4 lernen Sie einiges über den Zugriff auf Datenbanken mithilfe von ADO. Außerdem wird gezeigt, wie Sie programmgesteuert eine einfache Access-Datenbank anlegen können, auch ohne im Besitz dieses Datenbankprogramms zu sein.
- Kapitel 5 ist eine kleine Einführung in die Benutzung der Windows-API. Es werden die Fallstricke angesprochen, die auf Sie lauern und es wird beschrieben, wie Sie diese umgehen können.
- Kapitel 6 beschäftigt sich mit Dialogen. Dabei wird kurz auf die eingebauten Dialoge eingegangen und es werden die Windows-internen Dialoge zur Verzeichnis-, Farb- und Schriftartauswahl vorgestellt. Außerdem wird gezeigt, wie Sie die Dialoge zum Ändern der Systemeinstellungen aufrufen.
- Kapitel 7 beschäftigt sich ausgiebig mit Dateien und Verzeichnissen. Es wird unter anderem gezeigt, wie Sie Dateien suchen, löschen, die Dateiattribute auslesen und ändern können, wie Sie ganze Verzeichnisse mit den darin enthaltenen Dateien und Unterverzeichnissen kopieren, verschieben sowie Dateien und Verzeichnisse in den Papierkorb schieben können.
- In Kapitel 8 wird gezeigt, wie Informationen über die vorhandenen Laufwerke ausgelesen werden können. Außerdem wird gezeigt, wie Sie Netzlaufwerke programmgesteuert verbinden und trennen.
- In Kapitel 9 werden einige interessante Funktionen und Prozeduren vorgestellt, die sich mit dem Datum und der Zeit beschäftigen.
- Kapitel 10 beschäftigt sich mit Grafiken aller Art. Neben einer Bilderschau, dem Rippen von Icons und dem Anzeigen und Exportieren der Windows-internen Spielkarten ist ein weiteres Thema das Erzeugen einer Fortschrittsanzeige in der Statusleiste.
- Kapitel 11 zeigt die Möglichkeiten, die es gibt, Töne zu erzeugen und Klänge und Musikstücke abzuspielen. Dabei können beliebige Töne in den unterschiedlichsten Kurvenformen selbst erzeugt werden oder die Midifunktionen benutzt werden, welche die Töne der Tonleiter in allen Oktaven für die unterschiedlichsten Instrumente bereitstellen. Außerdem wird gezeigt, wie Sie Video-, Audiodateien und auch ganze Audio-CDs abspielen können.
- In Kapitel 12 werden UserForms manipuliert. UserForms mit Löchern und solche, die sich wie andere Fenster minimieren, maximieren und durch Ziehen am Rahmen in der Größe ändern lassen werden vorgestellt.
- Kapitel 13 enthält ein paar Beispiele, die zeigen, wie Sie mit OLE fremde, automatisierungsfähige Programme fernsteuern können. Ein anderes Beispiel beschäftigt sich damit, andere Anwendungen mit der Shell-Funktion zu starten, und erst nach deren Ende mit der weiteren Programmausführung fortzufahren.
- In Kapitel 14 über Netzwerke und das Internet lernen Sie, wie Netzwerkressourcen aufgelistet werden können. Eine vorgestellte Klasse erlaubt es, ohne fremde Hilfsmittel Dateien von und zu einem FTP zu übertragen, dort Dateien oder Verzeichnisse zu löschen, Verzeichnisse anzulegen, die Dateistruktur aufzulisten und Internetseiten auszulesen.

- Kapitel 15 beschäftigt sich mit sonstigen Themen. Eines ist die programmgesteuerte Änderung des aktuellen Druckers, ein weiteres befasst sich mit Beschreibungen benutzerdefinierter Funktionen, die auch im Funktionsassistent angezeigt werden. Des Weiteren wird das Erzeugen eines Hashcodes zum Freischalten beschrieben und wie Sie Text vom ASCII- in den ANSI-Zeichensatz transferieren. Darüber hinaus wird das Erzeugen einer GUID gezeigt.

Der Quellcode, den Sie in diesem Buch finden, steht auch auf der beiliegenden CD zur Verfügung. Jedes Beispiel enthält einen Hinweis auf die Position der Arbeitsmappe mitsamt Modul, in welcher sich der Code befindet.

Achtung

Da es sehr viele unterschiedliche Hard- und Softwarekombinationen gibt, die Probleme bereiten könnten, kann ich leider nicht garantieren, das alle vorgestellten Programme oder Programmteile immer und überall auf Anhieb funktionieren. Aus den gleichen Grund kann ich auch keine Haftung für irgendwelche Folgen übernehmen, die sich aus dem Benutzen des hier und auf der CD veröffentlichten Codes ergeben. Trotzdem bin ich mir aber ziemlich sicher, dass die vorgestellten Programme auch bei Ihnen ohne große Änderungen funktionieren werden.

1

Grundlagen

1.1 Was Sie in diesem Kapitel erwartet

In diesem Kapitel geht es nicht darum, VBA von Grund auf zu lernen. Die Zielgruppe für dieses Buch ist eine andere und vom Leser werden Kenntnisse in VBA vorausgesetzt.

Es geht vielmehr darum, Sie anzuleiten, Ihren Programmierstil zu verbessern und somit die Lesbarkeit und die Ausführungsgeschwindigkeit der von Ihnen geschriebenen Programme zu erhöhen. Dabei wird auf potenzielle Fehlerquellen eingegangen, die sich aus dem Umgang mit Variablen, Prozeduren und Funktionen ergeben. Wenn es dem Verständnis dient, wird aber auch schon Bekanntes wiederholt, aber dann nicht so vertiefend, dass es für einen Anfänger zum Selbststudium der Programmiersprache VBA geeignet wäre.

1.2 Kommentare

Wenn Sie Programme schreiben – und zu den Programmen zählen schließlich auch VBA- Prozeduren und Funktionen – werden Sie sich im Allgemeinen darauf konzentrieren, die Funktionalität fehlerfrei hinzubekommen. Das ist auch gut so, denn nur deshalb schreibt man ja schließlich Programme.

Sparen Sie nicht mit Kommentaren, wenn es dem Verständnis des Programmablaufs dient, aber verfallen Sie auch nicht ins Gegenteil und kommentieren jede Zeile. Das macht den Code unübersichtlich und es macht sich niemand mehr die Mühe, Ihre Kommentare überhaupt zu lesen!

Dabei wird leider immer wieder vergessen, dass die nächste Änderung des Programms schneller kommt, als man im Allgemeinen denkt. Sei es, um einen Bug zu beseitigen, die Funktionalität zu erweitern oder einfach nur, um die Verarbeitungsgeschwindigkeit zu erhöhen. Mir geht es meistens so, dass ich dann erst eine längere Zeit brauche, um mich sogar in meinem eigenen Code wieder zurechtzufinden.

Dabei geht es nicht darum, dass ich plötzlich nicht mehr erkenne, was eine Schleife oder ein Funktionsaufruf ist. Der Knackpunkt ist, nachzuvollziehen, warum möglicherweise abweichend von der normalen Vorgehensweise gerade dieser Weg gegangen wurde und welche Fallstricke damit umgangen werden.

Ich muss gestehen, dass ich in meiner Anfangszeit schon einmal Codeteile einer Funktion gelöscht habe, die scheinbar unnütz waren. Wochen später, nachdem die Funktion immer mal wieder ein falsches Ergebnis geliefert hatte, kam mir plötzlich die Erleuchtung. Als ich den Code entwickelt hatte, habe ich in einem wahren Anflug von Genialität diesen Fehler kommen gesehen und ihn mit ein paar Zeilen Code umgangen. Nur leider wurde von mir nichts dokumentiert.

Wenn man aber nicht gerade Beispielcode für ein Buch schreibt, braucht man auch nicht jede Schleife zu kommentieren. Wichtiger ist dabei, zu erklären, warum gerade jetzt, an dieser Stelle, die Prozedur XYZ aufgerufen wird. Oder einfach nur, welche nicht auf den ersten Blick sichtbare Klippen mit den nächsten paar Zeilen umschifft werden. Ein kleiner zusammenfassender Kommentar, was eine selbst geschriebene Funktion oder Prozedur überhaupt macht, ist sicherlich sehr hilfreich, um den Gesamtablauf besser zu verstehen.

Nicht ganz so wichtig ist dagegen in den meisten Fällen, wer, wann, wo und warum das vorliegende Programm oder die Routine geschrieben wurde. Man sieht häufig im Netz die reinsten Orgien von Quelltextkommentaren, die nicht nur den Autor und den Erstellungszeitpunkt beinhalten, was ja schließlich auch nicht ganz unwichtig ist, sondern auch noch, was er vorher gefrühstückt hat.

1.3 Geschwindigkeit von VBA

Ein Programm, das in VBA geschrieben ist, liegt in einer für den Prozessor nicht direkt ausführbaren Form vor. Das bedeutet, dass der Quellcode, der als Text in einem Modul steht, erst in eine Form umgewandelt werden muss, die der Prozessor auch versteht. Bei vielen Programmiersprachen übernimmt ein Compilerprogramm diese Aufgabe und erzeugt ausführbare Dateien in Maschinensprache. Dieses Übersetzen nennt man »kompilieren«.

Unter VBA wird der Quellcode erst während der Laufzeit von einem Interpreter übersetzt und steht somit nicht als eigenständiges Maschinenprogramm in einer Datei zur Verfügung. Die Übersetzung bei Bedarf ist natürlich langsamer, als ein Programm direkt aus einer in Maschinencode vorliegenden Datei auszuführen.

Man merkt den Unterschied, wenn man die Geschwindigkeit von selbst geschriebenen Tabellenfunktionen mit den eingebauten Funktionen von Excel vergleicht. Die eingebauten Tabellenfunktionen liegen in kompilierter Form vor und laufen extrem schnell ab. Benutzerdefinierte Funktionen sind erheblich langsamer, deshalb sollten Sie es vermeiden, diese allzu häufig im Tabellenblatt einzusetzen.

Viele Probleme lassen sich auch durch den Einsatz von Matrixfunktionen lösen, sogar so etwas wie Schleifen sind damit möglich. Fragen Sie am besten einmal in den einschlägigen Newsgroups nach oder bemühen eine Suchmaschine Ihrer Wahl mit den Schlüsselwörtern `Excel` und `Matrixfunktion` oder `Summenprodukt`.

Manche Sachen lassen sich aber nur durch den Einsatz von VBA vernünftig lösen. Wenn Sie dabei ein paar grundlegende Dinge beachten, können Sie auch mit VBA recht schnellen Code schreiben.

Ein Interpreter, wie er von VBA benutzt wird, ist zwar langsam, bietet aber auch einige Vorteile gegenüber einem Compiler. Da der Quellcode erst kurz vor der Ausführung übersetzt wird, ist es möglich, ein VBA-Programm im Einzelschrittmodus abzuarbeiten, den Programmablauf an einer beliebigen Stelle anzuhalten und sogar den Programmcode während der Laufzeit zu verändern. Das Debugging, also die Fehlerbeseitigung, wird dadurch erheblich vereinfacht.

1.4 Variablen

Zu den Grundpfeilern jeder Programmiersprache gehören die Variablen. Diese sind dafür da, alle möglichen Daten wie zum Beispiel Werte, Texteingaben oder Zwischenergebnisse aufzunehmen und für eine gewisse Zeit im Arbeitsspeicher am Leben zu erhalten, bis sie im weiteren Programmverlauf verarbeitet oder ausgegeben werden können.

1.4.1 Variablennamen

Folgende grundsätzliche Regeln zur Benennung von Variablen werden von VBA vorgegeben:

1. Der Name muss mit einem Buchstaben beginnen.
2. Der Name darf keine Leerzeichen, Punkte oder Zeichen enthalten, die von VBA für Vergleiche oder Berechnungen benutzt werden, wie zum Beispiel `+ - = * /` usw.
3. Die Länge des Namens darf nicht größer als 255 Zeichen sein.
4. Die Namen dürfen nicht mit gesperrten Schlüsselwörtern identisch sein, wie zum Beispiel `Name`, `Text` oder `Color`.
5. Variablennamen müssen in ihrem Gültigkeitsbereich eindeutig sein. VBA unterscheidet dabei nicht zwischen Groß- und Kleinschreibung.

Innerhalb dieser relativ weiten Grenzen dürfen Sie sich frei bewegen, was den Namen Ihrer Variablen betrifft. Sie sollten sich trotzdem an ein paar weitere Regeln zur Namensvergabe halten. Das dient nicht nur dazu, den Code einheitlich zu gestalten, es soll Ihnen vielmehr die Arbeit erleichtern und zudem ermöglichen, dass Sie sich auch nach längerer Zeit noch in Ihren Programmen auskennen.

Verwenden Sie für Ihre Variablen sprechende, also selbsterklärende Namen, sodass man auf Anhieb erkennt, welche Daten diese Variable aufnehmen soll. Es ist zwar gewöhnungsbedürftig und verlangt anfangs etwas Disziplin, aber wenn Sie sich einmal daran gewöhnt haben, bringt es Ihnen nur noch Vorteile. Ihr Code wird wartbarer, weil Sie sich auch nach längerer Zeit ohne größere Codeanalyse in Ihrem Quelltext zurechtfinden.

In diesem Buch benutze ich fast durchgängig englische Variablennamen, das müssen Sie aber nicht unbedingt in Ihrem Code nachmachen. Verwenden Sie ruhig Namen in der Sprache, mit der Sie und Ihre Kollegen am besten zurechtkommen.

Für Zählvariablen bietet sich eine Einbuchstabenlösung an. Allgemein benutzt man hier die Buchstaben i wie Index, `k`, `m`, `x`, `y`. Bewegen Sie sich durch ein Tabellenblatt, können Sie für die Spalten- und Zeilenposition `x` bzw. `y` benutzen, ich empfehle aber trotzdem, den Namen `Zeile` oder `Spalte` zu gebrauchen. Sie sollten dabei aber Präfixe für den Datentyp und die Gültigkeit benutzen, wie es in der Passage Namenskonventionen näher erläutert wird (siehe Seite 20). Man erkennt dabei auf Anhieb, um was es geht. Die Variable für die Zeile müssen und die für die Spalte sollten Sie als `Long` deklarieren, um Überläufe zu vermeiden.

Achtung

Benutzen Sie keine Variablennamen, die als Objektnamen oder als Eigenschaften und Methoden davon vorkommen. Wenn Sie diese Variablen in Klassenmodulen von Objekten benutzen, die eine Eigenschaft oder Methode mit dem gleichen Namen besitzen, wird statt der Variablen die Eigenschaft oder Methode des Objektes verwendet.

Es gibt keine Fehlermeldung, wenn Sie für einen Variablennamen, der in einem Modul als öffentlich, also `Public`, deklariert wurde, den Eigenschaftsnamen eines Objektes verwenden. Sie können mit dieser Variablen auch ohne Probleme arbeiten. Wollen Sie diese Variable aber in dem Klassenmodul eines Objektes verarbeiten, das solch eine Eigenschaft besitzt, sprechen Sie nicht die Variable, sondern die Eigenschaft des Objekts an.

Das kommt daher, dass das übergeordnete Objekt zum Benutzen einer Eigenschaft oder Methode nicht explizit angegeben werden muss. Sie können also im Klassenmodul eines Tabellenblatts beispielsweise statt `Me.Name` auch einfach `Name` schreiben und das `Me` als Bezeichner für das übergeordnete Objekt weglassen. VBA benutzt dann bei Namensgleichheit die Eigenschaft des Objektes.

Nachfolgend ein Beispiel (Listing 1.1) zur Demonstration der Probleme bei der Namensvergabe von Variablen.

Listing 1.1
Beispiele\01_Grundlagen_VBA\01_01_Variablen.xls\mdlNamenskonflikt

```
Public Name As String

Public Sub ShowAndRename()

   'Inhalt der Variablen "Name" ausgeben
   MsgBox "Inhalt der Variablen ""Name"" = " _
      & """" & Name & """", , "Namenskonflikt"

   ' Inhalt der Variablen löschen
   Name = ""
      ' Umbenennen einer Datei
   CreateAndKill
   On Error Resume Next
   With ActiveWorkbook
      Name (.Path & "\MasterClass.txt") As
           (.Path & "\MasterClass.csv")
   End With
End Sub
```

Listing 1.1 (Forts.)
Beispiele\01_Grundlagen_VBA\
01_01_Variablen.xls\
mdlNamenskonflikt

```
Private Sub CreateAndKill()
   On Error Resume Next
   With ActiveWorkbook
      ' Wenn die Quelldatei nicht existiert, wird sie angelegt
      Open .Path & "\MasterClass.txt" For Binary As 1
      Close

      ' Eventuell vorhandene Zieldatei löschen
      Kill .Path & "\MasterClass.csv"
   End With

End Sub

' Den folgenden Code bitte in das Klassenmodul eines Tabellenblattes
' einfügen, welches einen Button mit dem Namen cmdName besitzt:

Private Sub cmdName_Click()
   Dim strName As String

   strName = InputBox("Bitte einen Namen eingeben", _
      "Namenskonflikt Variable - Eigenschaft")

   If strName <> "" Then

      ' Wenn in der Inputbox etwas eingegeben wurde, versuchen,
      ' der öffentlichen Variablen "Name" diesen Wert zuzuweisen
      Name = strName

      ' Den Variableninhalt von "Name" ausgeben und die
      ' Name-Anweisung benutzen
      ShowAndRename

   End If
End Sub
```

Nach einem Klick auf den Button `cmdName` können Sie einen Namen in einer Inputbox eingeben. Obwohl der zurückgelieferte Text augenscheinlich der öffentlichen Variablen `Name` zugewiesen wird, ändern Sie in Wirklichkeit die Eigenschaft `Name` des Tabellenblattes.

In der Prozedur `ShowAndRename` erkennen Sie, dass es sogar noch eine VBA-Anweisung mit dem Namen `Name` gibt, mit der Sie Dateien umbenennen können. Das sorgt noch für zusätzliche Verwirrung.

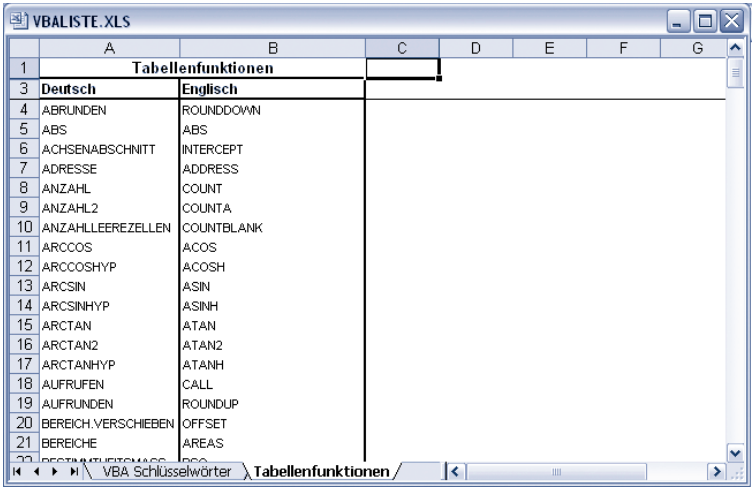
Die Prozedur `CreateAndKill` ist lediglich dafür zuständig, dass eine Datei zum Umbenennen existiert und die Zieldatei vor dem Umbenennen nicht vorhanden ist.

Wenn Sie ohne Präfixe arbeiten – das sind Vorsilben, die bei Variablennamen den Typ und die Gültigkeit kennzeichnen sollen –, dann sind Sie gut beraten, Namen zu meiden, die als VBA-Schlüsselwörter, Objektnamen oder als Eigenschaften und Methoden davon vorkommen. Ein gutes Hilfsmittel, solche verbotenen Namen zu erkennen, ist die Onlinehilfe. Setzen Sie den Cursor einfach in den Variablennamen und drücken Sie F1. Existiert eine solche Methode, Eigenschaft oder Schlüsselwort, bekommen Sie dafür Hilfe angeboten.

Hilfreich ist auch die Excelmappe *VBALISTE.XLS* (Abbildung 1.1), die beim Installieren von Office auf Ihrer Festplatte gelandet sein dürfte. Suchen Sie einmal im Officeordner nach dieser Datei; in den meisten Fällen ist sie unter *C:\Programme\Microsoft Office\OFFICE11\1031* zu finden. Eventuell müssen Sie aber auf Ihrer Office-CD nachschauen und diese auf Ihre Platte kopieren.

In dieser Excelmappe finden Sie die VBA- und Excel-Schlüsselwörter mitsamt ihren deutschen Übersetzungen, die in Versionen vor Excel 97 benutzt wurden. Als besonderes Bonbon findet man dort die Tabellenfunktionen in englischer und deutscher Sprache in einer Tabelle gegenübergestellt. Das ist sehr hilfreich, wenn Sie im Programmcode Tabellenfunktionen benutzen wollen, dort werden die englischen Namen benötigt.

Abbildung 1.1
Die Datei VBALISTE.XLS



1.4.2 Namenskonventionen

Zu einem guten Programmierstil gehört die Verwendung einer Notation, mit der man auf Anhieb den Datentyp und die Gültigkeit von Variablen erkennen kann. Dazu werden Präfixe, also Vorsilben, benutzt.

Es hört sich vielleicht etwas pedantisch an, Präfixe für den Datentyp und die Gültigkeit zu verwenden und es bereitet anfangs sicherlich etwas Mühe, die Vorteile liegen aber auf der Hand. Sie erkennen so an jeder Stelle des Codes sofort den Datentyp, ohne sich zuvor die Deklarationsanweisung angesehen zu haben.

Die Gefahr, Schlüsselwörter, Methoden und Eigenschaften als Namen für Variablen zu benutzen sinkt dadurch auf ein Minimum und Sie vermeiden Konflikte mit doppelt vergebenen Variablennamen.

Dass Variablennamen überhaupt doppelt vergeben werden können, liegt im unterschiedlichen Gültigkeitsbereich verborgen. Sie können zwar nicht zwei gleiche Namen für Variablen vergeben, die den gleichen Gültigkeitsbereich haben. Es ist aber ohne weiteres möglich, den Namen einer modulweit gültigen Variablen in einer Prozedur neu zu vergeben.

Beim Einsatz von Präfixen kann so etwas nicht passieren, damit unterscheidet sich grundsätzlich der Name auf Modulebene von dem, der auf Prozedurebene deklariert wurde.

Hier ein Vorschlag, welche Präfixe Sie verwenden sollten. Diese Liste ist zwar kein absolutes Muss, die Namenskonventionen haben sich aber international eingebürgert, einem Amerikaner beispielsweise sollten diese Vorsilben ebenso geläufig sein wie Ihnen. Einige in anderen Quellen aufgeführte Objekte wurden in dieser Tabelle ganz weggelassen, da diese fast ausschließlich den Einsatz unter Visual Basic finden. Bei Bedarf können Sie im Internet nachschauen, die erste Anlaufstelle dürfte dabei *MSDN* von Microsoft sein.

Das Präfix für die Gültigkeit (Tabelle 1.1) kommt dabei an die erste Position.

Tabelle 1.1
Präfixe der Gültigkeit

Gültigkeit	Präfix
Global	g
Modulweit	m
Prozedurweit	Kein Präfix

Anschließend folgt die Information, ob es sich bei der Variablen um ein Array handelt. Ist das der Fall, benutzen Sie den Buchstaben a.

Danach wird die Kennung des Datentyps (Tabelle 1.2) hinzugefügt.

Tabelle 1.2
Präfixe der Datentypen

Variablentyp	Präfix
Boolean	bln
Byte	byt
Collection	col
Currency	cur
Date	dtm
Double	dbl
Error	err
Integer	int
Long	lng
Object	obj
Single	sng
String	str
Benutzerdefinierter Typ	udt
Variant	vnt

Objektvariablen und Steuerelemente sollte man mit Präfixen versehen, welche die Art des Objekts (Tabelle 1.3) beschreiben.

Tabelle 1.3
Präfixe der Objekte und Steuerelemente

Objekttyp	Präfix
Object	obj
Klasse	cls
Modul	mdl
ADO Data	ado
Befehlsschaltfläche (CommandButton)	cmd
Bezeichnungsfeld (Label)	lbl
Bild (Picture)	pic
Bildlaufleiste Horizontal (HScrollBar)	hsb
Bildlaufleiste Vertikal (VScrollBar)	vsb
Diagramm (Graph)	gra
Drehfeld (SpinButton)	spn
Figur (Shape)	shp
Formular	frm
Kombinationsfeld, Dropdown-Listenfeld (ComboBox)	cbo
Kontrollkästchen (CheckBox)	chk
Listenfeld (ListBox)	lsb
Optionsfeld	opt
Rahmen (Frame)	fra
Range (Excel)	rng
Register (TabStrip)	tab
Steuerelement Allgemein (Control)	ctr
Symbolleiste	tlb
Textfeld (TextBox)	txt

Die Präfixe werden generell kleingeschrieben. Den ersten Buchstaben danach sollten Sie unbedingt großschreiben, genauso wie bei zusammengesetzten Namen der erste Buchstaben des zweiten Wortes mit einem Großbuchstaben beginnen sollte. Ein mappenweit (`g`) verfügbares Array (`a`) vom Datentyp Long (`lng`), das verschiedene Zeilennummern aufnehmen soll, bekäme nach dieser Notation den Namen `galngVerschiedeneZeilennummern`.

Unterstriche sind zwar erlaubt und es wird auch verschiedentlich propagiert, diese zur besseren Lesbarkeit als Ersatz für nicht erlaubte Leerzeichen zu benutzen. Ich halte die Benutzung dennoch für keine gute Idee. Unterstriche sollten den Ereignisprozeduren vorbehalten bleiben, der Prozedurname muss sich bei diesen aus dem Objektnamen, einem Unterstrich und dem Namen des Ereignisses zusammensetzen.

Verwenden Sie unbedingt Groß- und Kleinschreibung bei den Variablennamen. Bei der anschließenden Codeeingabe schreiben Sie dann die Variablen grundsätzlich klein. Wenn die Zeile abgeschlossen ist, werden die kleingeschriebenen Namen in die deklarierte Form umgewandelt. Wenn sich nichts ändert, erkennt man schon zu diesem Zeitpunkt einen fehlerhaft geschriebenen Variablennamen.

1.4.3 Datentypen

Für jeden Wert, den man in einer Variablen speichern will, gibt es den einen optimalen Datentyp. Es sollte im Allgemeinen der Typ benutzt werden, der die Daten mit dem geringsten Speicherbedarf in der gewünschten Genauigkeit aufnehmen kann, was ganz besonders bei größeren mehrdimensionalen Datenfeldern wichtig ist.

Nachfolgende Tabellenübersicht (Tabelle 1.4) zeigt den Speicherbedarf und Wertebereich der wichtigsten Typen:

Tabelle 1.4
Speicherbedarf und Wertebereich verschiedener Datentypen

Datentyp	Bedarf	Wertebereich
Byte Ganzzahl	1 Byte	0 bis 255
Boolean Wahrheitswert	2 Bytes	`True` oder `False`
Integer Ganzzahl	2 Bytes	–32.768 bis 32.767
Long Ganzzahl	4 Bytes	–2.147.483.648 bis 2.147.483.647
Single Gleitkommazahl einfacher Genauigkeit	4 Bytes	–3,402823 E38 bis –1,401298 E–45 0 1,401298 E–45 bis 3,402823 E38
Double Gleitkommazahl doppelter Genauigkeit	8 Bytes	1,79769313486231 E308 bis –4,94065645841247 E–324 0 4,94065645841247 E–324 bis 1,79769313486232 E308
Currency skalierte Ganzzahl	8 Bytes	–922.337.203.685.477,5808 bis 922.337.203.685.477,5807
Decimal skalierte Ganzzahl	14 Bytes	+/–79.228.162.514.264.337.593.543.950.335 ohne Dezimalzeichen +/–7,9228162514264337593543950335 mit 28 Nachkommastellen
Date Datum	8 Bytes	1.1.100 bis 31.12.9999
String variable Länge	10 Bytes plus Länge	bis ca. 2 Milliarden Zeichen

Tabelle 1.4 (Forts)
Speicherbedarf und Wertebereich verschiedener Datentypen

Datentyp	Bedarf	Wertebereich
String feste Länge	Länge	bis ca. 65.400 Zeichen
Variant (Zahlen)	16 Bytes	1,79769313486231 E308 bis –4,94065645841247 E–324 0 4,94065645841247 E–324 bis 1,79769313486232 E308
Variant (Text)	22 Bytes plus Länge	bis ca. 2 Milliarden Zeichen
Object	4 Bytes	

Wenn man es nicht gerade mit einem größeren Datenfeld zu tun hat, kann man in bestimmten Fällen von der Regel zum Minimieren des Speicherbedarfs Abstand nehmen. Es ist sicherlich nicht verkehrt, anstatt Variablen vom Typ `Integer` generell den Datentyp `Long` zu verwenden. Die 32 Bit dieses Typs werden von der Prozessorarchitektur am besten verarbeitet und der Mehrbedarf an Speicher hält sich bei einzelnen Variablen in erträglichen Grenzen. Als positiver Nebeneffekt bewahrt Sie das in vielen Fällen vor unvorhergesehenen Überläufen und die Ausführungsgeschwindigkeit sollte sich theoretisch noch etwas erhöhen.

Die angesprochenen Überläufe dagegen haben nach einem von Murphys Gesetzen (Jeder Fehler wird dort sitzen, wo er am spätesten entdeckt wird.) die Angewohnheit, immer erst dann aufzutreten, wenn die Anwendung schon die Feuertaufe hinter sich hat und möglicherweise schon auf vielen verschiedenen Rechnern läuft.

Selbst wenn Sie sich im Vorfeld Gedanken darüber gemacht haben, welche Werte einer Variablen zugewiesen werden und aufgrund dieser Überlegungen den Datentyp sehr sorgfältig ausgewählt haben, kann es bei Berechnungen unter bestimmten Umständen zu Laufzeitfehlern kommen.

In diesem Fall steht man vor dem Problem, einen Fehler zu finden, der sich außerordentlich gut tarnt und selbst manch sattelfeste VBA'ler zur Verzweiflung bringen kann. Probieren Sie folgendes Beispiel (Listing 1.2) aus:

Listing 1.2
Beispiele\01_Grundlagen_VBA\01_01_Variablen.xls\mdlOverflow

```
Public Sub ErrorOverflow ()
   Dim lngLong    As Long
   Dim intInteger As Integer
   Dim bytByte    As Byte

   bytByte = 128
   intInteger = 32767

   ' Ab hier immer ein Überlauf
   lngLong = 32767 + 2 - 32740
   lngLong = intInteger + 2 - 32740
   lngLong = bytByte + bytByte - 20
End Sub
```

Das Ergebnis dieser drei Berechnungen ist in allen Fällen ein Wert im Bereich des Datentyps `Byte`. Die Zielvariable, die das Ergebnis aufnehmen soll, ist sogar ein `Long` und deshalb dürfte die Zuweisung an diese Variable keine Probleme machen. Dennoch gibt es in allen drei Fällen einen Laufzeitfehler durch Überlauf.

Nun, die Zuweisung selbst bereitet auch keine Probleme. Bevor aber der Zielvariablen das Ergebnis einer Berechnung zugewiesen wird, muss erst alles, was rechts vom Gleichheitszeichen steht, berechnet und zwischengespeichert werden.

Vom Interpreter, der den Quellcode während der Laufzeit in Maschinencode übersetzt, wird aber standardmäßig zum Speichern von Zahlen der Datentyp benutzt, der am wenigsten Platz beansprucht. Wenn beispielsweise ein Ganzzahlenwert nicht mehr in den `Byte`-Typ, aber in einen `Integer` passt und kein angehängtes Typenkennzeichen besitzt, verwendet der Interpreter auch den Datentyp `Integer`. Dafür werden dann zwei Bytes reserviert. Passen die Absolutwerte rechts vom Gleichheitszeichen in den Integerbereich, das Ergebnis oder ein Zwischenergebnis aber nicht mehr, gibt es einen Überlauf.

Dem können Sie abhelfen, indem Sie an die Zahlen Typenkennzeichen hängen. Das zwingt den Interpreter, für diese Zahl den angegebenen Datentyp zu benutzen.

```
lngLong = 32767& + 2 - 32740
lngLong = intInteger + 2& - 32740
```

Nachfolgende Tabellenübersicht (Tabelle 1.5) zeigt die Typenkennzeichen der Datentypen:

Tabelle 1.5
Typenkennzeichen für verschiedene Datentypen

Datentyp	Typenkennzeichen	Datentyp	Typenkennzeichen
Byte	Kein	Currency	@
Boolean	Kein	Decimal	Kein
Integer	%	Date	Kein
Long	&	String, variable Länge	$
Single	!	String, feste Länge	Kein
Double	#	Variant	Kein

Eine weitere Möglichkeit besteht darin, die explizite Typenumwandlung zu benutzen, um eine Variable für diese Berechnung in den gewünschten Datentyp umzuwandeln.

```
lngLong = CLng(bytByte) + bytByte – 20
```

Wenn Sie im Quelltext Zahlen statt Variablen zur Berechnung einsetzen, kann es bei Berechnungen zu Überläufen kommen, obwohl die Zuweisung des Ergebnisses an eine Variable ohne Probleme funktionieren würde. Hängen Sie in diesem Fall Typenkennzeichen des erforderlichen Datentyps an die Zahlen oder setzen die explizite Typumwandlung ein.

Nachfolgende Tabelle (Tabelle 1.6) zeigt die Umwandlungsfunktionen:

Tabelle 1.6
Umwandlungsfunktionen

Umwandlungsfunktion	Umwandeln in den Datentyp
CByte	Byte
CBool	Boolean
CInt	Integer
CLng	Long
CCur	Currency
CDbl	Double
CSng	Single
CDate	Date
CStr	String
CVar	Variant

1.4.4 Variablendeklaration

VBA lässt Ihnen die Freiheit, Variablen zu deklarieren oder diese einfach durch die Benutzung eines beliebigen Variablennamens zu erzeugen.

Der Typ einer Variablen, die man ohne eine Deklaration erzeugt hat, ist generell ein `Variant`. Dieser Datentyp ist unheimlich flexibel und kann nahezu alle Daten mit Ausnahme von benutzerdefinierten Typen und Strings fester Länge aufnehmen. Diese Flexibilität erfordert aber auch einen größeren Verwaltungsaufwand, der sich durch einen höheren Speicherbedarf und eine geringere Verarbeitungsgeschwindigkeit negativ bemerkbar macht.

Am besten meiden Sie den Typ `Variant`, aber verteufeln Sie ihn auch nicht. Auch dieser hat durchaus seine Daseinsberechtigung. Deklarieren Sie beispielsweise im Kopf einer Funktion einen Parameter mit einem bestimmten Typ, können Sie nur noch diesen Typ an die Funktion übergeben, obwohl diese vielleicht mit anderen Datentypen auch richtige Ergebnisse liefern würde. Sie erhöhen durch den Einsatz eines Variants somit die Flexibilität Ihrer Funktion.

Unerwünschte Datentypen können Sie immer noch innerhalb dieser Funktion eliminieren. Dazu steht Ihnen in VBA die Funktion `VarType` zur Verfügung, mit der Sie nachprüfen können, welchen Datentyp eine Variable besitzt. VBA besitzt einige vordefinierte Konstanten, die mit den Rückgabewerten der Funktion `VarType` bei den entsprechenden Typen korrespondieren, wie Sie der folgenden Tabelle (Tabelle 1.7) entnehmen können.

Tabelle 1.7
Rückgabewerte der Funktion VarType

Rückgabewert	Vordefinierte VBA-Konstante	Bedeutung
0	vbEmpty	Nicht initialisiert
1	vbNull	Keine gültigen Daten
2	vbInteger	Integer
3	vbLong	Long
4	vbSingle	Single
5	vbDouble	Double
6	cbCurrency	Currency
7	vbDate	Date
8	vbString	String
9	vbObject	Object
10	vbError	Fehlerwert
11	vbBoolean	Wahrheitswert
12	vbVariant	Nur bei Datenfeldern Typ Variant
13	vbDataObject	Datenzugriffsobjekt
14	vbDecimal	Decimal
17	vbByte	Byte
8192	vbArray	Array Zu diesem Wert wird immer noch der Wert des Typs addiert, aus dem das Array besteht.

Zu beachten ist, dass bei einem Array zu dem Wert, der den Typ angibt, laut Onlinehilfe noch der Wert 8192 addiert wird. Dieser ominöse Zahl 8192 ist eigentlich das gesetzte Bit Nummer 13, das in diesem Fall als `Flag` (Kennzeichen) für ein Array dient (die Zählung der Bits beginnt bei null). An die Funktion `VarType` können auch keine benutzerdefinierten Typen als Argument übergeben werden. Nachfolgendes Beispiel (Listing 1.3) zeigt den Gebrauch der Funktion `VarType`:

Listing 1.3
Beispiele\01_Grundlagen_VBA\01_01_Variablen.xls\mdlGetMyVarType

```
Public Sub testGetMyVarType()
   Dim varDummy            As Variant
   Dim alngDummy(1 To 2)   As Long
   Dim avarDummy(1 To 2)   As Variant

   ' Stringvariable
   varDummy = CStr(12)
   MsgBox GetMyVarType(varDummy), , "strDummy"

   ' Longvariable
   varDummy = CLng(12)
   MsgBox GetMyVarType(varDummy), , "lngDummy"
```

Listing 1.3 (Forts.)
Beispiele\01_Grundlagen_VBA\
01_01_Variablen.xls\
mdlGetMyVarType

```
    ' Integervariable
    varDummy = CInt(12)
    MsgBox GetMyVarType(varDummy), , "intDummy"

    ' Doublevariable
    varDummy = CDbl(12)
    MsgBox GetMyVarType(varDummy), , "dblDummy"

    ' Singlevariable
    varDummy = CSng(12)
    MsgBox GetMyVarType(varDummy), , "dblDummy"

    ' Array, Datentyp Long
    MsgBox GetMyVarType(alngDummy), , "Array Long"

    ' Array, Datentyp Variant
    MsgBox GetMyVarType(avarDummy), , "Array Variant"

End Sub

Public Function GetMyVarType(ByVal varTyp As Variant) As String
    Dim lngTyp As Long

    lngTyp = VarType(varTyp)

    If (lngTyp And 2 ^ 13) > 0 Then
        ' Bit Nummer 13 ist gesetzt (8192),
        ' es handelt sich also um ein Array
       GetMyVarType = "Array, "

        ' Dieses gesetzte Bit 13 löschen, damit man
        ' den Typ unabhängig davon ermitteln kann
       lngTyp = lngTyp And Not (2 ^ 13)
    End If

    ' Rückgabewert der Funktion VarType ohne Bit 13 auswerten
    ' und zusammen mit der Info, ob es sich um ein Array handelt,
    ' als Funktionsergebnis zurückgeben
    Select Case lngTyp
       Case vbEmpty
          GetMyVarType = GetMyVarType & "Nicht initialisiert"
       Case vbNull
          GetMyVarType = GetMyVarType & "Null, keine gültigen Namen"
       Case vbInteger
          GetMyVarType = GetMyVarType & "Integer"
       Case vbLong
          GetMyVarType = GetMyVarType & "Long"
       Case vbSingle
          GetMyVarType = GetMyVarType & "Single"
       Case vbDouble
          GetMyVarType = GetMyVarType & "Double"
       Case vbCurrency
          GetMyVarType = GetMyVarType & "Currency"
       Case vbDate
          GetMyVarType = GetMyVarType & "vbDate"
       Case vbString
          GetMyVarType = GetMyVarType & "String"
```

```
        Case vbObject
            GetMyVarType = GetMyVarType & "Object"
        Case vbError
            GetMyVarType = GetMyVarType & "Error"
        Case vbBoolean
            GetMyVarType = GetMyVarType & "Boolean"
        Case vbVariant
            GetMyVarType = GetMyVarType & "Variant (bei Arrays)"
        Case vbDataObject
            GetMyVarType = GetMyVarType & "DataObject"
        Case vbByte
            GetMyVarType = GetMyVarType & "Byte"
    End Select

End Function
```

Listing 1.3 (Forts.)
Beispiele\01_Grundlagen_VBA\
01_01_Variablen.xls\
mdlGetMyVarType

Ein weiteres Problem bei nicht deklarierten Variablen ist der, dass sich falsch eingegebene Variablennamen beim Programmieren und während der Laufzeit unauffällig verhalten. Erst wenn Sie den Wert einer Variablen weiterverarbeiten wollen und stattdessen den Wert einer neuen, leeren Variablen benutzen, weil es beispielsweise beim Eingeben des Namens zu einem Buchstabendreher gekommen ist, bekommen Sie ein fehlerhaftes Ergebnis. Ein Laufzeitfehler, der Sie auf einen falschen Namen hinweisen würde, wird leider nicht ausgelöst.

Dem können Sie aber entgegentreten indem Sie in der Entwicklungsumgebung von Excel (VBE) unter dem Menüpunkt EXTRAS | OPTIONEN | EDITOR einen Haken bei VARIABLENDEKLARATION ERFORDERLICH setzen (Abbildung 1.2).

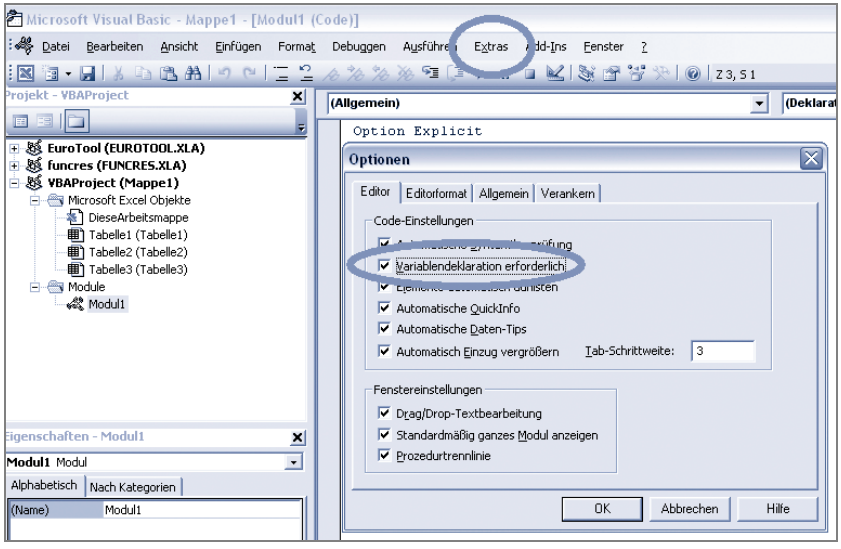


Abbildung 1.2
Variablendeklaration erzwingen

Dass diese Option aktiv ist, erkennen Sie, wenn Sie ein neues Modul in Ihr Projekt einfügen. In der ersten Zeile des Moduls finden Sie dann die Anweisung `Option Explicit`. Steht in dem entsprechenden Codemodul diese Anweisung, meckert der Interpreter bei der Programmausführung, wenn Sie dort eine nicht deklarierte Variable einsetzen.

Deklarieren Sie unbedingt die eingesetzten Variablen! Am besten machen Sie sofort in der Entwicklungsumgebung unter dem Menüpunkt EXTRAS | OPTIONEN | EDITOR ein Häkchen bei VARIABLENDEKLARATION ERFORDERLICH.

Sie sollten, wenn möglich, auch vorhandene Module nachträglich mit diesem Feature ausstatten, indem Sie diese zwei Worte in die erste Zeile einfügen. Sie haben dann zwar möglicherweise etwas Arbeit, um alle vorhandenen Variablen zu deklarieren, der Aufwand lohnt sich aber in jedem Fall.

Bei konsequenter Nutzung der expliziten Variablendeklaration beschleunigt sich auch die Ausführung Ihres Codes, denn der Speicherplatz für diese Variablen wird bereits vor der eigentlichen Codeausführung bereitgestellt und der Zeitaufwand während der Laufzeit für die Datenanalyse und Variablenerzeugung entfällt.

Bei der Deklaration sollten Sie unbedingt auch den Datentyp mit angeben. Tun Sie dies nicht, haben Sie es automatisch mit Variantvariablen zu tun. Oft werden Variantvariablen auch unbeabsichtigt angelegt.

Wenn Sie mehrere Deklarationen in einer Zeile durchführen wollen, benutzen Sie für jede zu deklarierende Variable den `As`-Abschnitt und geben dort explizit den Datentyp an. So vermeiden Sie das unbeabsichtigte Anlegen von Variantvariablen!

Folgendes Konstrukt findet man häufig in Quellcodes von unerfahrenen Anwendern oder Umsteigern, die solch eine Deklaration aus anderen Programmiersprachen kennen. Damit sollen eigentlich drei Stringvariablen angelegt werden:

```
Dim varVariable1, varVariable2, strVariable3 As String
```

Tatsächlich wird aber nur `strVariable3` als String deklariert, die beiden anderen sind automatisch Variantvariablen, weil bei ihnen der `As`-Abschnitt fehlt.

1.4.5 DefType

Eine weitgehend unbekannte Möglichkeit bietet die Anweisung `DefType`, mit der Sie auf Modulebene Standarddatentypen festlegen können. Diese Typen gelten für:

- Variablen
- Rückgabetypen von Funktionen
- Property-Get Prozeduren in Klassen
- Argumente von Funktionen und Prozeduren

Wenn eines dieser aufgeführten Elemente ohne Typ, also ohne den `As`-Abschnitt, deklariert wird und der Anfangsbuchstabe mit dem als Argument an `DefType` übergebenen Buchstaben übereinstimmt, wird der entsprechende Standardtyp benutzt. Folgendes Beispiel (Listing 1.4) dient zur Demonstration:

Listing 1.4
Beispiele\01_Grundlagen_VBA\01_01_Variablen.xls\mdlDefType

```
DefInt E-J
DefLng L-R
DefDbl D
DefStr S

' Zur Ansicht im Objektkatalog
Public iInteger
Public mLong

Public Sub testDefType()
   Dim lngDummy, strDummy
   Dim intDummy, dblDummy
   MsgBox GetMyVarTyp(lngDummy), , "lngDummy"
   MsgBox GetMyVarTyp(strDummy), , "strDummy"
   MsgBox GetMyVarTyp(intDummy), , "intDummy"
   MsgBox GetMyVarTyp(dblDummy), , "dblDummy"
End Sub
```

Die zum Ausführen benötigte Funktion `GetMyVarType` wurde im Listing 1.3 vorgestellt und ist deshalb an dieser Stelle weggelassen worden.

Sie können mit der `Deftype`-Anweisung auch ganze Buchstabenbereiche definieren, für den der angegebene Datentyp als Standard festgelegt wird. Mit `DefLng A-Z` legen Sie für alle Variablen den Standardtyp als `Long` fest, bei `DefLng L-R` beispielsweise nur für Variablen, die mit `L, M, N, O, P, Q` oder `R` beginnen (Abbildung 1.3). Es spielt dabei überhaupt keine Rolle, ob der Anfangsbuchstabe groß- oder kleingeschrieben ist, weil VBA bei Variablennamen generell nicht zwischen Groß- und Kleinschreibung unterscheidet.

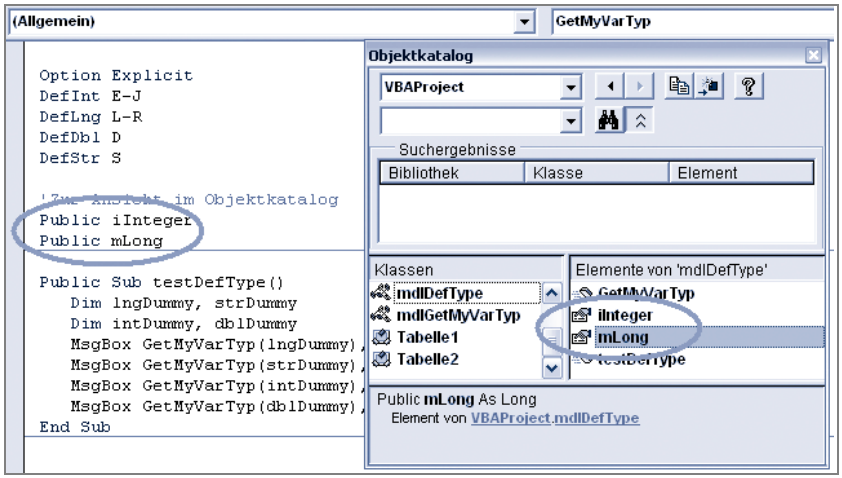


Abbildung 1.3
Deftype

In der folgenden Tabelle (Tabelle 1.8) sind alle Typen aufgeführt, die Sie für die `DefType`-Anweisung verwenden können:

DefType	Typ	DefType	Typ
DefBool	Boolean	DefDate	Date
DefByte	Byte	DefDbl	Double
DefInt	Integer	DefStr	String
DefLng	Long	DefSng	Single
DefCur	Currency	DefVar	Variant

Tabelle 1.8
Deftype-Anweisungen

Selbstverständlich können Sie bei der Deklaration immer noch explizit den Datentyp mit angeben und das würde ich Ihnen auch schon wegen der besseren Übersicht empfehlen. Ein weiterer Punkt, der gegen den generellen Einsatz von `DefType` spricht, ist die Verwendung von Präfixen, die den Typ und die Gültigkeit einer Variablen erkennen lassen.

Beim Einsatz dieser Notation haben Sie es nämlich häufig mit gleichen Anfangsbuchstaben für die unterschiedlichsten Datentypen zu tun. Beispielsweise wird

für modulweite Variablen immer das Präfix `m` vergeben, und zwar unabhängig vom Datentyp. Vergeben Sie jetzt für den Buchstaben M mit der `DefType`-Anweisung einen Standardwert, so besitzen alle modulweit gültigen Variablen den gleichen Datentyp, wenn Sie bei der Deklaration den `As`-Abschnitt weglassen.

1.4.6 Gültigkeitsbereich

In Prozeduren und Funktionen reicht ein einfaches `Dim` zur Deklaration einer Variablen. Dort ist die Gültigkeit standardmäßig auf die aktuelle Prozedur/Funktion begrenzt.

Bei Variablen mit modulweitem Gültigkeitsbereich sollten Sie bei der Deklaration im Deklarationsabschnitt des Moduls das Schlüsselwort `Private` verwenden. Mit `Dim` deklarierte Modulvariablen sind dort aber auch automatisch `Private`.

Soll die Variable mappenweit verfügbar sein, benutzen Sie `Public`, das Schlüsselwort `Global` ist dafür zwar auch möglich, sollte aber nicht mehr benutzt werden. Es ist noch ein Relikt aus längst vergangenen Zeiten und nur noch aus Kompatibilitätsgründen vorhanden.

Benutzen Sie außerhalb von Funktionen und Prozeduren generell die Schlüsselwörter `Private` oder `Public`, anstatt `Dim` zu verwenden. Nur durch eines dieser Schlüsselworte wird die Gültigkeit unabhängig vom Objekt und vom Ort der Deklaration eindeutig festgelegt.

Das Einsatzgebiet von `Private` und `Public` ist auf Klassen- oder Modulebene beschränkt. Im Deklarationsabschnitt eines Moduls sollten Sie immer eines dieser Schlüsselworte verwenden, obwohl es natürlich auch ohne die explizite Angabe des Gültigkeitsbereichs funktioniert. Das Weglassen dieser Schlüsselworte hat aber leider nicht zur Folge, dass der Gültigkeitsbereich dann generell nur auf das Modul beschränkt ist.

Ohne die explizite Angabe des Gültigkeitsbereiches gibt es für die Gültigkeit je nach Typ der Variablen gravierende Unterschiede, wobei es auch eine Rolle spielt, ob sich die Deklarationsanweisung in Standard- oder Klassenmodulen befindet. Nachfolgend eine Übersicht (Tabelle 1.9) der Gültigkeitsbereiche verschiedener Objekte bei Deklarationen ohne die Schlüsselwörter `Private` oder `Public`:

Tabelle 1.9
Gültigkeitsbereiche bei Deklaration ohne `Private` oder `Public`

Typ	Deklaration nur mit Dim bei Modulen	Deklaration nur mit Dim bei Klassenmodulen oder UserForms
Konstanten	`Private`	`Private`
Enum-Typen	`Public`	`Private`. Auch `Public` ergibt die Gültigkeit `Private`
Variablen	`Private`	`Private`
Benutzerdefinierte Typen	`Public`	Nicht möglich. `Private` ist immer nötig
Funktionen und Prozeduren	`Public`	`Public` (Eigenschaft/Methode)
Eigenschaften (Property)	`Public`	`Public`
API-Deklarationsanweisungen	`Public`	Nicht möglich. `Private` ist immer nötig

Denken Sie auch daran, dass Variablen, die in Prozeduren und Funktionen deklariert sind, eine höhere Priorität besitzen als modul- oder mappenweit gültige Variablen. Wenn diese mit dem gleichen Namen ausgestattet sind, wird in der Prozedur die Variable benutzt, die den niedrigsten Gültigkeitsbereich an der Einsatzstelle hat. Ein doppelt vergebener Name löst in diesem Fall keinen Fehler aus, da diese in verschiedenen Gültigkeitsbereichen deklariert wurden, sie stellen aber eine potenzielle Fehlerquelle dar.

Selbstverständlich haben Sie in der Prozedur mit dem doppelten Variablennamen auch noch Zugriff auf die modulweit deklarierte Variable gleichen Namens. Sie müssen dazu lediglich den Modulnamen durch einen Punkt getrennt vor den Variablennamen stellen. Besser ist es auf jeden Fall, gleiche Namen ganz zu vermeiden. Zur Demonstration folgender Code (Listing 1.5):

Listing 1.5
Beispiele\01_Grundlagen_VBA\
01_01_Variablen.xls \mdlDouble

```
Private strMyText As String

Public Sub DoubleVariable()
   Dim strMyText As String

   ' Unter Angabe des Modulnamens Setzen
   ' der modulweit gültigen Variable
   mdlDouble.strMyText = "ABC"

   ' Setzen der prozedurweit gültigen Variable
   strMyText = "DEF"

   ' Unter Angabe des Modulnamens Zugriff auf
   ' die modulweit gültige Variable
   MsgBox mdlDouble.strMyText, , "Modulweit"

   ' Zugriff auf die prozedurweit gültige Variable
   MsgBox strMyText, , "Prozedurweit"

End Sub
```

1.4.7 Lebensdauer

Soll eine Variable in einer Prozedur eine unbegrenzte Lebensdauer (nicht zu verwechseln mit der Gültigkeit) haben, verwenden Sie die `Static`-Anweisung. Haben Sie eine Variable mit `Static` deklariert, behält diese ihren Wert, bis das Modul zurückgesetzt oder neu gestartet wird. Das bedeutet, dass die Variable ihren Wert zwischen den Prozeduraufrufen behält.

Ihre Daseinsberechtigung hat die `Static`-Anweisung beispielsweise bei Funktionen, die laufend aufgerufen werden, und in denen jedes Mal eine Variable mit einem zeitaufwändig zu berechnenden Wert gefüllt werden muss. Wenn sich dieser Wert dann auch noch während der gesamten Lebensdauer der Mappe nicht oder nicht sehr häufig ändert, haben Sie einen hervorragenden Kandidaten für `Static`.

Auch Funktionen mit optionalen Parametern sind ein legitimes Einsatzgebiet, wenn dort die Staticvariable nur dann neu berechnet werden muss, wenn dieser optionale Parameter auch wirklich mit übergeben wurde. Andernfalls kann man

sich das Berechnen sparen und verwendet einfach den vorherigen Wert weiter. Sie können auch ganze Prozeduren oder Funktionen als `Static` deklarieren, alle Variablen darin behalten dann zwischen den Aufrufen ihre Werte.

Eine immer wieder kontrovers diskutierte Frage ist die, ob Objektvariablen, die in einer Prozedur oder Funktion angelegt wurden, vor dem Beenden zurückgesetzt werden müssen. Besonders Personen mit Programmiererfahrung in anderen Sprachen schwören darauf und setzen die Objektvariablen generell auf `Nothing`, bevor die Prozedur/Funktion beendet ist.

Nun, VB lässt Ihnen die Freiheit, das zu tun, räumt aber auch selbst beim Beenden der Prozedur/Funktion auf. Das bedeutet, dass Objektvariablen, welche auf Prozedurebene deklariert sind, nach Beendigung der Prozedur automatisch aus dem Speicher entfernt werden.

Ich habe auch noch keinen stichhaltigen Beleg dafür gefunden, dass irgendwelche Nachteile dadurch entstehen, sich in dieser Beziehung auf VB(A) zu verlassen. Es ist aber andererseits sicher kein Fehler, bei Objektvariablen vorher selbst mit `Set Objektvariable = Nothing` aufzuräumen und somit den Zeitpunkt von Terminate-Ereignissen, zum Beispiel bei Klassen, selbst zu bestimmen.

1.5 Konstanten

Konstanten werden überall dort eingesetzt, wo feste Werte benötigt werden. Im Gegensatz zu Variablen kann dieser Wert aber während der Laufzeit nicht mehr verändert werden, auch nicht unbeabsichtigt. Der Umstand, dass eine Konstante schon bei der Deklaration einen Wert zugewiesen bekommt, macht Ihren Code übersichtlicher und somit leichter nachvollziehbar.

Setzen Sie in Ihrem Quellcode Konstanten statt Zahlen ein, haben Sie die Möglichkeit, Ihr Programm schneller an veränderte Gegebenheiten anzupassen. Wenn Sie beispielsweise den Mehrwertsteuersatz als eine Konstante anlegen, brauchen Sie bei einer Änderung des Satzes nicht den gesamten Quellcode anzupassen. Es muss dann lediglich an einer Stelle dieser Wert geändert werden.

1.5.1 Normale Konstanten

Nachfolgend die Syntax für die Deklarationsanweisung einer Konstanten (optionale Parameter sind in eckige Klammern eingeschlossen, das Zeichen »|« steht für ein ODER):

```
[Public|Private] Const Name [As Typ] = Wert
```

Wenn Sie bei der Deklaration den Gültigkeitsbereich weglassen, ist die Konstante automatisch `Private`. In Prozeduren und Funktionen wird der Gültigkeitsbereich nie angegeben, die Gültigkeit der Konstanten ist dort auch nur auf diese Prozedur beschränkt. Noch eine Besonderheit von Konstanten ist, dass sie in Klassenmodulen nicht öffentlich (`Public`) gemacht werden können.

Der Datentyp der Konstanten kann optional festgelegt werden, wird dieser Parameter weggelassen, benutzt VBA den Datentyp, der für den angegebenen Wert am besten geeignet scheint.

VBA und Excel besitzen jede Menge eingebaute Konstanten, die Sie sich im Objektkatalog anschauen können. Benutzen Sie möglichst auch diese vordefinierten Konstanten bei der Übergabe an Funktionen. Das macht Ihren Code lesbarer, weil diese Konstanten zum Teil schon die Bedeutung im Namen tragen.

1.5.2 Enum-Typen

Eine Besonderheit von Konstanten stellt der für VBA neue Aufzählungstyp `Enum` dar, mit dem sich beispielsweise Konstanten für Eigenschaften von Objekten zusammenfassen lassen. Wenn Sie im Objektkatalog älterer Excelversionen nachschauen, werden Sie feststellen, dass dieser Typ eigentlich ein alter Hut ist, es ist nun aber erstmals möglich, selbst solche Typen zu erstellen.

Der praktische Nutzen ist aber doch sehr beschränkt, denn es lassen sich nur Konstanten vom Datentyp `Long` darin speichern. Wenn Sie zu älteren Excelversionen kompatibel bleiben wollen, deklarieren Sie diesen Typ am besten überhaupt nicht, es ist meiner Ansicht nach auch wirklich kein großer Verlust.

Mit der `Enum`-Anweisung wird ein Typ deklariert, der eine ganze Reihe von Konstanten des Datentyps `Long` aufnehmen kann. Die Deklaration kann aber nicht innerhalb von Prozeduren oder Funktionen erfolgen.

Hier folgt die Syntax für die `Enum`-Anweisung, die Ähnlichkeit mit der `Type`-Anweisung hat (optionale Parameter sind in eckige Klammern eingeschlossen, das Zeichen »|« steht für ein `ODER`):

```
[Public|Private] Enum Typname
   Elementname [= Longwert]
End Enum
```

Wenn Sie den Gültigkeitsbereich weglassen, ist die Konstante automatisch `Public`, d.h., der Typ ist im gesamten Projekt verfügbar. In Klassenmodulen können `Enum`-Typen nicht öffentlich gemacht werden. Sie können diesen Typ zwar in einer Klasse als `Public` deklarieren, aber nach außen hin wird dieser trotzdem nicht sichtbar.

Bei den Elementnamen gelten die gleichen Regeln wie die zur Benennung von Variablen. Zusätzlich können Sie optional jedem Element einen Longwert zuweisen. Andere Datentypen sind nicht erlaubt. Wird nichts zugewiesen, ist der Wert beim ersten Element null, bei anderen um eins größer als das vorhergehende Element. Sie können beliebig viele Elemente anlegen.

Nachfolgend ein kleines Beispiel (Listing 1.6) zum Einsatz von `Enum`:

```
Private Enum Tax
   maxi = 16
   midi = 8
   Mini = 0
End Enum
```

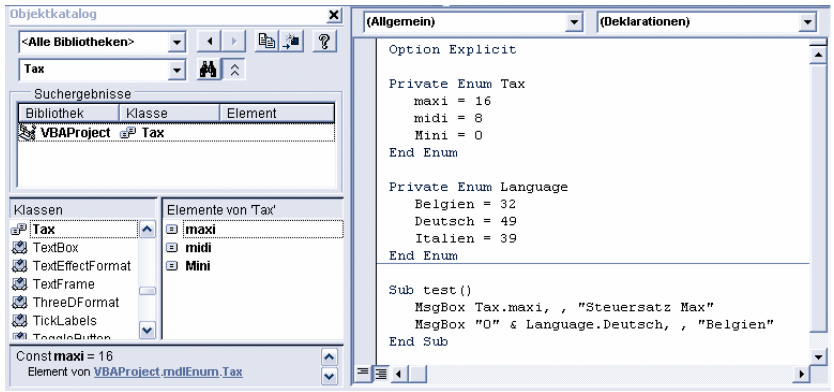
Listing 1.6
Beispiele\01_Grundlagen_VBA\
01_01_Variablen.xls\mdlEnum

Listing 1.6 (Forts.)
Beispiele\01_Grundlagen_VBA\
01_01_Variablen.xls\mdlEnum

```
Private Enum Language
   Belgien = 32
   Deutsch = 49
   Italien = 39
End Enum

Sub test()
   MsgBox Tax.maxi, , "Steuersatz Max"
   MsgBox "0" & Language.Deutsch, , "Belgien"
End Sub
```

Den erstellten Aufzählungstyp können Sie nach erfolgreicher Deklaration auch im Objektkatalog (Abbildung 1.4) wieder finden.

Abbildung 1.4
Enum-Typ im Objektkatalog



1.6 DoEvents

Es bereitet immer wieder Schwierigkeiten, eine Schleife durch eine Aktion von außen zu verlassen. Wenn Sie beispielsweise durch den Klick auf einen Button die Schleifenausführung beenden wollen, bekommen Sie ernsthaft Probleme. Das Klickereignis wird einfach nicht abgearbeitet, weil Sie sich noch mitten im Programmablauf befinden.

Um dem abzuhelfen, kommt die Anweisung `DoEvents` zum Einsatz. Damit wird der Anwendung die Möglichkeit gegeben, anstehende Ereignisse abzuarbeiten und auch sonstige Aktualisierungen durchzuführen. Der Programmablauf in der aufrufenden Prozedur wird in dieser Zeit gestoppt.

Um solch einen Schleifenabbruch zu realisieren, benötigen Sie eine Variable, die sowohl in der Prozedur mit der zu beendenden Schleife gültig ist als auch in dem Klickereignis des entsprechenden Buttons geändert werden kann. Den Wert dieser Variable fragen Sie nach jedem Aufruf von `DoEvents` in der Schleife ab. Ist in dem Klickereignis dieser Wert vorher auf `Wahr` gesetzt worden, kann die Schleife verlassen werden, ansonsten fahren Sie mit der normalen Programmausführung fort. Denken Sie für den nächsten Aufruf aber auch an das Zurücksetzen der Abbruchvariablen auf `False`.

In dem folgenden Beispiel (Listing 1.7) wird auf einem Tabellenblatt eine Schaltfläche mit dem Namen `cmdLoop` und der Caption `Endlosschleife starten` eingefügt. Ein Klick auf den Button startet die Endlosschleife, ein weiterer Klick darauf bewirkt, dass die Schleife verlassen wird. In das Klassenmodul dieses Tabellenblattes gehört nachstehender Code:

Listing 1.7
Beispiele\01_Grundlagen_VBA\
01_02_Allgemein.xls\

```
Private mblnLoopActive As Boolean

Private Sub cmdLoop_Click()

   ' Wahrheitswert der booleschen Abbruchvariable umkehren
   mblnLoopActive = Not mblnLoopActive

   If mblnLoopActive Then

      ' Schleife läuft gerade nicht (mblnLoopActive wurde
      ' vorher negiert). Beschriftung des Buttons
      ' ändern und Schleife starten (Prozedur myLoop)
      cmdLoop.Caption = "Endlosschleife beenden"
      myLoop
      MsgBox "Jetzt ist die erste Ereignisprozedur abgearbeitet"

   Else

      ' Schleife läuft gerade (mblnLoopActive wurde
      ' vorher negiert). Beschriftung des Buttons ändern.
      cmdLoop.Caption = "Endlosschleife starten"
      MsgBox "Jetzt ist die zweite Ereignisprozedur abgearbeitet"

   End If
End Sub

Private Sub myLoop()
   Dim i As Currency

   ' Beginn der Prozedur anzeigen
   MsgBox "Endlosschleife startet"

   Do ' Endlosschleife

      ' Ereignisse werden abgearbeitet
      DoEvents

      ' Wenn Abbruchbedingung erfüllt, Schleife verlassen
      If Not mblnLoopActive Then Exit Do

      ' Zähler erhöhen und Stand alle 10000 Durchläufe
      ' in der Statusleiste ausgeben
      i = i + 1
      If (i Mod 10000) = 0 Then _
         Application.StatusBar = "Durchlauf Nr. : " _
            & Format(i, "#,##0")

   Loop ' Zurück zum Beginn der Schleife
```

Listing 1.7 (Forts.)
Beispiele\01_Grundlagen_VBA\
01_02_Allgemein.xls\

```
    ' Statusbar zurücksetzen
    Application.StatusBar = False

    ' Ende der Prozedur anzeigen
    MsgBox "Endlosschleife beendet"
End Sub
```

Übertreiben Sie es aber nicht mit dem Einsatz von `DoEvents`, selbst wenn es in diesem Beispiel in jedem Durchlauf eingesetzt wird. Sie sollten `DoEvents` am besten nur alle paar Durchläufe einsetzen, je sparsamer, desto besser, denn der Programmablauf wird dadurch erheblich verlangsamt.

Sie sollten zusätzlich dafür sorgen, dass die gleiche Prozedur während der Laufzeit nicht noch einmal aufgerufen wird. Wie Sie beim Ausführen des Beispieles sehen konnten, wird beim erneuten Anklicken des gleichen Objekts das Klickereignis ein zweites Mal ausgeführt, obwohl die erste Ereignisprozedur noch nicht komplett abgearbeitet war.

Achtung

Ereignisprozeduren, die wiederum Ereignisse auslösen können, können ohne Gegenmaßnahmen eine Kettenreaktion auslösen und zum völligen Einfrieren der Anwendung führen, sodass nur noch ein gewaltsames Beenden des Prozesses hilft.

Man könnte das wirkungsvoll verhindern, indem man die Ereignisse mit `Application.EnableEvents` ausschaltet. Dann hätte man aber sicher kein `DoEvents` verwendet, denn man möchte damit ja schließlich erreichen, dass die Ereignisse abgearbeitet werden.

Eine mögliche Lösung ist eine boolesche, anwendungsweit gültige Variable, die beim ersten Ereignis auf `Wahr` gesetzt wird. In jeder anderen Ereignisprozedur, deren Ausführung verhindert werden soll, wird am Anfang der Wert dieser Variablen abgefragt und die Prozedur verlassen, wenn dieser Wert `Wahr` ist.

1.7 Benutzerdefinierte Tabellenfunktionen

Wenig bekannt und daher immer wieder ein Grund zur Irritation ist die Tatsache, dass benutzerdefinierte Tabellenfunktionen keine andere Zelle ändern können. Noch nicht einmal die `OnTime`-Methode funktioniert.

Der Grund ist darin zu suchen, dass die Änderung einer anderen Zelle eine Neuberechnung auslösen könnte, die wiederum andere Zellen ändert und eine Neuberechnung auslösen könnte, die andere Zellen ändert, usw. Das würde dann sehr schnell zum völligen Blockieren der Anwendung führen. Wenn Sie die

Funktion (Listing 1.8) dagegen aus dem Direktbereich oder durch einen Aufruf aus einer Prozedur testen, können Sie ohne Probleme Zellen ändern und beliebige Aktionen ausführen.

Listing 1.8
Beispiele\01_Grundlagen_VBA\01_02_Allgemein.xls\mdlChangeCell

```
Public Function ChangeCell(rngCell As Range) As String

   ChangeCell = "Zelle " & rngCell.Address & _
                " Hintergrundfarbe ändern"

   rngCell.Interior.Color = vbRed

End Function
```

Wird diese Funktion beispielsweise in der Zelle A2 mit der Formel »=ChangeCell(A2)« benutzt, steht anschließend in dieser Zelle der Text »Zelle A2 Hintergrundfarbe ändern«. Die Hintergrundfarbe selbst wird aber nicht angetastet.

Die Ausführung der Zeile »objZelle.Interior.Color=vbRed« wird ganz einfach übersprungen. Testen Sie dagegen die Funktion aus einer Prozedur, wird die Hintergrundfarbe von Zelle A2 ohne Probleme geändert.

Funktionen, die als Tabellenfunktionen eingesetzt werden, können keinerlei Code ausführen, der irgendein Ereignis auslösen könnte. Es gibt keinen Laufzeitfehler, die Anweisungen werden einfach ignoriert.

```
Private Sub TestChangeCell()
   MsgBox ChangeCell(Worksheets(1).Range("A2"))
End Sub
```

Der Vollständigkeit halber sei erwähnt, dass Sie Excel trotzdem austricksen können. Wenn Sie die OnTime-Methode mit einem API-Timer nachbauen, wird Excel nicht misstrauisch.

Die Benutzung einer Callback-Funktion ist aber nicht ganz unproblematisch. Schleicht sich in solch eine Rückruffunktion ein unbehandelter Fehler ein oder setzt man darin einen Haltepunkt, reißt es die ganze Anwendung in den Abgrund. Wenn Sie API-Funktionen einsetzen, verlassen Sie die schützende Hülle von VBA, speichern Sie deshalb sicherheitshalber vor der ersten Benutzung ihre Anwendung. Näheres zu der API finden Sie in Kapitel 3.

Unter Excel 97 gibt es zudem den Operator AddressOf noch nicht, worauf man besonders aus Kompatibilitätsgründen achten sollte. Den Operator kann man zwar nachbauen und solch eine Mappe lässt sich durch bedingte Kompilierung auch versionsunabhängig machen, es ist aber kein leichtes Unterfangen.

Trotz aller Bedenken will ich Ihnen den Code (Listing 1.9) nicht vorenthalten, der Einsatz dieser Funktionen geschieht aber wie bei allen API-Funktionen auf eigene Gefahr.

Der folgende Code gehört in ein allgemeines Modul:

Listing 1.9
Beispiele\01_Grundlagen_VBA\
01_02_Allgemein.xls\mdlApiTimer

```
Private Declare Function SetTimer
   Lib "user32" ( _
   ByVal hwnd As Long, _
   ByVal nIDEvent As Long, _
   ByVal uElapse As Long, _
   ByVal lpTimerFunc As Long _
   ) As Long

Private Declare Function KillTimer
   Lib "user32" ( _
   ByVal hwnd As Long, _
   ByVal nIDEvent As Long _
   ) As Long

Private mlngTimer As Long
Private mrngCell As Range

Private Sub ApiTimer( _
   ByVal hwndOwner As Long, _
   ByVal lngWindowMessage As Long, _
   ByVal hlngRückTimerKennung As Long, _
   ByVal lngTickCount As Long)
   ' Diese Funktion wird vom API-Timer zurückgerufen

   ' Jeder unbehandelte Fehler hat fatale Folgen
   On Error Resume Next

   ' Timer zerstören
   KillTimer 0, mlngTimer

   ' Zellfarbe ändern
   mrngCell.Interior.Color = vbRed

   ' Objektvariable löschen
   Set mrngCell = Nothing

End Sub

Public Function ChangeCell(rngCell As Range) As String

   ChangeCell = "Zelle " & rngCell.Address & _
                " Hintergrundfarbe ändern"

   Set mrngCell = rngCell

   ' Timer nach 1000 Millisekunden auslösen
   mlngTimer = SetTimer(0, 0, 1000, _
        AddressOf ApiTimer)
End Function
```

1.8 Parameterübergabe ByVal versus ByRef

Sie können Prozeduren/Funktionen aufrufen und dabei Parameter übergeben, Prozeduren können aber per Definition keinen Wert zurückgeben. Doch wenn Sie glauben, dass das die ganze Wahrheit ist, dann haben Sie sich getäuscht.

Es ist leider kaum bekannt, dass man in der aufgerufenen Funktion/Prozedur die Variablen ändern kann, die als Parameter übergeben wurden. Diese Variablen werden dann unter Umständen auch in der aufrufenden Prozedur geändert. Die eventuell unterschiedlichen Namen der Variablen in der aufrufenden bzw. aufgerufenen Prozedur spielen dabei überhaupt keine Rolle.

Das Problem oder das Feature – je nach Blickwinkel – tritt immer dann auf, wenn die Übergabe als Referenz (`ByRef`) erfolgt. `ByRef` bedeutet, dass die Adresse der Variablen übergeben wird. `ByRef` ist Voreinstellung und ohne ein ausdrückliches `ByVal` wird immer die Speicheradresse und somit die Kontrolle darüber aus der Hand gegeben.

Führen Sie folgendes Beispiel (Listing 1.10) aus:

Listing 1.10
Beispiele\01_Grundlagen_VBA\
01_02_Allgemein.xls\
mdlByValByRef

```
Public Sub ByValByRef()
   Dim strText As String

   ' Originalvariable
   strText = "Original"

   MsgBox strText, , "Text vor Übergabe"

   ' Parameter werden ohne Klammern übergeben
   myByValSub strText
   MsgBox strText, , "Original nach der Abarbeitung von myByValSub"

   ' Bei call werden Parameter mit Klammern übergeben
   Call myByRefSub(strText)
   MsgBox strText, , "Original nach der Abarbeitung von myByRefSub"

End Sub

Public Sub myByValSub(ByVal strByValText As String)

   ' Text ändern
   strByValText = "Geändert"
   ' Geänderten Variableninhalt ausgeben
   MsgBox strByValText, , "Text in myByValSub"

End Sub

Public Sub myByRefSub(strByRefText As String)

   ' Text ändern
   strByRefText = "Geändert"
   ' Geänderten Variableninhalt ausgeben
   MsgBox strByRefText, , "Text in myByRefSub"

End Sub
```

Wie man beim Ausführen sieht, ist der Inhalt der Originalvariablen, die als Parameter an die Prozedur `myByRefSub` übergeben wurde, nach der Ausführung geändert worden.

Die Übergabe als Referenz kann somit zu Fehlern führen, die außergewöhnlich schwer zu finden sind, besonders dann, wenn man diese Eigenschaft nicht kennt. Das Tolle daran ist andererseits, dass man Prozeduren und auch Funktionen schreiben kann, die mehrere Werte gleichzeitig manipulieren können. Und das funktioniert sogar ohne Variablen, die auf Modulebene deklariert oder global gültig sind.

Übrigens bedienen sich sehr viele API-Funktionen dieser Funktionalität. Sie manipulieren die Übergabeparameter und liefern gleichzeitig als Funktionsergebnis einen Wert zurück, der Aufschluss darüber gibt, ob die Aufgabe erfolgreich erledigt wurde.

Die Übergabe einer Variablen als Wert kostet natürlich etwas mehr Zeit, da ja erst eine Kopie der Variablen erstellt werden muss. Diese Kopie wird dann letztendlich auf den Stapelspeicher (Stack) gelegt und von dort holt sich die aufgerufene Funktion oder Prozedur die Übergabeparameter ab.

Bei der Übergabe als Referenz stehen dort die Speicheradressen, ab der die eigentlichen Daten zu finden sind. Wenn Sie die Prozeduren und Funktionen aber nicht millionenfach hintereinander aufrufen, werden Sie keine großen Geschwindigkeitsunterschiede feststellen.

Um eine Variable als Wert zu übergeben, obwohl im Funktionskopf `ByRef` angegeben ist, brauchen Sie bei der Übergabe die Variable nur in Klammern einschließen. Im vorangegangenen Beispiel (Listing 1.10) brauchen Sie dazu den Funktionsaufruf nur folgendermaßen zu ändern:

```
Call myByRefSub((strText))
```

1.9 Performance steigern

VBA ist langsam, das kann niemand bestreiten. Deshalb ist es besonders wichtig, bei zeitaufwändigen Berechnungen ein Optimum an Geschwindigkeit zu bekommen. Leider ist es aber so, dass die meisten Programme durch Unwissen langsamer laufen als möglich. In vielen Fällen merken Sie das nicht, weil der Zeitbedarf für andere Sachen wie zum Beispiel das Formatieren von Zellen erheblich höher ist. Bei benutzerdefinierten Tabellenfunktionen mit vielen Vergleichen können sich aber schon ein paar kleinere Optimierungen spürbar bemerkbar machen.

1.9.1 Select Case versus If Then

Einen messbaren Zeitgewinn erzielen Sie meistens schon mit recht einfachen Mitteln. Wenn Sie beispielsweise eine Variable auf viele Bedingungen testen müssen, verwenden Sie statt `If` und `Then` die Anweisung `Select Case`. Das ist

übersichtlich und in den meisten Fällen flinker als der einfache Einsatz von `If` und `Then`. Der Test, ob eine einzige Bedingung erfüllt ist, läuft dagegen mit `If` und `Then` schneller ab.

Zeitvorteil durch Select Case

Folgendes Beispiel (Listing 1.11) vergleicht den Zeitbedarf der `Select Case`- mit der `If Then`-Anweisung.

Listing 1.11
Beispiele\01_Grundlagen_VBA\
01_03_Zeitgewinn.xls\
mdlCondition

```
Public Sub Condition1()
   Dim dtmBegin   As Date
   Dim dtmEnd     As Date
   Dim strTime1   As String
   Dim strTime2   As String
   Dim strTest    As String
   Dim i          As Long

   ' Vergleichsvariable
   strTest = InputBox("Vergleichswert", "Bedingungen", "1234")
   If strTest = "" Then Exit Sub

   ' Zeitpunkt Schleifenbeginn speichern
   dtmBegin = Time
   'zwanzig Mio. Durchläufe
   For i = 1 To 20000000
      'Vergleich durchführen
      Select Case strTest
         Case "1234"
            i = i
         Case "2345"
            i = i
         Case "3456"
            i = i
         Case "4567"
            i = i
         Case "5678"
            i = i
         Case "6789"
            i = i
      End Select
   Next
   ' Zeitpunkt Schleifenende speichern
   dtmEnd = Time
   ' Zeitbedarf speichern
   strTime1 = "Zeit 'Select case' = " & _
      Format(dtmEnd - dtmBegin, "s ""Sekunden""")

   ' Zeitpunkt Schleifenbeginn speichern
   dtmBegin = Time
   'zwanzig Mio. Durchläufe
   For i = 1 To 20000000
      'Vergleich durchführen
      If strTest = "1234" Then i = i
      If strTest = "2345" Then i = i
      If strTest = "3456" Then i = i
```

Listing 1.11 (Forts.)
Beispiele\01_Grundlagen_VBA\
01_03_Zeitgewinn.xls\
mdlCondition

```
        If strTest = "4567" Then i = i
        If strTest = "5678" Then i = i
        If strTest = "6789" Then i = i
    Next
    ' Zeitpunkt Schleifenende speichern
    dtmEnd = Time
    ' Zeitbedarf speichern
    strTime2 = "Zeit 'If then' = " & _
        Format(dtmEnd - dtmBegin, "s ""Sekunden""")

    ' Ergebnis ausgeben
    MsgBox strTime1 & vbCrLf & strTime2, , "Zeitvergleich"

End Sub
```

Tabelle 1.10
Zeitbedarf im Vergleich
(`If Then` versus `Select Case`)

Vergleichen mit	Zeitbedarf (AMD Mobile Athlon 4 1200 MHz)
`Select Case (strTest = "1234")`	9 Sekunden
`If Then (strTest = "1234")`	18 Sekunden
`Select Case (strTest = "6789")`	23 Sekunden
`If Then (strTest = "6789")`	18 Sekunden

Wenn Sie dieses kleine Programm ausführen, bekommen Sie in einer Messagebox die benötigte Zeit für beide Varianten ausgegeben. Bei der vorgegebenen Konstellation wird die Prüfung mit `Select Case` doppelt so schnell ausgeführt wie die Überprüfung auf Übereinstimmung mit `If` und `Then`.

Bei der Überprüfung einer Übereinstimmung mit der `Select Case`-Anweisung sollten Sie den `Case`-Zweig, der am wahrscheinlichsten erfüllt wird, ganz nach vorne setzen.

Das liegt daran, dass bei der `Select Case`-Anweisung eine weitere Überprüfung auf Übereinstimmung entfällt, wenn der erste `Case`-Abschnitt `Wahr` ist, während in diesem Listing bei `If` und `Then` immer alle Tests durchgeführt werden. Wenn Sie den Wert der zu überprüfenden Variable auf `6789` ändern, stellen Sie fest, dass die Überprüfung mit `Select Case` nun länger dauert. Jetzt werden nämlich auch bei der `Select Case`-Anweisung alle Bedingungen überprüft.

Die Vorteile spielt `Select Case` erst so richtig bei vielen Prüfungen aus. Wenn eine Bedingung `Wahr` ist, werden die anderen, dahinter liegenden Prüfungen gar nicht mehr durchgeführt. Aus diesem Grund ist es von Vorteil, wenn man schon beim Schreiben des Codes weiß oder zumindest erahnen kann, welche Bedingungen am häufigsten erfüllt werden. Auf diese Übereinstimmung sollte dann möglichst früh geprüft werden.

Select Case mit If Then nachbauen

Bei zeitkritischen Anwendungen können Sie natürlich auch versuchen, die Funktionalität von `Select Case` mit `If` und `Then` nachzubilden. Dazu müssen Sie lediglich die anderen Bedingungsprüfungen in den `Else`-Zweig der vorherigen legen. Das wird zwar rasch unübersichtlich, ist aber meistens schneller und auch im ungünstigsten Fall zumindest gleich schnell.

Nachfolgendes Beispiel (Listing 1.12) demonstriert dies:

Listing 1.12
Beispiele\01_Grundlagen_VBA\
01_03_Zeitgewinn.xls\
mdlCondition

```
Public Sub Condition2()
   Dim dtmBegin   As Date
   Dim dtmEnd     As Date
   Dim strTime1   As String
   Dim strTime2   As String
   Dim strTest    As String
   Dim i          As Long

   ' Vergleichsvariable
   strTest = InputBox("Vergleichswert", "Bedingungen", "1234")
   If strTest = "" Then Exit Sub

   ' Zeitpunkt Schleifenbeginn speichern
   dtmBegin = Time
   'zwei Mio. Durchläufe
   For i = 1 To 2000000
      'Vergleich durchführen
      Select Case strTest
         Case "1234"
            i = i
         Case "2345"
            i = i
         Case "3456"
            i = i
         Case "4567"
            i = i
         Case "5678"
            i = i
         Case "6789"
            i = i
      End Select
   Next
   ' Zeitpunkt Schleifenende speichern
   dtmEnd = Time
   ' Zeitbedarf speichern
   strTime1 = "Zeit 'Select case' = " & _
      Format(dtmEnd - dtmBegin, "s ""Sekunden""")

   ' Zeitpunkt Schleifenbeginn speichern
   dtmBegin = Time
   'zwei Mio. Durchläufe
   For i = 1 To 2000000
      'Vergleich durchführen
      If strTest = "1234" Then
         i = i
      Else
         If strTest = "2345" Then
            i = i
         Else
            If strTest = "3456" Then
               i = i
            Else
               If strTest = "4567" Then
                  i = i
```

Listing 1.12 (Forts.)
Beispiele\01_Grundlagen_VBA\01_03_Zeitgewinn.xls\mdlCondition

```
                Else
                   If strTest = "5678" Then
                      i = i
                   Else
                      If strTest = "6789" Then
                         i = i
                      End If
                   End If
                End If
             End If
          End If
       End If
    Next
    ' Zeitpunkt Schleifenende speichern
    dtmEnd = Time
    ' Zeitbedarf speichern
    strTime2 = "Zeit 'If then' = " & _
       Format(dtmEnd - dtmBegin, "s ""Sekunden""")

    ' Ergebnis ausgeben
    MsgBox strTime1 & vbCrLf & strTime2, , "Zeitvergleich"

End Sub
```

Tabelle 1.11
Zeitbedarf im Vergleich (Nachbau `Select Case`)

Vergleichen mit	Zeitbedarf (AMD Mobile Athlon 4 1200 MHz)
`Select Case (strTest = "1234")`	9 Sekunden
`If Then Else (strTest = "1234")`	6 Sekunden
`Select Case (strTest = "6789")`	23 Sekunden
`If Then Else (strTest = "6789")`	18 Sekunden

Auch hier gilt, dass die Überprüfung auf den Fall mit der höchsten Wahrscheinlichkeit ganz an den Anfang muss. Die Verschachtelung `If Then Else` ist bei gleicher Reihenfolge der Bedingungen um bis zu ca. 30% schneller als `Select Case` und im ungünstigsten Fall immer noch gleich schnell. Den Zeitvorteil erkauft man sich aber durch einen undurchsichtigen Code und man muss sich in jedem Fall wieder neu die Frage stellen, ob damit die Gesamtperformance der Anwendung entscheidend gesteigert werden kann.

1.9.2 Logische Operatoren

Wenn Sie viel mit `If` und `Then` arbeiten, kommt es häufig vor, dass Bedingungen kombiniert werden müssen. Also, dass zum Beispiel zwei oder mehr Bedingungen erfüllt sein müssen oder eine von vielen Bedingung ausreicht. Dazu gibt es normalerweise die logischen Operatoren `AND` und `OR`. Ich habe aber festgestellt, dass eine kombinierte Bedingungsprüfung mit logischen Operatoren keine sehr gute Idee ist. Bei diesen Prüfungen werden grundsätzlich alle Bedingungen überprüft, und zwar unabhängig davon, ob vorherige Bedingungen `Falsch` bzw. `Wahr` sind und eine weitere Überprüfung somit sinnlos wäre.

Operator And

Nachfolgend ein Beispiel (Listing 1.13) zum Ersatz der logischen UND-Verknüpfung:

Listing 1.13
Beispiele\01_Grundlagen_VBA\01_03_Zeitgewinn.xls\mdlCondition

```
Public Sub Condition3()
   Dim dtmBegin   As Date
   Dim dtmEnd     As Date
   Dim strTime1   As String
   Dim strTime2   As String
   Dim avarTest   As Variant
   Dim i          As Long
   Dim k          As Long

   ' Array mit Werten füllen
   avarTest = Array(80, 8)

   'Jedes Element des Arrays testen
   For k = 0 To UBound(avarTest)

      ' Zeitpunkt Schleifenbeginn speichern
      dtmBegin = Time
      'zwei Mio. Durchläufe
      For i = 1 To 2000000
         'Test, ob Zahl größer als 10 ist und durch 2, 5
         'und 8 ohne Rest teilbar ist
         If (avarTest(k) > 10) And ((avarTest(k) Mod 2) = 0) _
            And ((avarTest(k) Mod 5) = 0) And _
            ((avarTest(k) Mod 8) = 0) Then
               i = i
         End If
      Next
      ' Zeitpunkt Schleifenende speichern
      dtmEnd = Time

      ' Zeitbedarf speichern
      strTime1 = "Zeit 'Verknüpfung mit AND' = " & _
         Format(dtmEnd - dtmBegin, "s ""Sekunden""")

      ' Zeitpunkt Schleifenbeginn speichern
      dtmBegin = Time

      'zwanzig Mio. Durchläufe
      For i = 1 To 2000000
         'Test, ob Zahl größer als 10 ist und durch 2, 5
         'und 8 ohne Rest teilbar ist
         If (avarTest(k) > 10) Then
            If (avarTest(k) Mod 2 = 0) Then
               If (avarTest(k) Mod 5 = 0) Then
                  If (avarTest(k) Mod 8 = 0) Then
                     i = i
                  End If
               End If
            End If
         End If
      Next
```

Listing 1.13 (Forts.)
Beispiele\01_Grundlagen_VBA\01_03_Zeitgewinn.xls\mdlCondition

```
        ' Zeitpunkt Schleifenende speichern
        dtmEnd = Time
        ' Zeitbedarf speichern
        strTime2 = "Zeit 'Verschachteltes IF' = " & _
           Format(dtmEnd - dtmBegin, "s ""Sekunden""")

        ' Ergebnis für den aktuellen Vergleichswert ausgeben
        MsgBox "Test, ob Zahl größer als 10 ist und durch 2, " _
          & "5 und 8 ohne Rest teilbar ist" _
          & vbCrLf & strTime1 & vbCrLf & strTime2, , _
          "Zeitvergleich Wert = " & avarTest(k)
    Next

End Sub
```

Tabelle 1.12
Zeitbedarf im Vergleich (logischer Operator And)

Vergleichen mit	Zeitbedarf (AMD Mobile Athlon 4 1200 MHz)
`And (Zahl = 80)`	6 Sekunden
`If Then (Zahl = 80)`	4 Sekunden
`And (Zahl = 8)`	6 Sekunden
`If Then (Zahl = 8)`	1 Sekunde

Ersetzen Sie den logischen Operator `AND` durch verschachtelte Überprüfungen mit `If` und `Then`, muss die Prüfung auf den Fall mit der niedrigsten Wahrscheinlichkeit ganz am Anfang durchgeführt werden. Ist bereits an dieser Stelle keine Übereinstimmung vorhanden, brauchen die anderen Prüfungen nicht mehr vorgenommen werden.

Wenn Sie, wie hier in diesem Beispiel, erst eine Bedingung überprüfen und dann entscheiden, ob es überhaupt notwendig ist, die nächste zu überprüfen, können Sie viel Rechenzeit sparen.

Sie werden feststellen, dass die Überprüfung mit dem `AND`-Operator immer gleich lange dauert, unabhängig davon, ob bereits ein Teilergebnis falsch ist. Beim Nachbau mit `If` und `Then` dagegen spielt die Reihenfolge der Überprüfung eine große Rolle. Die Zeitersparnis kann besonders bei vielen Überprüfungen enorm sein und ist umso höher, je genauer man die Häufigkeitsverteilung der zu überprüfenden Werte kennt.

Operator Or

Ersetzen Sie den Operator `Or` durch verschachtelte Überprüfungen mit `If` und `Then`, testen Sie zuerst auf die wahrscheinlichste Übereinstimmung. Bei einer Übereinstimmung brauchen die anderen Prüfungen nicht mehr vorgenommen werden.

Während `AND`-Operatoren relativ einfach mit verschachtelten `If Then`-Überprüfungen nachgebaut werden können, müssen Sie beim Nachbau von logischem `OR` etwas mehr Gehirnschmalz aufwenden.

Wenn bereits die erste Bedingung erfüllt ist, soll die weitere Überprüfung abgebrochen und die weitere Ausführung des Programmcodes davon beeinflusst werden. Sie könnten jetzt in jeden `IF`-Zweig, dessen Bedingung erfüllt ist, den Programmcode schreiben, der dann weiter ausgeführt werden soll. Das ist aber in höchsten Maße redundant und bläht den Quellcode nur unnötig auf.

Eine hervorragende Alternative dazu bieten Unterprogramme, die aber durch den Prozeduraufruf den gesamten Programmablauf etwas langsamer machen. Bei längeren Codeabschnitten, die bei einer Übereinstimmung ausgeführt werden sollen, ist das sicher die erste Wahl.

Eine andere Möglichkeit besteht darin, eine boolesche Variable zu führen, die auf `WAHR` gesetzt wird, wenn keine der Bedingungen erfüllt sind. Das weitere Ausführen von Codeabschnitten wird dann vom Wert dieser Variablen abhängig gemacht Das erfordert aber noch eine zusätzliche `If Then` Anweisung, die immer ausgeführt wird und auch etwas Zeit kostet.

Im folgenden Beispiel (Listing 1.14) ist eine Möglichkeit dargestellt, ein logisches `Or` mit einem Zeitvorteil nachzubauen.

Listing 1.14
Beispiele\01_Grundlagen_VBA\
01_03_Zeitgewinn.xls\
mdlCondition

```
Public Sub Condition4()
   Dim dtmBegin   As Date
   Dim dtmEnd     As Date
   Dim strTime1   As String
   Dim strTime2   As String
   Dim avarTest   As Variant
   Dim i          As Long
   Dim k          As Long
   Dim blnIsFalse As Boolean

   ' Array mit Werten füllen
   avarTest = Array(80, 6)

   'Jedes Element des Arrays testen
   For k = 0 To UBound(avarTest)
      ' Zeitpunkt Schleifenbeginn speichern
      dtmBegin = Time
      'zwei Mio. Durchläufe
      For i = 1 To 2000000
         'Test, ob Zahl größer als 10 ist oder durch 8, 5
         'und 6 ohne Rest teilbar ist
         If (avarTest(k) > 10) Or ((avarTest(k)) Mod 8 = 0) _
            Or ((avarTest(k) Mod 5) = 0) Or _
            ((avarTest(k) Mod 6) = 0) Then
               i = i
         End If
      Next
      ' Zeitpunkt Schleifenende speichern
      dtmEnd = Time
      ' Zeitbedarf speichern
      strTime1 = "Zeit 'Verknüpfung mit OR' = " & _
         Format(dtmEnd - dtmBegin, "s ""Sekunden""")

      ' Zeitpunkt Schleifenbeginn speichern
      dtmBegin = Time
      ' zwanzig Mio. Durchläufe
      For i = 1 To 2000000
         ' Test, ob Zahl größer als 10 ist oder durch 8, 5
         ' und 6 ohne Rest teilbar ist
         If (avarTest(k) > 10) Then
         Else
            If (avarTest(k) Mod 8 = 0) Then
            Else
               If (avarTest(k) Mod 5 = 0) Then
               Else
```

Listing 1.14 (Forts.)
Beispiele\01_Grundlagen_VBA\
01_03_Zeitgewinn.xls\
mdlCondition

```
                If (avarTest(k) Mod 6 = 0) Then
                Else
                    ' Keine Bedingung erfüllt
                    blnIsFalse = True
                End If
            End If
        End If
    End If
Next
If Not (blnIsFalse) = True Then
    ' Eine Bedingung wurde erfüllt
    i = i
End If
' Zeitpunkt Schleifenende speichern
dtmEnd = Time
' Zeitbedarf speichern
strTime2 = "Zeit 'Verschachteltes IF' = " & _
    Format(dtmEnd - dtmBegin, "s ""Sekunden""")

' Ergebnis für den aktuellen Vergleichswert ausgeben
MsgBox "Test, ob Zahl größer als 10 ist oder durch 8, " _
  & "5 und 6 ohne Rest teilbar ist" _
  & vbCrLf & strTime1 & vbCrLf & strTime2, , _
  "Zeitvergleich Wert = " & avarTest(k)
Next

End Sub
```

Tabelle 1.13
Zeitbedarf im Vergleich (logischer Operator Or)

Vergleichen mit	Zeitbedarf (AMD Mobile Athlon 4 1200 MHz)
`Or (Zahl = 80)`	6 Sekunden
`If Then (Zahl = 80)`	1 Sekunde
`Or (Zahl = 6)`	6 Sekunden
`If Then (Zahl = 6)`	5 Sekunden

1.9.3 Zeitvorteil durch GoTo

Außer bei Fehlerbehandlungen sollten Sie `GoTo` als Verzweigung zu einer Zeile oder Sprungmarke innerhalb des Quellcodes weglassen. Dafür gibt es schließlich Funktionen, Unterprogramme und VBA-Anweisungen wie `Select Case`, die es in den Anfängen von Basic noch nicht gab. Diesen Grundsatz sollte jeder Programmierer verinnerlicht haben.

Wie überall bestätigen aber Ausnahmen die Regel. In einigen wenigen Fällen bringt der Einsatz von `GoTo` noch echte Vorteile und dort ist der Einsatz auch angebracht. Die Vorteile müssen aber deutlich sein, beispielsweise wenn der Code dadurch entscheidend verkürzt, schneller oder übersichtlicher wird.

Im folgenden Beispiel (Listing 1.15) können Sie durch den Einsatz von `GoTo` einen echten Geschwindigkeitsvorteil erzielen:

Listing 1.15
Beispiele\01_Grundlagen_VBA\
01_03_Zeitgewinn.xls\mdlGoTo

```
Public Sub GoodGoto()
   Dim lngTest    As Long
   Dim i          As Long
   Dim dtmBegin   As Date
   Dim dtmEnd     As Date
   Dim dtmTime1   As Date
   Dim dtmTime2   As Date

   lngTest = 1

   ' Bei großen Schleifen langsam
   dtmBegin = Now() ' Zeit Beginn der Schleife
   For i = 1 To 1000000
      ' Es werden immer alle Vergleiche durchgeführt
      If (lngTest And 2 ^ 0) Or (lngTest And 2 ^ 1) Or _
         (lngTest And 2 ^ 2) Or (lngTest And 2 ^ 3) Or _
         (lngTest And 2 ^ 4) Or (lngTest And 2 ^ 5) Or _
         (lngTest And 2 ^ 6) Or (lngTest And 2 ^ 7) Or _
         (lngTest And 2 ^ 8) Or (lngTest And 2 ^ 9) Or _
         (lngTest And 2 ^ 10) Or (lngTest And 2 ^ 11) Or _
         (lngTest And 2 ^ 12) Or (lngTest And 2 ^ 13) Or _
         (lngTest And 2 ^ 14) Or (lngTest And 2 ^ 15) _
         Then
         ' Tu dies
      Else
         ' Tu das
      End If
   Next
   dtmEnd = Now() ' Zeit Ende der Schleife

   ' Zeitdauer speichern
   dtmTime1 = dtmEnd - dtmBegin

   ' Nicht gerade schön, aber schnell
   dtmBegin = Now() ' Zeit Beginn der Schleife
   For i = 1 To 1000000
      ' Es werden nur die Vergleiche bis zur ersten
      ' Übereinstimmung durchgeführt
      If lngTest And 2 ^ 0 Then GoTo DoIt
      If lngTest And 2 ^ 1 Then GoTo DoIt
      If lngTest And 2 ^ 2 Then GoTo DoIt
      If lngTest And 2 ^ 3 Then GoTo DoIt
      If lngTest And 2 ^ 4 Then GoTo DoIt
      If lngTest And 2 ^ 5 Then GoTo DoIt
      If lngTest And 2 ^ 6 Then GoTo DoIt
      If lngTest And 2 ^ 7 Then GoTo DoIt
      If lngTest And 2 ^ 8 Then GoTo DoIt
      If lngTest And 2 ^ 9 Then GoTo DoIt
      If lngTest And 2 ^ 10 Then GoTo DoIt
      If lngTest And 2 ^ 11 Then GoTo DoIt
      If lngTest And 2 ^ 12 Then GoTo DoIt
      If lngTest And 2 ^ 13 Then GoTo DoIt
      If lngTest And 2 ^ 14 Then GoTo DoIt
      If lngTest And 2 ^ 15 Then GoTo DoIt
```

Listing 1.15 (Forts.)
Beispiele\01_Grundlagen_VBA\01_03_Zeitgewinn.xls\mdlGoTo

```
        'Tu das
        GoTo DoItNot
DoIt:
        'Tu dies
DoItNot:
    Next
    dtmEnd = Now()  ' Zeit Ende der Schleife

    ' Zeitdauer speichern
    dtmTime2 = dtmEnd - dtmBegin

    ' Zeit ausgeben
    MsgBox "Zeit 1 = " & dtmTime1 * 24 * 3600 _
        & vbCrLf & _
        "Zeit 2 = " & dtmTime2 * 24 * 3600
End Sub
```

In diesem Listing ist die Schleife mit `GoTo` sechsmal schneller. Selbst im ungünstigsten Fall, wenn der Wert von `lngTest` auf null gesetzt wird, ist die Version mit `GoTo` trotzdem noch flinker.

1.9.4 Referenzierung und Objektvariablen

Bei häufigen Zugriffen auf die gleichen Objekte benutzen Sie am besten Objektvariablen oder die `With`-Anweisung, anstatt mit einer vollständigen Referenzierung zu arbeiten. Dies macht den Code kürzer, übersichtlicher und wartbarer.

Bei einer vollständigen Referenzierung muss jedes Mal auf die Objekte aller Verschachtelungsebenen zugegriffen werden, während Sie bei einer Objektvariablen direkt auf das Zielobjekt zugreifen können. Mit der `With`-Anweisung können Sie eine Reihe von Anweisungen für ein bestimmtes Objekt ausführen, ohne mehrmals den Namen angeben zu müssen. Ich selbst bevorzuge die Variante mit `With`, in vielen Fällen erspart man sich damit einen Zeilenumbruch, der zusammengehörende Codeteile optisch auseinander reißt.

Nun könnte man annehmen, dass die Referenzierung über eine Objektvariable gleichwertig zu der Referenzierung mit der `With`-Anweisung ist, da bei der `With`-Anweisung intern ja auch irgendeine Referenz auf das Objekt angelegt werden muss.

Wenn Sie Ihren Code auf Geschwindigkeit trimmen wollen, legen Sie mit einer Objektvariablen eine Referenz auf das Zielobjekt an. Damit sparen Sie erheblich Zeit gegenüber einer vollständigen Referenzierung.

Beim Testen des Zeitbedarfs hat mich das Ergebnis total überrascht. Dass die vollständige Referenzierung die langsamste ist, war mir dabei schon im Vorfeld klar. Ich hatte aber auch erwartet, dass die Version mit der `With`-Anweisung in etwa gleich schnell ist wie die Referenzierung über eine Objektvariable. Das ist aber nicht der Fall, die `With`-Anweisung legt im Hintergrund keine Objektvariable an.

Es scheint vielmehr so, dass der Interpreter während der Laufzeit einfach das, was hinter `With` steht, vor die Punkte setzt, die sich hinter einem Leerzeichen im `With`-Block befinden. Steht hinter dem `With` eine Objektvariable, wird die Objektvariable benutzt, befindet sich dahinter eine vollständige Referenzierung, verwendet der Interpreter diese Referenzierung.

Das Benutzen der With-Anweisung mit einer vollständigen Referenzierung bringt aber trotzdem noch etwas an Zeitersparnis gegenüber der vollständigen Referenzierung. Das wird daran liegen, dass dem Interpreter das zu benutzende Objekt schon zu Beginn des With-Blocks bekannt ist.

Benutzen Sie die With-Anweisung zusammen mit einer Objektvariablen hinter dem Schlüsselwort With, kombinieren Sie die Vorteile einer Objektvariablen mit denen der With-Anweisung.

Im nachfolgenden Beispiel (Listing 1.16) ein Vergleich:

Listing 1.16
Beispiele\01_Grundlagen_VBA\01_03_Zeitgewinn.xls \mdlWith

```
Public Sub WithAndTime()
   Dim objWS   As Worksheet
   Dim lngColumn1 As Long
   Dim lngColumn2 As Long
   Dim lngRow1    As Long
   Dim lngRow2    As Long
   Dim dtmBegin   As Date
   Dim dtmEnd     As Date
   Dim strTime1   As String
   Dim strTime2   As String
   Dim strTime3   As String
   Dim strTime4   As String
   Dim strDummy   As String
   Dim i          As Long

   ' 1. Zelle eines Bereichs
   lngColumn1 = 1 'Spalte A
   lngRow1 = 1 ' Zeile 1

   ' Letzte Zelle eines Bereichs
   lngColumn2 = 1 ' Spalte A
   lngRow2 = 1 'Zeile 1

   dtmBegin = Now() ' Zeit Beginn der Schleife
   For i = 1 To 400000
      strDummy = ActiveWorkbook.Worksheets(1).Range( _
         ActiveWorkbook.Worksheets(1).Cells(lngRow1, lngColumn1), _
         ActiveWorkbook.Worksheets(1).Cells(lngRow2, lngColumn2))
   Next
   dtmEnd = Now() ' Zeit Ende der Schleife
   ' Zeitbedarf mit vollständiger Referenzierung
   strTime1 = Format(dtmEnd - dtmBegin, "nn:ss")

   dtmBegin = Now() ' Zeit Beginn der Schleife
   Set objWS = ActiveWorkbook.Worksheets(1)
   For i = 1 To 400000
      strDummy = objWS.Range(objWS.Cells(lngRow1, lngColumn1), _
      objWS.Cells(lngRow2, lngColumn2))
   Next
   dtmEnd = Now() ' Zeit Ende der Schleife
   ' Zeitbedarf mit Objektvariable
   strTime2 = Format(dtmEnd - dtmBegin, "nn:ss")

   dtmBegin = Now() ' Zeit Beginn der Schleife
   With ActiveWorkbook.Worksheets(1)
      For i = 1 To 400000
         strDummy = .Range(.Cells(lngRow1, lngColumn1), _
            .Cells(lngRow2, lngColumn2))
      Next
   End With
```

Listing 1.16 (Forts.)
Beispiele\01_Grundlagen_VBA\01_03_Zeitgewinn.xls \mdlWith

```
    dtmEnd = Now() ' Zeit Ende der Schleife
    ' Zeitbedarf mit With und vollständiger Referenzierung
    strTime3 = Format(dtmEnd - dtmBegin, "nn:ss")

    dtmBegin = Now() ' Zeit Beginn der Schleife
    With objWS
       For i = 1 To 400000
          strDummy = .Range(.Cells(lngRow1, lngColumn1), _
             .Cells(lngRow2, lngColumn2))
       Next
    End With
    dtmEnd = Now() ' Zeit Ende der Schleife
    ' Zeitbedarf mit With und Objektvariable
    strTime4 = Format(dtmEnd - dtmBegin, "nn:ss")

    ' Meldungsausgabe
    MsgBox "Vollständige Referenzierung : " & strTime1 _
       & vbCrLf & _
       "Mit Objektvariable : " & strTime2 _
       & vbCrLf & _
       "Mit With und vollständiger Referenzierung: " _
       & strTime3 & vbCrLf & _
       "Mit With und Objektvariable: " & strTime4

End Sub
```

Tabelle 1.14
Zeitbedarf beim Referenzieren im Vergleich

Referenzieren mit	Zeitbedarf (Athlon 2400 +)
Vollständige Referenzierung	16 Sekunden
Referenzierung über Objektvariable	5 Sekunden
Benutzen der `With`-Anweisung mit vollständiger Referenzierung	13 Sekunden
Benutzen der `With`-Anweisung mit einer Objektvariablen	5 Sekunden

Deklarieren Sie bei Objektvariablen wenn möglich auch den Typ. Dann steht Ihnen `IntelliSense` zur Verfügung und die Ausführungsgeschwindigkeit sollte sich noch etwas erhöhen. `IntelliSense` ist, wie Sie sicher schon wissen, das nützliche Hilfsmittel in der Entwicklungsumgebung, bei dem Sie nach der Eingabe eines Punktes hinter einer Objektvariablen die Eigenschaften und Methoden des Objekts angezeigt bekommen und per Mausklick auswählen können.

1.9.5 Bildschirmaktualisierungen

Mit der VBA-Zeile `Application.ScreenUpdating=False` wird die Bildschirmaktualisierung deaktiviert (Listing 1.17). Damit kann man die Laufzeit eines Programms erheblich beschleunigen, sofern natürlich überhaupt sichtbare Änderungen an einem Tabellenblatt vorgenommen werden.

Listing 1.17
Beispiele\01_Grundlagen_VBA\01_03_Zeitgewinn.xls \mdlWith

```
Public Sub TestScreenUpdating()
   Dim lngRow           As Long
   Dim dtmBegin         As Date
   Dim dtmEnd           As Date
   Dim astrTime(1 To 4) As String
   Dim objWorksheet     As Worksheet
   Dim i                As Long
   Dim k                As Long

   ' Objektvariable anlegen
   Set objWorksheet = ActiveWorkbook.Worksheets(1)

   For i = 1 To 4
      ' Abhängig von i die Zeile und das Screenupdating festlegen
      If i = 1 Then Application.ScreenUpdating = False: lngRow = 1
      If i = 2 Then Application.ScreenUpdating = True: lngRow = 1
      If i = 3 Then Application.ScreenUpdating = False: lngRow = 99
      If i = 4 Then Application.ScreenUpdating = True: lngRow = 99

      dtmBegin = Now() ' Zeit Beginn der Schleife
      For k = 1 To 50000
         objWorksheet.Cells(lngRow, 1).Interior.ColorIndex = 3
      Next
      dtmEnd = Now() ' Zeit Ende der Schleife
      ' Zeitbedarf bestimmen und im Array speichern
      astrTime(i) = Format(dtmEnd - dtmBegin, "nn:ss")
   Next
   ' Meldungsausgabe
   MsgBox "Zeit sichtbarer Bereich ohne Aktualisierung: " _
      & astrTime(1) & vbCrLf & _
      "Zeit sichtbarer Bereich mit Aktualisierung: " _
      & astrTime(2) & vbCrLf & _
      "Zeit unsichtbarer Bereich ohne Aktualisierung: " _
      & astrTime(3) & vbCrLf & _
      "Zeit unsichtbarer Bereich mit Aktualisierung: " _
      & astrTime(4) & vbCrLf

   ' Blatt zurücksetzen
   objWorksheet.Cells.Clear
End Sub
```

Tabelle 1.15
Zeitbedarf beim Referenzieren im Vergleich

Aktionen	Zeitbedarf (Athlon 2400 +)
Sichtbarer Bereich ohne Aktualisierung	7 Sekunden
Sichtbarer Bereich mit Aktualisierung	38 Sekunden
Unsichtbarer Bereich ohne Aktualisierung	7 Sekunden
Unsichtbarer Bereich mit Aktualisierung	8 Sekunden

Bei Manipulationen des sichtbaren Bereiches sind damit durchaus Geschwindigkeitssteigerungen auf das Vierfache möglich. Sie dürfen aber nicht vergessen, die Bildschirmaktualisierung am Ende wieder auf `True` zu stellen.

Zum Einschalten der Aktualisierung brauchen Sie einfach nur die Codezeile `Application.ScreenUpdating=True` auszuführen.

1.9.6 Berechnungen ausschalten

Wenn Sie mit VBA Zellen von Tabellenblättern ändern, auf die sich Tabellenfunktionen beziehen, wird eine Neuberechnung angestoßen. Je nach Funktion und Anzahl der Berechnungen kann das bis fast zum völligen Stillstand der Anwendung führen.

Zwar arbeitet Excel unablässig an der Neuberechnung, die Abarbeitung des Programmcodes wird aber ausgebremst, da mit dem Programmablauf erst fortgefahren wird, wenn die Berechnung abgeschlossen ist.

In dem folgenden Beispiel (Listing 1.18) wird programmgesteuert die Zelle A10 im Tabellenblatt „Neuberechnung“ geändert. Auf diese Zelle bezieht sich eine Formel in der Zelle A9, die bei einer Änderung in A10 neu berechnet wird. In den ersten 50000 Schleifendurchläufen ist die automatische Neuberechnung aus-, in den nächsten 50000 eingeschaltet. Der Zeitbedarf mit und ohne automatische Neuberechnung wird ermittelt und ausgegeben.

Listing 1.18
Beispiele\01_Grundlagen_VBA\01_03_Zeitgewinn.xls\mdlCalculate

```
Public Sub TestCalculate()
   Dim dtmBegin          As Date
   Dim dtmEnd            As Date
   Dim astrTime(1 To 2) As String
   Dim objWorksheet      As Worksheet
   Dim i                 As Long
   Dim k                 As Long

   'Objektvariable anlegen
   Set objWorksheet = ActiveWorkbook.Worksheets("Neuberechnung")

   Application.ScreenUpdating = False

   For i = 1 To 2
      'Abhängig von i das Neuberechnen deaktivieren
      If i = 1 Then Application.Calculation = xlCalculationManual
      If i = 2 Then Application.Calculation = xlCalculationAutomatic

      dtmBegin = Now() ' Zeit Beginn der Schleife
      For k = 1 To 50000
         ' In Zelle A9 steht die Formel   =5^A10
         objWorksheet.Range("A10") = k Mod 10
      Next
      dtmEnd = Now() ' Zeit Ende der Schleife

      Application.Calculate
      ' Zeitbedarf bestimmen und im Array speichern
      astrTime(i) = Format(dtmEnd - dtmBegin, "nn:ss")
   Next

   Application.ScreenUpdating = True
   ' Meldungsausgabe
   MsgBox "Zeit ohne Neuberechnung: " _
      & astrTime(1) & vbCrLf & _
      "Zeit mit Neuberechnung: " _
      & astrTime(2)

End Sub
```

Aktionen	Zeitbedarf (Athlon 2400 +)
Neuberechnung ausgeschaltet	3 Sekunden
Neuberechnung eingeschaltet	18 Sekunden

Tabelle 1.16
Zeitbedarf mit der automatischen Neuberechnung im Vergleich

Achtung

Wenn Sie mit Matrixfunktionen arbeiten, die sich auf größere Bereiche beziehen, und in diesem Bereich programmgesteuert sehr viele Zellen ändern, kann das Ausschalten der automatischen Berechnung den Verlust einer zusätzlichen Pause nach sich ziehen.

Nachdem alle Zellen geändert sind, müssen Sie die automatische Berechnung wieder aktivieren, indem Sie vor dem Beenden der Prozedur die Codezeile `Application.Calculation = xlCalculationAutomatic` ausführen. Anschließend sollten Sie mit der Methode `Application.Calculate` eine Neuberechnung durchführen.

1.9.7 Weitere Optimierungsmöglichkeiten

Es gibt noch weitere Möglichkeiten, den Code zu optimieren. In den meisten Fällen ist aber die subjektiv wahrgenommene Geschwindigkeit nicht unbedingt höher, obwohl real doch ein Geschwindigkeitsvorteil erzielt wurde.

Führen Sie beispielsweise laufend Änderungen an Zellen aus, wird eine etwas schnellere Berechnung sich nicht unbedingt positiv auf die wahrgenommene Geschwindigkeit auswirken. In diesem Fall bringt Ihnen das Ausschalten der automatischen Berechnung und der Bildschirmaktualisierung weitaus mehr.

Bei benutzerdefinierten Tabellenfunktionen, die zudem noch häufig eingesetzt werden, kann ein kleiner Zeitvorteil dagegen spürbare Auswirkungen haben. In diesem Fall ist das Verhältnis Gesamtausführungsdauer zur Zeitersparnis weitaus höher.

Select vermeiden

Viele Objekte in Excel wie zum Beispiel Zellen (Bereiche), Spalten oder Zeilen besitzen die Methode `Select`. Das ist an sich eine wichtige Methode.

Sie sollten die Selektiererei von Zellen oder Bereichen aber nicht übertreiben. Damit können Sie die Performance eines selbst geschriebenen Programms recht schnell auf den Nullpunkt bringen, selbst wenn der Code sorgfältig auf Geschwindigkeit getrimmt wurde.

Wenn Sie die Ereignisse nicht durch `Application.EnableEvents = False` ausgeschaltet haben, wird bei jedem Select das Ereignis `Worksheet_SelectionChange` ausgelöst. Existiert eine solche Ereignisprozedur, wird diese abgearbeitet, wenn Sie vom Programmcode aus eine Zelle selektieren. Der Programmablauf in der

aufrufenden Prozedur stoppt so lange, bis der Code in dieser Ereignisprozedur fertig ausgeführt ist.

Folgender Code, eingefügt in das Klassenmodul eines Tabellenblatts, demonstriert sehr schön, dass die erste, anstoßende Ereignisprozedur erst beendet wird, wenn die Spalte größer 200 ist. Dann erst werden von hinten nach vorn alle Ereignisprozeduren nacheinander beendet.

Listing 1.19
Der Fluch von Select

```
Private Sub Worksheet_SelectionChange(ByVal Target As Range)
   Static lngEnd As Long

   ' Aufrufzählung bei Null beginnen
   If Target.Column < 10 Then lngEnd = 0

   ' Aufrufnummer erhöhen und ausgeben
   lngEnd = lngEnd + 1
   MsgBox "Aufruf Nummer : " & lngEnd

   If Target.Column < 200 Then
      ' Neue Zelle Selektieren, wenn Spalte < 200
      Me.Cells(Target.Row, Target.Column + 10).Select
   End If

   ' Zähler vor dem Beenden der Prozedur um eins verringern
   lngEnd = lngEnd - 1
End Sub
```

Neben Endlosschleifen wird bei vielen noch nicht abgearbeiteten Aufrufen recht schnell der Speicher auf dem Stack knapp. Das gibt dann einen schönen `Laufzeitfehler 28: nicht genügend Stapelspeicher`.

Hans W. Herber, der ultimative Excel-Guru, hat es in einem Newsgroup-Beitrag einmal so auf den Punkt gebracht: »Der Cursor ist kein Dackel, der Gassi geführt werden muss«.

Aber auch ohne Ereignisprozeduren kostet das Selektieren viel Zeit. Ein Select ist auch wirklich nur in den allerwenigsten Fällen notwendig. Anstatt erst eine Zelle zu selektieren und dann beispielsweise mittels `Selection.Value` zu arbeiten, sollten Sie direkt referenzieren. Das würde dann in etwa so aussehen:

Anstatt Folgendes einzusetzen,

```
Worksheets("Tabelle1").Range("A1").Select
Selection.Value=22
```

verwenden Sie lieber folgende Zeile:

```
Worksheets("Tabelle1").Range("A1").Value = 22
```

Aufgezeichnete »Makros« müssen grundsätzlich überarbeitet werden. Dort können Sie wahre Selektorgien sehen. Das ist aber auch nicht sehr verwunderlich, denn Sie selektieren beim Aufzeichnen ja auch tatsächlich diese Zellen.

Vermeiden von Variantvariablen

Variantvariablen verbrauchen nicht nur mehr Speicherplatz, die Berechnungen damit sind auch erheblich langsamer (Listing 1.20).

Listing 1.20
Beispiele\01_Grundlagen_VBA\01_03_Zeitgewinn.xls\mdlVariant

```
Public Sub TestVar()
   Dim dtmBegin          As Date
   Dim dtmEnd            As Date
   Dim astrTime(1 To 2) As String
   Dim adblVar(1 To 2)   As Double
   Dim avarVar(1 To 2)   As Variant
   Dim dblResult         As Double
   Dim k                 As Long

   avarVar(1) = 12000000
   avarVar(2) = 2
   adblVar(1) = 12000000
   adblVar(2) = 2

   dtmBegin = Now() ' Zeit Beginn der Schleife
   For k = 1 To 50000000
      dblResult = adblVar(1) / adblVar(2)
   Next
   dtmEnd = Now() ' Zeit Ende der Schleife
   ' Zeitbedarf bestimmen und im Array speichern
   astrTime(1) = Format(dtmEnd - dtmBegin, "nn:ss")

   dtmBegin = Now() ' Zeit Beginn der Schleife
   For k = 1 To 50000000
      dblResult = avarVar(1) / avarVar(2)
   Next
   dtmEnd = Now() ' Zeit Ende der Schleife
   ' Zeitbedarf bestimmen und im Array speichern
   astrTime(2) = Format(dtmEnd - dtmBegin, "nn:ss")

   ' Meldungsausgabe
   MsgBox "Zeit mit Variablen vom Typ Double: " _
      & astrTime(1) & vbCrLf & _
      " Zeit mit Variablen vom Typ Variant: " _
      & astrTime(2)

End Sub
```

Tabelle 1.17
Zeitbedarf bei Berechnungen mit Variantvariablen im Vergleich

Aktionen	Zeitbedarf (Athlon 2400 +)
Berechnungen mit Variablen vom Typ Double	5 Sekunden
Berechnungen mit Variablen vom Typ Variant	14 Sekunden

Logische Verknüpfungen

Benötigen Sie eine boolesche Variable, die den Wahrheitswert eines logischen Vergleichs enthält, können Sie Folgendes schreiben:

```
If Variable=32 Then
   blnXYZ = True
Else
   blnXYZ = False
End If
```

Weitaus schneller dagegen ist:

```
blnXYZ = (Variable=32)
```

In diesem Beispiel (Listing 1.21) können Sie den Zeitvorteil sehen:

Listing 1.21
Beispiele\01_Grundlagen_VBA\01_03_Zeitgewinn.xls\mdlVariant

```
Public Sub TestLogical()
   Dim dtmBegin         As Date
   Dim dtmEnd           As Date
   Dim astrTime(1 To 6) As String
   Dim lngValue         As Long
   Dim blnRes           As Boolean
   Dim k                As Long

   lngValue = 1234

   dtmBegin = Now() ' Zeit Beginn der Schleife
   For k = 1 To 100000000
      If lngValue = 2345 Then
         blnRes = True
      Else
         blnRes = False
      End If
   Next
   dtmEnd = Now() ' Zeit Ende der Schleife
   'Zeitbedarf bestimmen und im Array speichern
   astrTime(1) = Format(dtmEnd - dtmBegin, "nn:ss")

   dtmBegin = Now() ' Zeit Beginn der Schleife
   For k = 1 To 100000000
      blnRes = (lngValue = 2345)
   Next
   dtmEnd = Now() ' Zeit Ende der Schleife
   'Zeitbedarf bestimmen und im Array speichern
   astrTime(2) = Format(dtmEnd - dtmBegin, "nn:ss")

   ' Meldungsausgabe
   MsgBox "If lngValue = 2345 Then : " _
      & astrTime(1) & vbCrLf & _
      "blnRes = (ngValue = 2345) : " _
      & astrTime(2)

End Sub
```

Tabelle 1.18
Zeitbedarf bei Berechnungen mit booleschen Variablen

Aktionen	Zeitbedarf (Athlon 2400 +)
Setzen der booleschen Variable mit `If Then`	8 Sekunden
Direktes Setzen der booleschen Variable	4 Sekunden

Einen ähnlichen Zeitvorteil können Sie erwarten, wenn Sie statt

```
If blnRes = False Then
   blnRes = True
Else
   blnRes = False
End If
```

Folgendes einsetzen:

```
blnRes = Not blnRes
```

Sonstiges

Einen geringen Zeitvorteil können Sie erwarten, wenn Sie statt

```
If strDummy = "" Then
```

die Zeile

```
If Len(strDummy) = 0 Then
```

benutzen.

Potenzieren ist langsam, erheblich schneller ist dagegen das Multiplizieren. Sie sollten deshalb bei kleinen ganzzahligen Potenzen das Multiplizieren benutzen. Bei einem Zeitvergleich mit einem ähnlichen Code wie in Listing 1.21 wurden folgende Zeiten ermittelt:

Aktionen	Zeitbedarf (Athlon 2400 +)
Potenzieren (lngResult = 100& ^ 2)	24 Sekunden
Multiplizieren (lngResult = 100& * 100&)	4 Sekunden

Tabelle 1.19
Zeitbedarf bei Berechnungen mit Potenzen im Vergleich

1.10 Rekursionen

Rekursionen, d.h. sich selbst aufrufende Funktionen und Prozeduren, gelten meist zu Unrecht als extrem schwierig zu durchschauen. In sehr vielen Fällen ist das Lösen eines Problems damit einfacher zu bewerkstelligen und der Code wird sogar kürzer und übersichtlicher.

Immer wieder wird in diesem Zusammenhang die Berechnung einer Fakultät als Paradebeispiel benutzt. Das ist meiner Meinung nach ein hervorragendes Beispiel dafür, warum man gerade dafür nicht rekursive Funktionen benutzen sollte.

Beim Berechnen einer Fakultät werden alle natürlichen Zahlen von eins bis n miteinander multipliziert. Dabei können Sie iterativ in einer Schleife oder rekursiv mit sich selbst aufrufenden Funktionen oder Prozeduren vorgehen, wie in dem folgenden Beispiel (Listing 1.22) verdeutlicht wird.

Listing 1.22
Beispiele\01_Grundlagen_VBA\01_04_More.xls\mdlFakultaet

```
Public Function Fakultaet1(ByVal dblX As Double) As Double
   Fakultaet1 = 1
   If dblX = 1 Then Exit Function
   Fakultaet1 = Fakultaet1(dblX - 1) * dblX
End Function

Public Function Fakultaet2(ByVal dblX As Double) As Double
   Fakultaet2 = 1
   For dblX = 1 To dblX
       Fakultaet2 = Fakultaet2 * dblX
   Next
End Function

Public Sub test()
   Dim dtmBegin   As Date
   Dim dtmEnd     As Date
   Dim strTime1   As String
   Dim strTime2   As String
   Dim k          As Long
   dtmBegin = Now
   For k = 1 To 300000
      Fakultaet1 170
   Next
   dtmEnd = Now
   strTime1 = "Zeit Rekursiv = " & _
      Format(dtmEnd - dtmBegin, "s ""Sekunden""")

   dtmBegin = Now
   For k = 1 To 300000
      Fakultaet2 170
   Next
   dtmEnd = Now
   strTime2 = "Zeit Iterativ = " & _
      Format(dtmEnd - dtmBegin, "s ""Sekunden""")

   ' Meldungsausgabe
   MsgBox strTime1 & vbCrLf & strTime2
End Sub
```

Tabelle 1.20
Zeitbedarf Fakultätsberechnung

Aktionen	Zeitbedarf (Athlon 2400 +)
Rekursiv	13 Sekunden
Iterativ	3 Sekunden

Verwenden Sie rekursive Funktionen oder Prozeduren nur dann, wenn der zusätzliche Zeitbedarf für einen Funktions- oder Prozeduraufruf eine untergeordnete Rolle spielt. Ohne zwingenden Grund sollten Sie Rekursionen vermeiden, der Code sollte dadurch zumindest übersichtlicher werden und leichter zu durchschauen sein.

Das Aufrufen einer Funktion erfordert Zeit! Die Übergabe des Parameters als Referenz könnte zwar noch einen kleinen Geschwindigkeitsgewinn bewirken, aber die iterative Lösung ist auch dann noch viermal schneller. Der Code ist noch nicht einmal übersichtlicher und auch nur marginal kürzer. Die Benutzung der rekursiven Lösung ist also in diesem Fall keine gute Idee.

Hervorragend geeignet ist der Einsatz von Rekursionen beim Durchlaufen von Baumstrukturen. Sie müssen dann nur noch Code für das Durchlaufen einer Ebene schreiben und ihn verstehen. Untergeordnete Strukturen werden mit der gleichen Prozedur auf die gleiche Art durchlaufen. Das Wichtigste ist dabei

immer wieder das Finden einer geeigneten Abbruchbedingung, ohne die sich das Ausführen solch einer Funktion als tödlich für die Anwendung erweisen kann.

Im folgenden Beispiel (Listing 1.23) wird durch eine rekursive Prozedur eine Dateiliste erstellt und in ein Arbeitsblatt eingetragen. Dabei werden alle Unterverzeichnisse bis ins letzte Glied abgearbeitet.

Listing 1.23
Beispiele\01_Grundlagen_VBA\01_04_More.xls\mdlRekursion

```
Sub testFileList()
   Application.ScreenUpdating = False
   FileList "C:\Windows\System32", 2
   Application.ScreenUpdating = True
End Sub

Private Sub FileList( _
   ByVal strPath As String, _
   Optional lngRow As Long)

   Dim strFile       As String
   Dim varItem       As Variant
   Dim colDirectory  As New Collection
   Dim strFilepath   As String
   Dim curSize       As Currency
   Dim lngAttr       As Long
   Dim objDest       As Worksheet

   On Error Resume Next

   If Right(strPath, 1) <> "\" Then strPath = strPath & "\"

   Set objDest = Worksheets("Tabelle1")

   ' Bei allen Attributen kann es dauern
   lngAttr = vbDirectory Or vbHidden Or vbSystem

   ' Erste Datei dieser Ebene holen
   strFile = Dir(strPath & "*", lngAttr)

   With objDest

      ' Am Anfang den Inhalt löschen
      If lngRow = 0 Then .Range("A:D").ClearContents

      ' So lange durchlaufen, bis keine Datei mehr gefunden wird
      Do While strFile <> ""

         ' Punkte ( . , .. ) sind auch Verzeichnisse, aber
         ' übergeordnete. Die brauchen wir aber nicht
         If Right(strFile, 1) <> "." Then

            ' Pfad und Namen verbinden
            strFilepath = strPath & strFile

            If GetAttr(strFilepath) And vbDirectory Then
               ' Unterverzeichnisse zwischenspeichern,
               ' da Dir keine Rekursion verträgt
               colDirectory.Add strFilepath, strFilepath
            End If
```

Listing 1.23 (Forts.)
Beispiele\01_Grundlagen_VBA\01_04_More.xls\mdlRekursion

```
            ' Filelänge ermitteln
            curSize = FileLen(strFilepath)

            ' Infos eintragen
            lngRow = lngRow + 1

            ' Spalte 1 Datei
            .Cells(lngRow, 1) = strFile

            ' Spalte 2 Pfad
            .Cells(lngRow, 2) = strFilepath

            ' Hyperlink auf die Datei setzen
            .Hyperlinks.Add .Cells(lngRow, 2), strFilepath

            ' Spalte 3 Größe der Dateien
            .Cells(lngRow, 3) = Format(curSize, "0")
            .Cells(lngRow, 3).NumberFormat = "#,##0"

            ' Spalte 4 Erstellungsdatum
            .Cells(lngRow, 4) = FileDateTime(strFilepath)
            .Cells(lngRow, 4).NumberFormat = "DD DD.MM.YYYY hh:mm"

         End If

         ' Nächste Datei holen
         strFile = Dir()
      Loop
   End With

   ' Unterverzeichnisse rekursiv abarbeiten
   For Each varItem In colDirectory
      FileList varItem & "\", lngRow
   Next
End Sub
```

Die zusätzlichen Funktionsaufrufe fallen dabei zeitlich überhaupt nicht ins Gewicht. Das Eintragen der Ergebnisse und das Ermitteln der eigentlichen Informationen erfordert im Verhältnis zum Zeitbedarf der zusätzlichen Funktionsaufrufe gigantisch viel mehr an Zeit.

Wenn Sie Lust zum Frust haben, können Sie ja mal versuchen, das Beispiel so umzuschreiben, dass es iterativ funktioniert und trotzdem noch einigermaßen übersichtlich bleibt.

1.11 Sortieren und Mischen

Häufig kommt es vor, dass man Listen oder Datenfelder sortieren oder die Elemente zufällig verteilen – also mischen – will. Um programmgesteuert Zellinhalte zu sortieren, zeichnen Sie sich am besten mit dem Makrorekorder einen Sortiervorgang auf und passen diesen anschließend Ihren Bedürfnissen an.

Wenn Sie mit VBA Datenfelder sortieren wollen, ist aber der Umweg über ein Tabellenblatt in den meisten Fällen ungeeignet. Für solche Zwecke gibt es Sortieralgorithmen. Die zwei am häufigsten benutzten sind `Bubble-` und `Quicksort`. Beim Sortieren von Text sollten Sie auf Modulebene das Vergleichsverfahren festlegen, beispielsweise `Option Compare Text` oder `Option Compare Binary`.

Nachfolgend der Code, mit dem Sie den Zeitbedarf der verschiedenen Sortieralgorithmen ermitteln können:

Listing 1.24
Beispiele\01_Grundlagen_VBA\01_04_More.xls\mdlSort

```
Public Sub testSort()
   Dim dtmBegin   As Date
   Dim dtmEnd     As Date
   Dim strTime1   As String
   Dim strTime2   As String
   Dim alngMix1() As Long
   Dim alngMix2() As Long
   Dim i          As Long
   Dim lngCount1  As Long
   Dim lngCount2  As Long

   ReDim alngMix1(1 To 10000)

   ' Array erzeugen
   For i = 1 To UBound(alngMix1)
      alngMix1(i) = i
   Next

   ' Mischen
   Shuffle alngMix1
   alngMix2 = alngMix1

   ' ########## Quicksort ##########
   dtmBegin = Now 'Beginn Quicksort

   ' Array aufsteigend sortieren
   QuickSortArr alngMix1

   ' Array absteigend sortieren
   ' QuickSortArr varArray:=alngMix1, blnDown:=True
   dtmEnd = Now 'Ende Quicksort
   strTime1 = "Zeit Quicksort = " & _
      Format(dtmEnd - dtmBegin, "s ""Sekunden""")

   ' ########## Bubblesort ##########
   dtmBegin = Now 'Beginn Bubblesort

   ' Array aufsteigend sortieren
   lngCount1 = Bubblesort(alngMix2)

   dtmEnd = Now 'Ende Bubblesort
   strTime2 = "Zeit Bubblesort = " & _
      Format(dtmEnd - dtmBegin, "s ""Sekunden""")

   MsgBox strTime1 & vbCrLf & strTime2

End Sub
```

1.11.1 Bubblesort

`Bubblesort` ist leicht zu verstehen, aber im Allgemeinen langsamer als `Quicksort`. Der Algorithmus beruht auf dem Prinzip des direkten Austauschs, d.h., aufeinander folgende Elemente werden verglichen und gegebenenfalls vertauscht. Das Datenfeld wird mehrfach durchlaufen, wobei alle kleineren Elemente nach unten, alle größeren nach oben wandern. Die größeren Elemente steigen also langsam wie Blasen aus der Masse der Elemente auf, deshalb auch wahrscheinlich der Name. Das können Sie auf verschiedene gleichwertige Arten implementieren, ich beschreibe aber nur eine davon.

Beim ersten Durchlauf gehen Sie alle Elemente von hinten nach vorne durch und vergleichen sie mit dem ersten Element. Ist das erste größer, werden beide getauscht. Nach dem ersten Durchlauf ist dann das kleinste Element ganz vorne. Jetzt vergleichen Sie das zweite Element mit allen anderen dahinter und tauschen, wenn ein Element kleiner ist. Dies führen Sie weiter durch, bis alle Elemente sortiert sind.

Bei großen Datenfeldern nimmt aber der Zeitbedarf ganz schnell zu. Die Zahl der Durchläufe errechnet sich nach folgender Formel:

$$0{,}5 \cdot n \cdot (a_1 + a_n) - n$$

D.h., beim Sortieren von 10 Elementen werden 45 Vergleiche durchgeführt, bei 100 sind es 4.950 und bei 10.000 kommt man schon auf 49.995.000 Vergleiche. Bei wenigen Elementen ist Bubblesort durchaus eine Alternative zu anderen Sortiermethoden, aber bei größeren Arrays kommt dann schließlich auch irgendwann der schnellste Rechner ins Schwitzen.

Nachfolgend (Listing 1.25) ein Beispiel dazu:

Listing 1.25
Beispiele\01_Grundlagen_VBA\01_04_More.xls\mdlSort

```
Private Function Bubblesort(varArray As Variant)
   Dim i             As Long
   Dim k             As Long
   Dim m             As Long
   Dim lngLow        As Long
   Dim lngUp         As Long
   Dim varBuffer     As Variant

   If Not IsArray(varArray) Then Exit Function

   ' Grenzen des Arrays bestimmen
   lngLow = LBound(varArray)
   lngUp = UBound(varArray)

   For i = lngLow To lngUp
   ' Vom kleinsten zum größten Index

      For k = lngUp To i + 1 Step -1
      ' Vom größten zum Index i+1

         If varArray(i) > varArray(k) Then
            ' Die zwei gefundenen Elemente tauschen
            varBuffer = varArray(k)
```

```
            varArray(k) = varArray(i)
            varArray(i) = varBuffer
        End If
        m = m + 1

    Next

  Next
  Bubblesort = m
End Function
```

Listing 1.25 (Forts.)
Beispiele\01_Grundlagen_VBA\01_04_More.xls\mdlSort

1.11.2 Quicksort

Wenn es gilt, große Datenmengen zu sortieren, hat sich Quicksort bestens bewährt. Bereits 1962 wurde dieser rekursive Algorithmus von C.A.R. Hoare entdeckt und umfangreich untersucht.

Das Grundprinzip beruht darauf, den Sortierungsprozess in immer kleinere Häppchen aufzuteilen, die wiederum durch den gleichen Sortieralgorithmus sortiert werden. Die gesamte Datenmenge wird dabei durch ein Schnittelement, auch Pivotelement genannt, in zwei Teile zerlegt. Die Elemente links von diesem Pivotelement sollen am Ende einer Prozedur alle kleiner, die rechts davon alle größer als das Pivotelement sein. Danach wird die linke und die rechte Hälfte rekursiv weitersortiert.

Wenn Sie das praktisch umsetzen wollen, bestimmen Sie im einfachsten Fall das Pivotelement, indem Sie einfach ein Element benutzen, das sich in der Mitte befindet. Danach werden beide Partitionen durchlaufen, und zwar einmal vom Anfang (links) in Richtung Mitte und zum anderen vom Ende (rechts) in die Richtung des Pivotelements.

Dabei wird das aktuelle Element von links kommend mit dem Pivotelement verglichen. Ist das aktuelle kleiner, wird das nächste Element mit dem Pivotelement verglichen. Ist dagegen das aktuelle Element größer, beginnen Sie, die Elemente von rechts kommend mit dem Pivotelement zu vergleichen, und zwar so lange, wie das aktuelle Element auf der rechten Seite größer ist.

Haben Sie von links kommend ein Element gefunden, das größer als das Pivotelement ist, und von rechts kommend eins gefunden, das kleiner ist, werden die Elemente getauscht. Das wird aber nur gemacht, wenn die linke Position auch tatsächlich kleiner als die rechte ist.

Danach setzen Sie den Prozess beim nächsten Element mit dem gleichen Verfahren fort. Dies wird so lange fortgesetzt, wie die aktuelle rechte Position größer als die linke Position ist. Anschließend befinden sich im linken Teilbereich nur noch Elemente, die kleiner, und im rechten nur noch Elemente, die größer als das Pivotelement sind.

Die zwei Teilbereiche werden dann jeweils mit dem gleichen, vorher vorgestellten Verfahren bearbeitet. Dazu ruft sich die Prozedur selbst auf und übergibt dabei noch die Grenzen der Teilbereiche.

Es wird dann in der aufgerufenen Prozedur wieder ein Pivotelement festgelegt, von beiden Seiten kommend werden die Elemente durchlaufen und bei Bedarf getauscht, bis die linke Position größer als die rechte ist. Am Ende sind wieder alle Elemente links davon kleiner und alle Elemente rechts davon größer als das

Pivotelement. Die zwei entstehenden Teilbereiche werden anschließend wiederum rekursiv mit der gleichen Prozedur bearbeitet (Listing 1.26).

Listing 1.26
Beispiele\01_Grundlagen_VBA\01_04_More.xls\mdlSort

```
Private Function QuickSortArr( _
   varArray As Variant, _
   Optional ByVal lngLow As Long, _
   Optional ByVal lngUp As Long, _
   Optional blnDown As Boolean)

   Dim lngLowIndex   As Long
   Dim lngUpIndex    As Long
   Dim varElement    As Variant
   Dim strSort       As String
   Dim varBuffer     As Variant

   If Not IsArray(varArray) Then Exit Function

   If lngUp + lngLow = 0 Then
      ' Grenzen des Arrays bestimmen, wenn
      ' nichts übergeben wurde
      lngLow = LBound(varArray)
      lngUp = UBound(varArray)
   End If

   lngLowIndex = lngLow
   lngUpIndex = lngUp

   ' Ein Vergleichselement aus der Mitte der verbleibenden holen
   varElement = varArray((lngLowIndex + lngUpIndex) / 2)

   Do
      If blnDown Then

         ' Absteigend sortieren
         Do While varArray(lngLowIndex) > varElement
            ' So lange das Array von unten her
            ' durchlaufen, bis ein Element kleiner
            ' oder gleich dem Vergleichselement ist
            lngLowIndex = lngLowIndex + 1
         Loop
         Do While varArray(lngUpIndex) < varElement
            ' So lange das Array von oben her
            ' durchlaufen, bis ein Element größer
            ' oder gleich dem Vergleichselement ist
            lngUpIndex = lngUpIndex - 1
         Loop

      Else

         ' Aufsteigend sortieren
         Do While varArray(lngLowIndex) < varElement
            ' So lange das Array von unten her
            ' durchlaufen, bis ein Element größer
            ' oder gleich dem Vergleichselement ist
            lngLowIndex = lngLowIndex + 1
         Loop
```

Listing 1.26 (Forts.)
Beispiele\01_Grundlagen_VBA\01_04_More.xls\mdlSort

```
        Do While varArray(lngUpIndex) > varElement
            ' So lange das Array von oben her
            ' durchlaufen, bis ein Element kleiner
            ' oder gleich dem Vergleichselement ist
            lngUpIndex = lngUpIndex - 1
        Loop

    End If
    If lngLowIndex <= lngUpIndex Then

        'Die zwei gefundenen Elemente tauschen
        varBuffer = varArray(lngLowIndex)
        varArray(lngLowIndex) = varArray(lngUpIndex)
        varArray(lngUpIndex) = varBuffer

        ' Jeweils auf die nächsten Elemente zeigen
        ' Einmal von unten auf das nächste
        lngLowIndex = lngLowIndex + 1
        ' Von oben auf das nächste
        lngUpIndex = lngUpIndex - 1

    End If

' Noch einmal, wenn der untere Index kleiner
' oder gleich dem oberen ist
Loop Until (lngLowIndex > lngUpIndex)

If lngLow < lngUpIndex Then
    ' Der Zeiger, der von oben nach unten durchlaufen
    ' wird, ist noch größer als der untere, an die Prozedur
    ' übergebene. Wenn nichts übergeben wurde, ist er größer
    ' als die untere Grenze des Arrays. Noch einmal die
    ' Prozedur rekursiv aufrufen, mit angepasster oberer
    ' Grenze. Jetzt werden die Elemente zwischen den
    ' übergebenen Grenzen bearbeitet.
    QuickSortArr varArray, lngLow, lngUpIndex, blnDown
End If

If lngLowIndex < lngUp Then
    ' Der Zeiger, der von unten nach oben durchlaufen
    ' wird, ist noch kleiner als der obere, an die Prozedur
    ' übergebene. Wenn nichts übergeben wurde, ist er kleiner
    ' als die obere Grenze des Arrays. Noch einmal die
    ' Prozedur rekursiv aufrufen, mit angepasster unterer
    ' Grenze. Jetzt werden die Elemente zwischen den
    ' übergebenen Grenzen bearbeitet.
    QuickSortArr varArray, lngLowIndex, lngUp, blnDown
End If

End Function
```

Das Verfahren lässt sich noch etwas optimieren, dabei ist die Auswahl des Pivotelements entscheidend. Wenn das Pivotelement den größten oder kleinsten Wert enthält, müssen die meisten Vertauschungen vorgenommen werden. Also ist es am besten, ein Element zu benutzen, das dem mittleren Wert ziemlich nahe ist.

Eine Möglichkeit besteht darin, aus drei zufällig gewählten Elementen das günstigste als Pivotelement zu benutzen.

1.11.3 Mischen

Häufig benötigt man ein zufällig zusammengewürfeltes Datenfeld, sei es, um Bingo zu spielen, im Lotto zu gewinnen oder auch nur, um die verschiedenen Sortieralgorithmen zu testen. Dazu ist eine Prozedur zum Mischen von geordneten Datenfelder hervorragend geeignet.

VBA bietet, nachdem der Zufallsgenerator mit `Randomize` initialisiert worden ist, mit der `Rnd`-Funktion die Möglichkeit, eine zufällige Zahl zwischen null und eins zu erzeugen. Das allein nutzt nicht viel, denn man will ja ein geordnetes Datenfeld in Unordnung bringen, d.h. die Elemente zufällig verteilen.

Dazu wird ein zufälliger Index mit dem Maximum `n-1` erzeugt, der auf ein Element des Datenfeldes zeigt. Dieses Element tauschen Sie mit dem letzten Element des Datenfeldes. Danach erzeugen Sie eine Zufallszahl, die maximal den Wert der um eins reduzierten Anzahl der Datenfelder (`n-2`) entsprechen darf. Das Element mit diesem Zufallsindex wird dann mit dem Element `n-1` getauscht. Sie reduzieren also jedes Mal das mögliche Maximum der Zufallszahl um eins und tauschen dieses Element gegen das Element (`n+1-Anzahl der Durchläufe`) aus. Folgender Code (Listing 1.27) erledigt das:

Listing 1.27
Beispiele\01_Grundlagen_VBA\01_04_More.xls\mdlSort

```
Public Function Shuffle(varArray As Variant)
   Dim i             As Long
   Dim k             As Long
   Dim m             As Long
   Dim lngHigh       As Long
   Dim lngLow        As Long
   Dim varDummy      As Variant
   ' Array wird als Referenz übergeben, deshalb
   ' wird Original geändert

   If Not IsArray(varArray) Then Exit Function

   On Error Resume Next

   ' Zufallsgenerator initialisieren
   Randomize

   'Grenzen ermitteln
   lngHigh = UBound(varArray)
   lngLow = LBound(varArray)

   For i = lngLow To lngHigh

      ' Datenfeld mit allen Zahlen füllen
      ' Zufallszahl erzeugen
      k = Int((lngHigh - 1) * Rnd) + lngLow
```

```
        ' Das zufällig gewählte mit dem Element
        ' am Ende des noch zu mischenden
        ' Bereichs vertauschen
        varDummy = varArray(k)
        varArray(k) = varArray(lngHigh)
        varArray(lngHigh) = varDummy

        lngHigh = lngHigh - 1
        m = m + 1
    Next
    ' Anzahl gemischter Elemente zurückgeben
    Shuffle = m
End Function
```

Listing 1.27 (Forts.)
Beispiele\01_Grundlagen_VBA\01_04_More.xls\mdlSort

1.12 Bedingte Kompilierung

Auch VBA wird weiterentwickelt!

Wollen Sie Code schreiben, der versionsunabhängig ist, sollten Sie nur die Sprachsyntax verwenden, die unter dem kleinsten gemeinsamen Nenner auch existiert – und der ist momentan Excel 97. Benötigen Sie aber trotzdem Funktionen und Sprachelemente von späteren Versionen wie `Split` oder `AddressOf`, müssen Sie diese für den Einsatz unter Excel 97 mühsam nachbauen. Diese Nachbauten sind aber meistens erheblich langsamer als die Originale und sollten nur verwendet werden, wenn es unbedingt notwendig ist.

Man könnte jetzt auf die Idee kommen, die neue Sprachsyntax einzusetzen, zum Beispiel die Funktion `Split`, um dann auf einen Laufzeitfehler zu reagieren, indem man die Ersatzfunktion benutzt, wenn diese Funktion einen Fehler verursacht. Das funktioniert auch, solange Sie diese Mappe nicht unter Excel 97 einsetzen. Tun Sie dies doch, führt das zu einem Syntaxfehler beim Kompilieren und solch ein Fehler kann leider nicht abgefangen werden.

Es spricht aber gar nichts dagegen, mittels bedingter Kompilierung fehlende Sprachelemente in Excel 97 nachzubauen. Wenn Sie dann noch den gleichen Namen verwenden wie beispielsweise bei der Funktion `Split`, haben Sie wirklich portablen Code. Durch die bedingte Kompilierung können Sie auch größere Codeteile ganz von der Kompilierung ausschließen oder zulassen.

Das mit den gleichen Namen für eingebaute Sprachelemente funktioniert aber nicht in jedem Fall. `AddressOf` können Sie beispielsweise nicht ohne weiteres nachbauen. Dieser Operator ist nämlich keine Funktion, die einen Zeiger liefert. Außerdem wird der Funktionsname, dessen Zeiger benötigt wird, nicht als String übergeben.

Dies ist aber auch kein großes Problem, da Sie ganze Codeabschnitte von der Kompilierung ausschließen oder zulassen können. In diesen Abschnitten können Sie ja einmal mit dem Operator und einmal mit einer Funktion arbeiten. Solch einen Nachbau für `AddressOf` gibt es übrigens tatsächlich, dazu werden Funktionen der Bibliothek `vba332.dll` benutzt.

Und so setzen Sie die bedingte Kompilierung ein:

```
Private Const KonstanteXL97 = True

#If KonstanteXL97 = True Then
     ' Wenn XL97, dann
     VorherigeProcAdresse = SetWindowLong(FormHwnd1, _
         GWL_WNDPROC, GetFuncAddress("WindowProc"))
#Else
     VorherigeProcAdresse = SetWindowLong(FormHwnd1, _
         GWL_WNDPROC, AddressOf WindowProc)
#End If
```

Die Raute am Anfang der Zeile zeigt an, dass es sich um eine bedingte Kompilierung handelt. In der Bedingung hinter `If` kann nur ein konstanter Wert abgefragt werden, es kann also kein Berechnungsergebnis oder Ähnliches dazu benutzt werden.

In vielen Fällen benötigt man eine Konstante, die Auskunft über die Version gibt. Diese können Sie selber deklarieren, dann ist die bedingte Kompilierung aber nicht mehr so sinnvoll, weil Sie dann per Hand eingreifen und die Konstante selbst auf den entsprechenden Wert setzen müssen.

Es gibt aber einen Ausweg – und das ist die Konstante `VBA6`, die nur in Versionen größer Excel 97 existiert. Diese erscheint aber nicht im Objektkatalog und auch im Direktbereich der IDE mittels »? VBA6« erfolgt keine Ausgabe des Werts, selbst wenn die Konstante existiert. Dennoch klappt es damit bei der bedingten Kompilierung hervorragend. Hier nun der gleiche Code unter Verwendung dieser Konstante:

```
#If VBA6 = 0 Then
     ' Wenn XL97, dann
     VorherigeProcAdresse = SetWindowLong(FormHwnd1, _
         GWL_WNDPROC, GetFuncAddress("WindowProc"))
#Else
     VorherigeProcAdresse = SetWindowLong(FormHwnd1, _
         GWL_WNDPROC, AddressOf WindowProc)
#End If
```

2

Auflistungen und Collections

2.1 Was Sie in diesem Kapitel erwartet

In diesem Kapitel geht es darum, Ihnen zu zeigen, wie Sie eigene Auflistungen in Form von Collections erzeugen und verwalten. Die Vor- und Nachteile solcher selbst erzeugter Auflistungen werden dargestellt und es werden die Methoden beschrieben, mit denen Sie am besten über alle Elemente einer Auflistung iterieren, Elemente hinzufügen oder löschen.

Außerdem werden die zeitlichen Aspekte angesprochen, die darüber mitentscheiden, ob und wann der Einsatz einer Collection überhaupt sinnvoll ist. Zum Schluss wird noch eine praktische Einsatzmöglichkeit dargestellt.

2.2 Auflistungen

Auflistungen sind Sätze gleichartiger Objekte wie zum Beispiel Tabellenblätter, Zellen oder Arbeitsmappen. Sie haben mit Collections aber auch die Möglichkeit, selber solche Auflistungen anzulegen.

Erkennbar sind vorhandene Auflistungen meistens schon an ihren Namen. Sie besitzen ein Plural-s am Ende, das an den Namen der in ihnen enthaltenen Objekte gehängt ist. Ein Beispiel ist die Auflistung `Workbooks`, die einen Satz von Objekten des Typs `Workbook` enthält.

Jedes Element in einer solchen Auflistung ist unter einem eindeutigen Index ansprechbar. Dazu wird die `Item`-Eigenschaft der Auflistungen benutzt. Bei jeder Änderung der Auflistung kann sich aber die Reihenfolge und somit auch der Index eines bestimmten Elementes ändern. Deshalb ist es keine gute Idee, sich den Index eines bestimmten Elements zu merken, um später über diesen auf das Element zuzugreifen.

2.2.1 Durchlaufen von Auflistungen

Schleifen mit der `For Each...Next`-Anweisung sind extra dafür gemacht, nacheinander alle Elemente einer Auflistung oder eines Datenfeldes zurückzugeben. Der Aufbau solch einer Schleife ist folgender:

```
For Each Element In Gruppe
    ' Schleifenkörper
Next [Element]
```

`Element` ist dabei die Variable, die nacheinander eine Referenz oder den Wert jedes einzelnen Elementes der Auflistung erhält. Beim Iterieren durch Datenfelder kann als Variablentyp nur Variant gewählt, bei Auflistungen gleichartiger Objekte kann Variant, Object oder der spezielle Objekttyp benutzt werden.

Wenn eine Variantvariable für `Element` verwendet wird, bekommen Sie die Inhalte der Elemente als Wert (`ByVal`) zurückgeliefert. Bei Auflistungen von Objekten haben Sie die Wahl, ob Sie einen Variant, den Typ Object oder eine Objektvariable von dem Typ benutzen, der in der Auflistung enthalten ist. Verwenden Sie Objektvariablen des richtigen Typs, erhalten Sie eine Referenz auf das Element, mit der Konsequenz, dass Sie Zugriff auf das Originalobjekt bekommen.

Nachfolgend ein Beispiel (Listing 2.1), das zeigt, unter welchen Umständen die Variable `Element` eine Referenz und wann sie einen Wert zurückliefert.

Listing 2.1
Beispiele\02_Collections\02_01_Auflistungen.xls\mdlForEachByRefByVal

```
Public Sub ForEachByRefByVal()
   Dim avarElement   As Variant
   Dim rngRange      As Range
   Dim rngElement    As Range

   ' Objektvariable anlegen
   Set rngRange = Worksheets("For Each ByRef ByVal").Range("A7")
    'Initialisierungsstring Zelle A1
   rngRange = "Original"

   ' Mit For...Each und Variant > > Element als Wert
   For Each avarElement In rngRange

      avarElement = "Geändert" 'Element ändern

      ' Originalelement wird nicht geändert
      ' Originalelement und Variantvariable enthalten
      ' unterschiedliche Werte. Beide ausgeben!
      MsgBox "Element geändert auf : " & avarElement _
         & vbCrLf & _
         "Zelle A1 enthält : " & rngRange.Cells(1), , _
         "Element = Variant"
   Next

   ' Mit For...Each und passendem Objekttyp > > Element als Referenz
   For Each rngElement In rngRange

      rngElement = "Geändert"  ' Element ändern
```

```
        ' Originalelement wird geändert
        ' Originalelement und Objektvariable vom
        ' Typ Excel.Range enthalten gleiche
        ' Werte. Beide ausgeben!
        MsgBox "Element geändert auf : " & rngElement _
           & vbCrLf & _
           "Zelle A1 enthält : " & rngRange, , _
           "Element = Objektvariable vom Typ Excel.Range"
        ' Gleichwertig: rngRange.Cells,
        ' rngRange.Cells (1), rngRange(1), rngRange

    Next

End Sub
```

Listing 2.1 (Forts.)
Beispiele\02_Collections\
02_01_Auflistungen.xls\
mdlForEachByRefByVal

Da `Item` die Standardeigenschaft von Auflistungen ist, kann der Eigenschaftsname `Item` getrost weggelassen werden. Es können in dem vorhergehenden Beispiel (Listing 2.1) sogar noch mehr Standardeigenschaften weggelassen werden, sodass folgende Möglichkeiten gleichwertig sind und unter Umständen für Verwirrung sorgen können (die Objektvariable `objMyRange` ist diesem Fall ein `Range`-Objekt):

Wollen Sie alle Elemente einer Auflistungen durchlaufen, wenden Sie am besten die Schleife `For Each...Next` an. Das ist schneller, als die einzelnen Elemente über den Index anzusprechen.

- `objMyRange.Cells.Item(1)`

  Vollständige Referenzierung

- `objMyRange.Cells(1)`

  Hier wird die Standardeigenschaft `Item` der `Cells`-Auflistung weggelassen.

- `objRange(1)`

  Hier wird die Standardeigenschaft `Cells` des `Range`-Objekts weggelassen.

- `objRange`

  Für das erste Element können Sie sogar den Index weglassen.

Das Durchlaufen von Auflistungen mit der `For Each`-Anweisung ist nach meinen Tests um bis zu 40 Prozent schneller, als jedes einzelne Elemente per Index anzusprechen. Probieren Sie dazu einfach einmal folgendes Beispiel (Listing 2.2) aus:

```
Public Sub LoopList()
   Dim dtmBegin  As Date
   Dim dtmEnd    As Date
   Dim strTime1  As String
   Dim strTime2  As String
   Dim rngRange  As Range
   Dim rngCell   As Range
   Dim varDummy  As Variant
   Dim i         As Long

   ' Wenn im Dialog nicht OK angeklickt wurde, verlassen
   If MsgBox("Die Ausführung kann je nach Rechnerausstattung" & _
      " mehrere Minuten dauern", vbOKCancel, "Ausführungsdauer") _
      <> vbOK Then Exit Sub

   'Objektvariable anlegen
   Set rngRange = Worksheets( _
      "Durchlaufen von Auflistungen").Range("A:I")
```

Listing 2.2
Beispiele\02_Collections\
02_01_Auflistungen.xls\
mdlLoopList

Listing 2.2 (Forts.)
Beispiele\02_Collections\
02_01_Auflistungen.xls\
mdlLoopList

```
    ' Objektverweis mit With auf Spalte A bis I
    With rngRange

        ' Zeitbedarf mit For ... Next ermitteln
        ' Zeitpunkt Beginn der Schleife speichern
        dtmBegin = Time

        ' Alle Elemente der Cells-Auflistung des mit
        ' With referenzierten Objektes ansprechen
        For i = 1 To .Cells.Count
            'Wert über Index
            varDummy = .Cells(i)
        Next

        ' Zeitpunkt Ende der Schleife speichern
        dtmEnd = Time

        ' Zeitbedarf speichern
        strTime1 = "Zeit 'For Next' = " & _
            Format(dtmEnd - dtmBegin, "s ""Sekunden""")

        ' Zeitbedarf mit For ... Each ermitteln
        ' Zeitpunkt Beginn der Schleife speichern
        dtmBegin = Time

        ' Alle Elemente der Cells-Auflistung des mit
        ' With referenzierten Objektes ansprechen
        For Each rngCell In .Cells
            'Wert über Objektvariable
            varDummy = rngCell
        Next

        ' Zeitpunkt Ende der Schleife speichern
        dtmEnd = Time

        ' Zeitbedarf speichern
        strTime2 = "Zeit 'For Each' = " & _
            Format(dtmEnd - dtmBegin, "s ""Sekunden""")

        'Zeitbedarf ausgeben
        MsgBox strTime1 & vbCrLf & strTime2, , "Zeitvergleich"

    End With
End Sub
```

2.2.2 Löschen aus Auflistungen

Wollen Sie aus einer Auflistung alle Elemente löschen, können Sie in einer `For...Next`-Schleife nacheinander die einzelnen Elemente über ihren Index ansprechen und aus der Auflistung entfernen.

Durch das Löschen eines Elementes enthält die Liste anschließend aber ein Element weniger, d.h., die Zählvariable zeigt beim folgenden Durchlauf nicht mehr auf das nächste Element, sondern auf das übernächste. Wenn Sie an dieser

Stelle nicht durch eine Anpassung der Zählvariablen darauf reagieren, überspringen Sie ein Element.

Außerdem bekommen Sie, wenn die Elemente per Index angesprochen werden, gegen Ende der Schleife einen Laufzeitfehler, weil Sie irgendwann einen Index benutzen, der höher ist, als die Anzahl der noch darin vorhandenen Elemente.

Besser ist es in diesem Fall, das Pferd von hinten aufzuzäumen und die Liste von hinten nach vorne zu durchlaufen. Das Löschen eines Elementes ändert dann nicht die Reihenfolge und Anzahl der Elemente mit einem niedrigeren Index.

Wollen Sie in einer `For...Next`-Schleife Elemente aus einer Auflistung entfernen, durchlaufen Sie die Liste von hinten nach vorne. Dadurch stellen Sie sicher, dass kein Element übersprungen wird und vermeiden den Laufzeitfehler 9 »Index außerhalb des gültigen Bereichs«.

Das folgende Beispiel (Listing 2.3) veranschaulicht die Problematik, die sich aus dem Löschen von Elementen unter Einsatz von Schleifen ergeben kann. Als Liste zum Löschen wird eine Collection benutzt, aber das Gleiche gilt selbstverständlich auch für jede andere Auflistung wie zum Beispiel `Worksheets`, `Workbooks`, `Shapes` und andere.

Listing 2.3
Beispiele\02_Collections\
02_01_Auflistungen.xls\mdlErase

```
Public Sub DelFromList()
   Dim colSource1 As New Collection
   Dim colSource2 As New Collection
   Dim i          As Long
   Dim k          As Long
   Dim strMsg     As String

   ' Fehlerbehandlung einrichten
   On Error Resume Next

   For i = 1 To 6
      ' Zwei Collections mit je 6 Elementen erzeugen
      ' um Auflistungen zur Demonstration zu haben
      colSource1.Add "Inhalt " & i, "Item" & i
      colSource2.Add "Inhalt " & i, "Item" & i

      ' String mit dem Inhalt der Collection erzeugen
      strMsg = strMsg & "Inhalt " & i & vbCrLf
   Next

   ' Ausgabe des Collectioninhalts
   MsgBox "Die zwei neuen Collections enthalten jeweils " & _
      colSource1.Count & " Elemente" & vbCrLf & strMsg

   ' Aufsteigend löschen
   For i = 1 To colSource1.Count

      ' Ausgabestring und Fehlerspeicher löschen
      strMsg = ""
      Err.Clear

      ' Element mit dem Index i löschen
      colSource1.Remove i

      If Err.Number <> 0 Then
         ' Ein Fehler ist aufgetreten
         strMsg = "Fehler beim Löschen" & vbCrLf
      End If
```

Listing 2.3 (Forts.)
Beispiele\02_Collections\
02_01_Auflistungen.xls\mdlErase

```
        ' Info über den aktuellen Zählerstand holen
        strMsg = strMsg & "Zählvariable bei : " & i & vbCrLf

        ' Zusammenstellen des Collectioninhalts
        For k = 1 To colSource1.Count
           strMsg = strMsg & colSource1(k) & vbCrLf
        Next

        ' Ausgabe des aktuellen Zustandes
        MsgBox strMsg, , "Inhalt nach dem " & i & _
           ". aufsteigenden Löschen"

    Next

    ' Ergebnisausgabe aufsteigendes Löschen
    strMsg = "Die Collection enthält nach" & _
       " dem aufsteigenden Löschen " & _
       colSource1.Count & " Elemente"
    MsgBox strMsg

    ' Absteigend löschen
    For i = colSource2.Count To 1 Step -1

        ' Ausgabestring löschen
        strMsg = ""

        ' Element mit dem Index i löschen
        colSource2.Remove i

        ' Info über den aktuellen Zählerstand holen
        strMsg = strMsg & "Zählvariable bei : " & i & vbCrLf

        ' Zusammenstellen des Collectioninhalts
        For k = 1 To colSource2.Count
           strMsg = strMsg & colSource2(k) & vbCrLf
        Next

        ' Ausgabe des aktuellen Zustandes
        MsgBox strMsg, , "Inhalt nach dem " & 7 - i & _
           ". absteigenden Löschen"

    Next

    'Ergebnisausgabe aufsteigendes Löschen
    strMsg = "Die Collection enthält nach" & _
       " dem absteigenden Löschen " & _
       colSource2.Count & " Elemente"
    MsgBox strMsg
End Sub
```

2.3 Collections

VBA bietet die Möglichkeit, eigene Auflistungen zu erzeugen. Ein wenig bekanntes, immer wieder unterschätztes und wahrscheinlich deswegen selten eingesetztes Objekt ist die `Collection`. Das ist wirklich sehr schade, denn die Vorteile sind nicht zu unterschätzen. Im Gegensatz zu einem Datenfeld ist das Durchlaufen einer `Collection` zwar langsam, das Collection-Objekt macht aber den Nachteil in anderer Beziehung wieder wett.

2.3.1 Vorteile von Collections

Keine Beschränkung auf einen Datentyp

Bei Collections sind Sie nicht an einen einzelnen Datentyp gebunden. Sie können jedem Element einen anderen Datentyp geben, Objektverweise sind möglich und sogar neue Collections können als Element benutzt werden. Mit diesen Collections in Collections lassen sich ganz hervorragend Baumstrukturen abbilden.

Speichern von Objekten möglich

Es ist möglich, in Collections Objekte zu speichern. Damit lassen sich beispielsweise Instanzen von Klassen am Leben erhalten. Das kann notwendig werden, wenn Sie dynamische Steuerelemente verwalten und die Ereignisse dieser Steuerelemente benötigen. Mehr dazu erfahren Sie in Kapitel 3.

Ansprechen der Elemente über Namen

Jedes Element einer Collection kann direkt über den Index oder über seinen eindeutigen Schlüsselnamen angesprochen werden. Haben Sie in einer Collection beispielsweise Kundennummern für die Schlüsselnamen benutzt, können Sie über diese direkt auf das Element zugreifen, ohne durch alle Elemente eines Datenfeldes zu iterieren.

Unkompliziertes Löschen und Einfügen von Elementen

An jeder Stelle der Collection lassen sich Elemente ohne das Verschieben von anderen, bereits enthaltenen Elementen einfügen oder herauslöschen. Wollen Sie dagegen bei einem Datenfeld ein Element komplett löschen, müssen Sie sich erst einmal Gedanken darüber machen, was überhaupt an die Stelle des gelöschten Elements kommen soll.

Löschen Sie nur den Wert selbst, indem Sie beispielsweise eine numerische Variable auf null setzen, existiert das Element mit dem zugehörigen Index weiterhin und der dafür reservierte Speicherplatz innerhalb des Datenfeldes ist immer noch belegt. Um das Element tatsächlich zu entfernen, haben Sie zwei Möglichkeiten.

1. Das Element an der zu löschenden Position bekommt den Wert des letzten Elementes zugewiesen und danach reduzieren Sie mit `ReDim Preserve` die Anzahl der Elemente um eins. Das funktioniert aber nur bei dynamischen Datenfeldern und anschließend passt die ursprüngliche Reihenfolge der Elemente nicht mehr.
2. Sie verschieben alle Elemente hinter dem zu löschenden um je ein Element nach vorne und reduzieren mit `ReDim Preserve` die Anzahl der Elemente um eins. Jetzt bleibt zwar die Reihenfolge erhalten, das Verschieben selbst kostet aber viel Zeit.

Wollen Sie bei Datenfeldern ein Element einfügen, müssen Sie erst einmal die Anzahl der Elemente des Datenfeldes um eins erhöhen, damit für das neue Element Platz vorhanden ist. Für das weitere Vorgehen gibt es wie beim Löschen von Elementen wiederum zwei Möglichkeiten.

1. Sie kopieren den Elementinhalt, der sich an der einzufügenden Stelle befindet, an die letzte Position. Dann kann an die freigewordene Stelle der neue Elementinhalt eingefügt werden. Anschließend passt aber die ursprüngliche Reihenfolge der Elemente nicht mehr.
2. Sie verschieben alle Elemente ab der Position des neuen Elements um je eins nach hinten und können den neuen Elementinhalt an die freigewordene Stelle einfügen. Die Reihenfolge bleibt zwar dadurch erhalten, aber die Nachteile sind die gleichen wie beim Löschen von Elementen, das Verschieben der Elemente erfordert viel Zeit.

Bei einer Collection läuft das Ganze dagegen vollkommen unkompliziert ab. Sie löschen oder fügen ein Element einfach an jeder beliebigen Stelle ein. Beim Einfügen gibt es bei der Methode `Add` noch die optionalen Parameter `before` und `after`, die festlegen, an welcher Stelle das neue Element eingefügt werden soll. Wenn Sie den optionalen Parameter für die Position weglassen, wird das neue Element am Ende eingefügt.

```
' An das Ende
coltest.Add "Wert1", "Key1"

' Vor das erste Element
coltest.Add "Wert2", "Key2", before:=1

' Vor das Element mit dem Schlüssel "Key1"
coltest.Add "Wert3", "Key3", before:="Key1"

' Hinter das dritte Element
coltest.Add "Wert4", "Key4", after:=3

' Hinter das Element mit dem Schlüssel "Key3"
coltest.Add "Wert5", "Key5", after:="Key3"
```

Wenn Sie in einem Datenfeld ein bestimmtes Element suchen, müssen Sie im schlimmsten Fall das komplette Datenfeld durchlaufen und jeden Eintrag mit dem Suchkriterium vergleichen.

Alle Elemente von Collections besitzen Schlüsselnamen, über die Sie jedes einzelne Element direkt ansprechen können, und zwar ohne vorher die Liste zu

durchlaufen. Sie müssen nur einen eindeutigen und einzigartigen Schlüsselnamen für jedes Element benutzen.

Das kann ein zusammengesetzter String als Kombination von Vorname, Nachname und Geburtstag sein oder eine eindeutige Kundennummer. Selbstverständlich muss intern das gewünschte Element auch aus einer Liste herausgesucht werden, davon merken Sie aber überhaupt nichts.

2.3.2 Nachteile von Collections

Collections verbrauchen viel Speicherplatz

Durch den Umstand, dass Collections so flexibel sind, ist der Verwaltungsaufwand auch weitaus höher als beispielsweise bei einem Array vom Typ `Long`. Die zusätzlich benötigten Informationen schlagen auch mit einem höheren Speicherbedarf zu Buche.

Das Durchlaufen von Collections ist langsam

Vergleicht man die benötigte Zeit zum Durchlaufen von Arrays mit dem Durchlaufen von Auflistungen und Collections, stellt man einen gewaltigen Zeitunterschied fest. Bei einem Array läuft das erheblich schneller ab.

Das Löschen und Einfügen von Elementen ist langsam

Der höhere interne Verwaltungsaufwand sorgt auch hier für Zeitverluste.

Der Elementinhalt kann nachträglich nicht geändert werden

Der größte Nachteil ist meiner Ansicht nach der, dass einmal erstellte Elemente einer Collection nachträglich nicht mehr geändert werden können. Stattdessen müssen Sie das Element löschen und mit den geänderten Werten wieder einfügen.

Der Schlüsselname kann nicht ausgelesen werden

Leider kann auch der Schlüssel nicht ausgelesen werden. Das wäre hilfreich, wenn man mit der `For Each`-Schleife nacheinander alle Elemente anspricht.

2.3.3 Collections anlegen

Um eine `Collection` zu erzeugen, muss diese wie andere Objekte angelegt werden. Das können Sie folgendermaßen erledigen:

1. Verwenden der Dim-Anweisung zusammen mit New

   ```
   Dim myCol As New Collection
   ```

2. Mit der Dim-Anweisung eine Objektvariable anlegen, mit »`Set As New`« ein Objekt erstellen und gleichzeitig der Objektvariablen zuweisen

   ```
   Dim myCol As Collection
   Set myCol = New Collection
   ```

Der Unterschied zwischen beiden Varianten ist der Zeitpunkt, ab dem das Objekt tatsächlich existiert. Um diesen Sachverhalt zu demonstrieren, wird im folgenden Beispiel (Listing 2.4) als Objekt eine Klasse statt einer Collection benutzt, das Prinzip ist aber das gleiche.

Klassen haben aber für die Demonstration des Geburtszeitpunkts den Vorteil, dass sie über die Ereignisprozedur `Class_Initialize` verfügen, die beim Initialisieren der Klasse abgearbeitet wird. Das ist der Zeitpunkt, an dem das Objekt tatsächlich zum Leben erweckt wird.

Erstellen Sie eine neue Klasse mit dem Namen `clsGeburt` und fügen Sie den folgenden Code in das Klassenmodul ein:

Listing 2.4
Beispiele\02_Collections\
02_01_Auflistungen.xls\Tabelle1

```
Private Sub Class_Initialize()

    ' Ereignis Initialisierung (Erstellungszeitpunkt)
   MsgBox "Initialisierung der Klasse läuft", , " clsBirthday "

End Sub

Public Property Get FirstAccess() As String

   FirstAccess = "Zugriff"
   MsgBox " Zugriff auf eine Eigenschaft", , " clsBirthday "

End Property
```

Als Ereignisprozedur `Click` eines Buttons mit dem Namen `cmdSetNew` folgender Code:

```
Private Sub cmdSetNew_Click()
   Dim strRet     As String
   Dim objClass   As clsBirthday

   MsgBox "Impliziertes Erstellen einer Instanz mit" & _
      vbCrLf & "Set objClass = New clsBirthday", , _
      "Dim objClass As clsGeburt"

   ' Objekt an Objektvariable zuweisen
   Set objClass = New clsGeburt

   MsgBox "Erster Zugriff auf das Objekt mit" & _
      vbCrLf & "strRet = objClass.FirstAccess", , _
      "Dim objClass As clsBirthday"

   ' Auf eine Eigenschaft des Objekts zugreifen
   strRet = objClass.FirstAccess
End Sub
```

Als Ereignisprozedur `Click` eines Buttons mit dem Namen `cmdDimNew` folgender Code:

```
Private Sub cmdDimNew_Click()
   Dim strRet     As String
   Dim objClass   As New clsBirthday

   MsgBox "Erster Zugriff auf das Objekt mit" & _
      vbCrLf & "strRet = objClass.FirstAccess", , _
      "Dim objClass As New clsBirthday"

   ' Auf eine Eigenschaft des Objekts zugreifen
   strRet = objClass.FirstAccess
End Sub
```

Nach einem Klick auf den Button `cmdDimNew` werden Sie erkennen, dass die Initialisierung der Klasse `objClass` durch die Zeile `Set objClass = New clsGeburt` angestoßen wird.

Wenn das Schlüsselwort New zusammen mit Set verwendet wird, erstellen Sie implizit eine neue Instanz des Objektes. Wenn das Schlüsselwort New dagegen bereits bei der Deklaration der Objektvariablen verwendet wird, wird eine neue Instanz des Objekts erst aufgrund des ersten Verweises darauf erstellt.

Klicken Sie dagegen auf den Button `cmdSetNew`, stellen Sie fest, dass eine Instanz der Klasse `objClass` tatsächlich erst durch den ersten Zugriff auf die Klasse erstellt wird und nicht, wie man annehmen könnte, bereits durch die Deklaration mit `Dim objClass As New clsGeburt`.

In den meisten Fällen ist es besser, selber zu bestimmen, ab wann ein Objekt existiert, als sich auf den ersten Zugriff zu verlassen, dessen Zeitpunkt möglicherweise noch variieren kann. Ist es dagegen häufiger der Fall, dass das Objekt überhaupt nicht verwendet wird, können Sie sich bei Nichtgebrauch die Initialisierung des Objekts sparen. In diesem Fall deklarieren Sie einfach mit `New` oder weisen der Objektvariablen erst bei Bedarf das Objekt zu.

2.3.4 Bingo mit Collection

Collections sind hervorragend dazu geeignet, doppelte Eingaben oder Werte zu verhindern. Dazu benötigen Sie die Fehlerbehandlungsroutine `On Error Resume Next` und zusätzlich ein `Collection-Objekt`.

Anschließend fügen Sie ganz einfach jeden Wert mit der Methode `Add` in die Collection ein, und zwar so, dass dieser Wert als Schlüsselname benutzt wird. Da Schlüsselnamen von Collections Strings sein müssen, setzt man den Namen so zusammen, dass vor diesem Wert eine selbst festgelegte Zeichenfolge steht. Bisher hatte ich aber auch mit reinen Zahlen keine Probleme, die automatische Typenumwandlung sorgt dann schon für einen String.

Tritt nun beim Einfügen eines neuen Elementes ein Laufzeitfehler auf, weil der Schlüsselname bereits existiert, hat man es mit einem doppelten Eintrag zu tun. Ob solch ein Fehler aufgetreten ist, erkennen Sie, wenn Sie die `Number`-Eigenschaft des `Err`-Objektes auslesen, die bei einem Fehler ungleich null (0) ist. Vor dem Einfügen in die Collection sollte deshalb die Zeile `Err.Clear` stehen. Dadurch werden alle Einstellungen des `Err`-Objekts aus dem Speicher gelöscht und man kann sicher sein, dass der Fehler auch wirklich von dieser Aktion stammt.

Im folgenden Beispiel (Listing 2.5) wird eine `Collection` benutzt, um doppelte Zufallszahlen zu verhindern. Wie schon angesprochen sind Collections ziemlich langsam. Bei der Ausführung des nachfolgenden Programms zur Ziehung von zehntausend Zahlen aus einer Menge von zehntausend Zahlen mit einem AMD Mobile Athlon 4 1200 MHz liegt der Zeitbedarf bei 21 Sekunden. Wenn relativ wenig Zahlen aus einer großen Menge gebraucht werden, spielt das aber keine große Rolle:

Listing 2.5
Beispiele\02_Collections\
02_01_Auflistungen.xls\mdlBingo

```
Public Sub Bingo1()
   Dim colBingo   As New Collection
   Dim i          As Long
   Dim k          As Long
   Dim strMessage As String
   Dim dtmTime    As Date
   Dim lngFrom    As Long
   Dim lngItems   As Long

   ' Fehler ignorieren
   On Error Resume Next

   ' Zufallsgenerator initialisieren
   Randomize

   ' Zehntausend Zahlen aus zehntausend Zahlen ziehen
   lngFrom = 10000
   lngItems = 10000

   dtmTime = Now ' Zeitpunkt Schleifenbeginn

   ' Schleife durchlaufen. 1 bis Anzahl der gewünschten Zahlen
   For i = 1 To lngItems

      Do
         ' Zufallszahl erzeugen
         k = Int(lngFrom * Rnd) + 1

         ' Error-Objekt löschen
         Err.Clear

         ' Versuchen, gezogene Zahl in die Collection einzufügen
         colBingo.Add k, "Bingo" & k

      ' Wenn Fehler aufgetreten ist, noch mal probieren
      Loop While Err.Number <> 0

      ' Gezogene Zahl in einem String speichern
      strMessage = strMessage & colBingo(i) & vbCrLf

   Next

   ' Hinzufügen des Zeitbedarfs zu den gezogenen Zahlen
   strMessage = "Zeit : " & Format(Now - dtmTime, "nn:ss") & _
      vbCrLf & strMessage

   ' Ausgabe
   MsgBox Left(strMessage, Len(strMessage) - 2)
End Sub
```

2.3.5 Bingo mit Datenpool

Bei vielen Zahlen aus einer fast gleich großen Menge Zahlen wird die Sache schon etwas schlechter, weil man zum Ende hin immer öfter eine schon gewählte Zufallszahl generiert. Sie müssen dann jedes Mal so lange neue Zufallszahlen erzeugen, bis Sie eine finden, die noch nicht gewählt wurde.

Dem können Sie abhelfen, indem Sie eine `Collection` mit allen verfügbaren Zahlen füllen und sich aus diesem Zahlenpool bedienen. Das zufällig gewählte Element wird dann aus der `Collection` entfernt und steht somit nicht mehr zur Verfügung.

Diese Variante hat gegenüber der ersten zudem noch den großen Vorteil, dass der Zeitbedarf linear zu der Anzahl der zu ziehenden Zahlen steigt und damit berechenbar wird. Bei der Ausführung des nachfolgenden Programms (Listing 2.6) zur Ziehung von zehntausend Zahlen aus einer Menge von zehntausend Zahlen mit einem AMD Mobile Athlon 4 1200 MHz liegt der Zeitbedarf bei 16 Sekunden. Das ist ein Zeitvorteil von fast 25% gegenüber der ersten Methode (Listing 2.5).

Listing 2.6
Beispiele\02_Collections\
02_01_Auflistungen.xls\mdlBingo

```
Public Sub Bingo2()
   Dim colBingo   As New Collection
   Dim i          As Long
   Dim k          As Long
   Dim strMessage As String
   Dim dtmTime    As Date
   Dim lngFrom    As Long
   Dim lngItems   As Long

   ' Zufallsgenerator initialisieren
   Randomize

   ' Zehntausend Zahlen aus zehntausend Zahlen ziehen
   lngFrom = 10000
   lngItems = 10000

   dtmTime = Now ' Zeitpunkt Schleifenbeginn

   For i = 1 To lngFrom
      ' Collection mit allen Zahlen füllen
      colBingo.Add i, "Lotto" & i
   Next

   ' Schleife durchlaufen. 1 bis Anzahl der gewünschten Zahlen
   For i = 1 To lngItems

      ' Zufallszahl erzeugen
      k = Int((lngFrom) * Rnd) + 1

      ' Das Element mit dem zufällig gewählten Index auswählen
      ' und Zahl in einem String speichern
      strMessage = strMessage & colBingo(k) & vbCrLf

      ' Anschließend aus der Collection entfernen
      colBingo.Remove k
```

Listing 2.6 (Forts.)
Beispiele\02_Collections\
02_01_Auflistungen.xls\mdlBingo

```
        ' Menge der verfügbaren Zahlen anpassen
        lngFrom = lngFrom - 1

    Next

    ' Hinzufügen des Zeitbedarfs zu den gezogenen Zahlen
    strMessage = "Zeit : " & Format(Now - dtmTime, "nn:ss") & _
        vbCrLf & strMessage

    ' Ausgabe
    MsgBox Left(strMessage, Len(strMessage) - 2)
End Sub
```

Der doch etwas magere Zeitgewinn liegt daran, dass das Hinzufügen und Entfernen von Elementen in einer `Collection` recht lange dauert.

2.3.6 Bingo mit Datenfeld

Wie lange das Hinzufügen und Entfernen von Elementen in und aus einer `Collection` wirklich dauert, kann man erst richtig ermessen, wenn man das zweite Beispiel umschreibt (Listing 2.7) und anstatt Collections normale Datenfelder benutzt. Der Zeitbedarf bei dem gleichen System beträgt nur noch ca. 3 Sekunden, das ist etwa ein Fünftel der Zeit wie in Listing 2.6.

Listing 2.7
Beispiele\02_Collections\
02_01_Auflistungen.xls\mdlBingo

```
Public Sub Bingo3()
    Dim alngBingo()   As Long
    Dim i             As Long
    Dim k             As Long
    Dim strMessage    As String
    Dim dtmTime       As Date
    Dim lngFrom      As Long
    Dim lngItems     As Long
    ' AMD Mobile Athlon 4 1200 MHz Zeitbedarf: 3 Sekunden


    ' Zufallsgenerator initialisieren
    Randomize

    ' Zehntausend Zahlen aus zehntausend Zahlen ziehen
    lngFrom = 10000
    lngItems = 10000

    ' Datenfeld in der gewünschten Größe anlegen
    ReDim alngBingo(1 To lngFrom)

    dtmTime = Now ' Zeitpunkt Schleifenbeginn

    For i = 1 To lngFrom
        ' Datenfeld mit allen Zahlen füllen
        alngBingo(i) = i
    Next

    ' Schleife durchlaufen. 1 bis Anzahl der gewünschten Zahlen
    For i = 1 To lngItems
```

Listing 2.7 (Forts.)
Beispiele\02_Collections\
02_01_Auflistungen.xls\mdlBingo

```
        ' Zufallszahl erzeugen
        k = Int((lngFrom) * Rnd) + 1

        ' Das Element mit dem zufällig gewählten Index auswählen
        strMessage = strMessage & alngBingo(k) & vbCrLf

        ' Das letzte nicht gewählte Element an die Stelle
        ' des gewählten verschieben
        If lngFrom > 0 Then alngBingo(k) = alngBingo(lngFrom)

        ' Menge der verfügbaren Zahlen anpassen
        lngFrom = lngFrom - 1

    Next

    ' Hinzufügen des Zeitbedarfs zu den gezogenen Zahlen
    strMessage = "Zeit : " & Format(Now - dtmTime, "nn:ss") & _
        vbCrLf & strMessage

    ' Ausgabe
    MsgBox Left(strMessage, Len(strMessage) - 2)
End Sub
```

Ein paar kleine Erläuterungen zu dem vorhergehenden Codebeispiel (Listing 2.7):

Zu Beginn erstellen Sie ein Datenfeld in solch einer Größe, dass alle Ausgangszahlen hineinpassen. Mit der Zahlenmenge, aus denen die Zufallszahlen gezogen werden sollen, wird das Datenfeld dann gefüllt. Die Reihenfolge der Zahlen im Datenfeld spielt dabei absolut keine Rolle.

Nun ermitteln Sie unter Zuhilfenahme der `Rnd`-Funktion eine Zufallszahl zwischen 1 und der Anzahl der verfügbaren Zahlen. `Randomize` sollten Sie zum Initialisieren des Zufallsgenerators aber nicht vergessen.

Aus dem Feld mit dem zufällig gewählten Index holen Sie sich nun den Wert als die eigentliche Zufallszahl. Der Wert in diesem Datenfeld wird anschließend nicht einfach gelöscht. Man kopiert sich eine noch nicht gewählte Zahl in dieses Feld. Dazu benutzen Sie das letzte Feld in der verfügbaren Menge von Datenfeldern und reduzieren die Anzahl der verfügbaren Felder um eins.

Nun stehen n-1 Datenfelder mit noch nicht gewählten Zahlen zur Verfügung. Für die zweite Zufallszahl ermitteln Sie eine Zufallszahl im Bereich von 1 bis n-1 und wiederholen das ganze Spiel. Das können Sie so lange fortführen, wie Elemente vorhanden sind.

2.3.7 Beispiel Änderungen markieren

Hier noch ein kleines Beispiel für den Einsatz von Collections. Damit werden die Adressen der geänderten Zellen eines Tabellenblattes als neue Collection in einer bestehenden Collection gespeichert. In dieser eingefügten Collection können Sie beliebige Informationen speichern, beispielsweise die Farbe des Zellhintergrundes oder des Textes. Was Sie darin speichern, bleibt aber letztendlich Ihnen überlassen.

Der Code gehört in das Klassenmodul eines Tabellenblattes, zwei Buttons mit den Namen `cmdMark` und `cmdUnmark` sollten vorhanden sein.

Im Change-Ereignis des Tabellenblatts wird jede geänderte Zelle als ein Element vom Typ Collection in der klassenweit gültigen Collection `mcolChanged` gespeichert. Als eindeutiger Schlüsselname wird die Adresse der Zelle benutzt.

In diese neue Collection wird dann ein Element mit dem Schlüsselnamen »Address« und der Adresse als Wert angelegt. Das wird gemacht, da man zwar ein Element über den Schlüsselnamen ansprechen, diesen Schlüsselnamen bedauerlicherweise aber nicht auslesen kann. Als Nächstes legen Sie die Elemente »Color« und »FontColor« an, und speichern darin die entsprechenden Zelleigenschaften als Wert.

Im Klickereignis des Buttons `cmdMark` werden nun alle Elemente der Collection nacheinander durchlaufen. Dabei werden die aktuellen Eigenschaften der Zelle, auf die sich das Element bezieht, in einer neuen Collection gespeichert. Das Element wird anschließend aus der Collection entfernt und dafür das neu angelegte mit den aktuellen Informationen an der ersten Position gespeichert. Jetzt können Sie die Zelle beliebig formatieren, die Informationen über das Aussehen vor der Änderung sind schließlich gespeichert.

Um zu verhindern, dass Formate gespeichert werden, die nach einer Markierung herrschen, wird die boolesche Variable `mblnIsMarked` auf Klassenebene mitgeführt, die nach der Markierung auf »Wahr« gesetzt wird. Ist diese Variable wahr, wird die Prozedur `cmdMark_Click` unverzüglich verlassen.

Ein Klick auf den Button `cmdUnmark` stellt den ursprünglichen Zustand wieder her. Alle gespeicherten Zellen werden in den Zustand vor der Markierung versetzt. Auch die Variable `mblnIsMarked` wird wieder auf »Falsch« gesetzt.

Listing 2.8
Beispiele\02_Collections\02_01_Auflistungen.xls\Tabelle8

```
Private mcolChanged  As New Collection
Private mblnIsMarked  As Boolean

Private Sub cmdMark_Click()
   Dim varMark    As Variant
   Dim colCell    As Collection
   Dim strAddress As String

   ' Beenden, wenn schon markiert wurde
   If mblnIsMarked Then Exit Sub

   ' Kennzeichnen, dass bereits markiert wurde
   mblnIsMarked = True

   ' Alle Elemente der Collection durchlaufen
   For Each varMark In mcolChanged

      ' Neue Kollektion anlegen
      Set colCell = New Collection

      ' Aktuelle Adresse holen
      strAddress = varMark("Address")
```

Listing 2.8 (Forts.)
Beispiele\02_Collections\
02_01_Auflistungen.xls\Tabelle8

```
      With Me.Range(strAddress) ' Aktuelle Zelle

         ' Neue Collection mit den aktuellen Zellformaten anlegen
         colCell.Add strAddress, "Address"
         colCell.Add .Interior.ColorIndex, "Color"
         colCell.Add .Font.ColorIndex, "FontColor"

         ' Element löschen,
         mcolChanged.Remove strAddress

         ' um die letzte Formatierung zu speichern
         If mcolChanged.Count = 0 Then
            mcolChanged.Add colCell, strAddress
         Else
            mcolChanged.Add colCell, strAddress, before:=1
         End If
         ' Jede geänderte Zelle Rot färben
         .Interior.ColorIndex = 3

         ' Und Textfarbe auf Schwarz
         .Font.ColorIndex = 1

      End With
   Next
End Sub

Private Sub cmdUnmark_Click()
   Dim varMark As Variant

   'Kennzeichnen, dass zurückgesetzt wurde
   mblnIsMarked = False

   'Alle Elemente der Collection durchlaufen
   For Each varMark In mcolChanged

      With Me.Range(varMark("Adresse"))

         ' Die Farben zurücksetzen
         .Interior.ColorIndex = varMark("Color")
         .Font.ColorIndex = varMark("FontColor")

      End With

   Next
End Sub

Private Sub Worksheet_Change(ByVal Target As Range)
   Dim colCell As Collection
   Dim rngCell As Range

   On Error Resume Next

   ' Alle Zellen des geänderten Bereichs durchlaufen
   For Each rngCell In Target

      ' Neue Kollektion anlegen
      Set colCell = New Collection
```

Listing 2.8 (Forts.)
Beispiele\02_Collections\
02_01_Auflistungen.xls\Tabelle8

```
        ' Fehlerobjekt zurücksetzen
        Err.Clear

        ' Für jede Zelle eine eigene Kollektion
        mcolChanged.Add colCell, rngCell.Address
        ' Wenn Element vorhanden, Fehlernummer <> 0
        If Err.Number <> 0 Then

            ' Wenn Zelle schon geändert wurde, Element löschen,
            mcolChanged.Remove rngCell.Address

            ' Neues Element, um letzte Änderung zu speichern
            mcolChanged.Add colCell, rngCell.Address

        End If

        With mcolChanged (rngCell.Address)

            ' Adresse als Element speichern
            .Add rngCell.Address, "Address"

            ' Hintergrundfarbe als Element speichern
            .Add rngCell.Interior.ColorIndex, "Color"

            ' Hintergrundfarbe als Element "FontColor" speichern
            .Add rngCell.Font.ColorIndex, "FontColor"

        End With
    Next
End Sub
```

2.3.8 Fazit

Collections können das Leben eindeutig erleichtern. Es lässt sich komfortabel damit arbeiten, der Verwaltungsaufwand wie bei Datenfeldern fällt weg und Sie haben die Möglichkeit, ein Element direkt über einen eindeutigen Namen anzusprechen. Den Zeitnachteil kann man in den meisten Fällen verschmerzen.

Benutzen Sie also Collections, wenn es auf die Zeit nicht ankommt. Sie können dadurch leichter lesbaren Code schreiben und viele Fehler, die hinter dem Löschen und Einfügen von Elementen in Datenfeldern stecken, können dadurch gar nicht erst auftreten.

Da es möglich ist, in Collections sehr einfach Objekte zu speichern, lassen sich damit Instanzen von Klassen am Leben erhalten. Das kann für die Ereignisverwaltung von dynamischen Steuerelementen sehr hilfreich sein, wie Sie im nächsten Kapitel sehen werden.

3

Klassen

3.1 Was Sie in diesem Kapitel erwartet

In diesem Kapitel geht es darum, zu zeigen, was Klassen sind und warum Sie diese viel häufiger einsetzen sollten.

Ein kleines Beispiel zur Manipulation von Initialisierungsdateien zeigt die Vorgehensweise, mit der Sie möglichst unkompliziert und ohne umfangreiche Planung kleinere Klassen anlegen können.

Außerdem wird gezeigt, wie Sie Steuerelemente dynamisch in eine Userform und ein Tabellenblatt einfügen und mit den Ereignissen dieser Steuerelemente umgehen. Dazu sind Klassen nämlich unerlässlich.

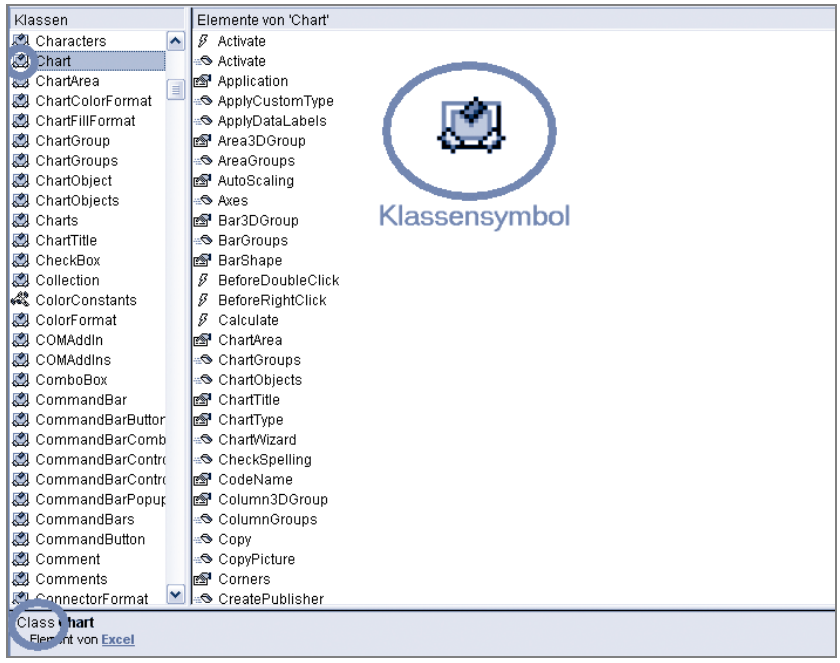
3.2 Allgemeines

Klassen bilden die Grundlage der objektorientierten Programmierung. Sie kapseln die Funktionalität von Objekten, machen den Code übersichtlicher und man kann den übergeordneten Programmablauf besser verstehen.

Nahezu alle Objekte, die Sie in Excel benutzen, beruhen auf Klassen. Wenn Sie den Objektkatalog öffnen (Abbildung 3.1), werden Sie hunderte verschiedener Objektklassen vorfinden.

Wenn Sie Codeteile besitzen, die Sie immer wieder benutzen, sollten Sie sich auf jeden Fall dazu entschließen, dafür eine eigene Klasse anzulegen, auch wenn es Ihnen anfangs etwas umständlich vorkommt. Die Vorteile sind enorm und je häufiger Sie selbst Klassen schreiben und einsetzen, umso mehr werden auch Sie davon überzeugt sein.

Abbildung 3.1
Klassen im Objektkatalog



Der wohl größte Vorteil ist der, dass Sie mit einer Klasse ein kompaktes, wiederverwendbares Objekt mit Eigenschaften und Methoden besitzen und beim Einsatz dieser Klasse nicht mehr wissen müssen, wie die Funktionalität implementiert ist. Es reicht, wenn Sie deren Eigenschaften, Methoden und Funktionen kennen.

Arbeiten Sie im Team, müssen Sie gemeinsam lediglich die öffentlichen Eigenschaften und Methoden der Klassen festlegen. Die anderen Programmierer können anschließend die von Ihnen geschriebene Klassen benutzen, ohne dass diese sich mit dem Code auseinander setzen müssen, der dahinter steht. Für andere ist also Ihre Klasse eine Blackbox, vergleichbar mit einem Fernseher oder DVD-Player, bei denen man auch nicht unbedingt wissen muss, wie sie funktionieren, um diese Geräte benutzen zu können.

Vielleicht stellt sich irgendwann einmal heraus, dass sich in Ihre Klasse ein Fehler eingeschlichen hat. Dann brauchen Sie nicht in jeder Anwendung, die diese Klasse benutzt, den Code anzupassen. Sie tauschen in den betroffenen Arbeitsmappen lediglich die fehlerhafte Klasse gegen eine fehlerfreie aus. Auch Erweiterungen der Klasse sind ohne weiteres möglich und beeinflussen die bestehenden Anwendungen nicht, wenn die vorhandenen Eigenschaften und Methoden beibehalten werden.

Finden Sie später Möglichkeiten, bestimmte Eigenschaften und Methoden der Klasse effizienter zu gestalten, nur zu. Es hat keine negativen Auswirkungen auf die Anwendungen, die sich der Klasse bedienen. Sie können auch auf einfache Art einen Schreibschutz für bestimmte Eigenschaften verwirklichen, indem Sie die Prozeduren `Property Let` und `Property Set` ganz einfach weglassen.

Sie können sogar mithilfe von Klassen benutzerdefinierte Typen nachbilden, die im Gegensatz zu den normalen Typen eine gewisse Eigenintelligenz besitzen. Denkbar wären dann beispielsweise Typen, die in verschiedenen Ländern mit unterschiedlichen Maßeinheiten eingesetzt werden können und notwendige Umrechnungen intern durchführen.

3.3 Instanzierung

Klassen sind keine fertigen Objekte, sie sind quasi die Schablonen dazu. Erst durch die Instanzierung werden daraus reale Objekte, mit denen man auch etwas anfangen kann. Diese Schablone kann beliebig oft verwendet werden und jedes erzeugte Objekt ist dabei unabhängig von allen anderen und besitzt einen eigenen Satz von internen Variablen.

Eine Instanz einer Klasse können Sie auf zwei Wegen anlegen:

1. Verwenden der `Dim`-Anweisung zusammen mit `New`

   ```
   Dim myClass As New clsTest
   ```

2. Mit der `Dim`-Anweisung eine Objektvariable anlegen, mit `Set Objektvariable = New Klasse` ein Objekt erstellen und gleichzeitig der Objektvariablen zuweisen

   ```
   Dim myClass As clsTest
   Set myClass = New clsTest
   ```

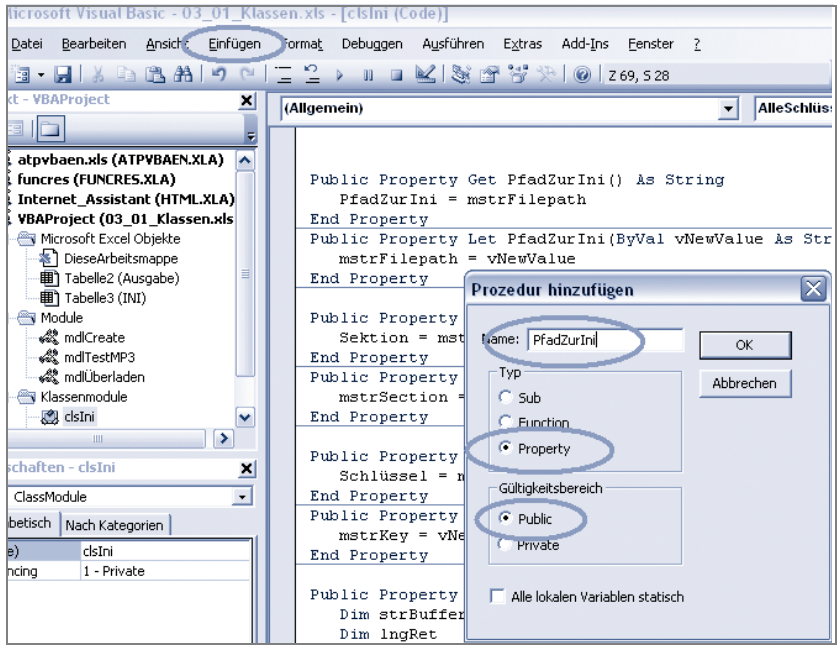
Der Unterschied zwischen beiden Varianten ist der Zeitpunkt, ab dem das Objekt tatsächlich existiert. Bei der ersten Variante wird das Objekt erst bei einem Zugriff auf eine Methode oder Eigenschaft angelegt. Bei der zweiten Methode wird das Objekt schon bei der `Set`-Anweisung zum Leben erweckt.

3.4 Eigenschaften und Methoden

Für jede Eigenschaft benötigen Sie zwei `Property`-Prozeduren oder eine öffentliche Variable. Die `Property`-Prozeduren stellen eine mögliche Schnittstelle zur Außenwelt der Klasse dar und werden beim Setzen und Lesen ausgeführt, sind also nicht unbedingt mit öffentlichen Variablen vergleichbar. Beim Auslesen wird die `Get`-, beim Setzen die `Let`- und bei Objekten die `Set`-Prozedur ausgeführt. Wenn Sie für eine Eigenschaft den Schreibschutz benötigen, lassen Sie einfach die `Property Let|Set`-Prozeduren weg.

Zum Einfügen der `Property`-Prozeduren können Sie das Menü EINFÜGEN | PROZEDUR | PROPERTY (Abbildung 3.2) in der VBE benutzen.

Abbildung 3.2
Property einfügen



Passen Sie die `Property`-Prozeduren so an, dass diese ausschließlich den Datentyp der internen Variablen aufnehmen und zurückgeben. Standardmäßig wird nämlich eine Property mit dem Datentyp `Variant` erzeugt.

Für jede Eigenschaft, die über eine `Property`-Prozedur gesetzt und ausgelesen wird, benötigen Sie zudem noch eine klassenweit gültige Variable, welche die Werte aufnimmt und für die gesamte Lebensdauer der Klasseninstanz speichert. In der Ereignisprozedur `Class_Initialize` können Sie diese Variablen initialisieren, also mit Werten vorbelegen. Diese Ereignisprozedur wird beim Anlegen der Klasseninstanz ausgeführt.

Für Methoden können Sie öffentliche Prozeduren oder Funktionen verwenden. Was Sie letztendlich verwenden, bleibt Ihnen überlassen. Eine Funktion bietet sich beispielsweise dann an, wenn als Rückgabewert der Erfolg einer Aktion signalisiert werden soll.

3.5 Überladen von Parametern

Um `Variant` bei Eigenschaftsprozeduren zu vermeiden, können Sie bei Klassen auch `Property`-Prozeduren mit gleichen Namen für verschiedene Datentypen schreiben, die je nach übergebenem Datentyp ausgeführt werden. Diese Funktionalität nennt man `Überladen`.

Hier ein kleines Beispiel (Listing 3.1) dazu:

Listing 3.1
Beispiele\03_Klassen\
03_01_Klassen.xls\mdlÜberladen

```
Sub testÜberladen()
   Dim Ueberladen As New clsUeberladen
   With Ueberladen

      .Tabellenblatt = "Ausgabe"
      MsgBox .Tabellenblatt, , "Übergabe als String"

      Set .Tabellenblatt = Worksheets(1)
      MsgBox .Tabellenblatt, , "Objektübergabe"

   End With
End Sub
```

Die Klasse `clsUeberladen` (Listing 3.2):

Listing 3.2
Beispiele\03_Klassen\
03_01_Klassen.xls\clsUeberladen

```
Option Explicit
Private mstrBlatt As String

Public Property Get Tabellenblatt() As String
   Tabellenblatt = mstrBlatt
End Property

Public Property Let Tabellenblatt(ByVal vNewValue As String)
   mstrBlatt = vNewValue
End Property

Public Property Set Tabellenblatt(ByVal vNewValue As Worksheet)
   mstrBlatt = vNewValue.Name
End Property
```

Ich glaube aber nicht, dass Sie allzu häufig in die Verlegenheit kommen, mit VBA so etwas zu realisieren.

3.6 Erstellen von Klassen

Es hört sich vielleicht etwas seltsam an, aber der beste Weg, eine Klasse zu programmieren ist der, sie erst zu benutzen und anschließend die Funktionalität der Klasse zu realisieren. Das will ich an einem kleinen Beispiel demonstrieren, wobei Einträge aus einer *.ini*-Datei gelesen und Werte in diese hineingeschrieben werden können.

3.6.1 Initialisierungsdateien

Auch in Zeiten der Registry haben Initialisierungsdateien noch ihre Daseinsberechtigung. Diese Dateien entlasten die Registry, können mit einem Texteditor bearbeitet werden und sind sehr leicht weiterzugeben. VBA bietet zwar die Möglichkeit, mittels `SaveSetting` und `GetSetting` Daten in der Registry abzulegen, leider werden diese Daten dann nur im Zweig *HKEY_CURRENT_USER\Software\VB and VBA Program Settings* abgelegt. Darauf hat aber ein anderer Benutzer keinen Zugriff, weshalb allgemeine Einstellungen für jeden Benutzer separat abgelegt werden müssen. Das ist dann aber eine redundante Datenvorhaltung in Reinkultur.

Deshalb sollten mittels `SaveSetting` und `GetSetting` nur benutzerabhängige Einstellungen gespeichert werden. Allgemeine Einstellungen von Software werden üblicherweise unter *HKEY_LOCAL_MACHINE\Software* abgelegt, aber das lässt sich unter VBA nicht so einfach realisieren.

Bei dem Weg über die Registry wird zudem noch sehr häufig vergessen, diese Einträge auch wieder zu löschen, wenn sie nicht mehr benötigt werden. Mit der Zeit wird die Registry dann so aufgebläht, dass sogar Performanceprobleme auftreten können.

Man darf auch nicht vergessen, dass es auch nicht sehr sinnvoll ist, dem normalen Benutzer das Manipulieren der Registry zuzumuten, um solche überflüssigen Einträge zu entfernen. Neben irreparablen Fehlern, die durch Löschen oder die Manipulation der falschen Schlüssel auftreten, sind die meisten Benutzer mit solch einer Aktion auch schlicht und einfach überfordert.

Eine *.ini*-Datei können Sie mit dem VBA-Befehl `Open` öffnen, auslesen und bearbeiten, Werte werden darin als Strings gespeichert. Wenn man den Aufbau dieser Dateien kennt, ist das Auslesen und Speichern gar kein so großes Problem.

Eine Initialisierungsdatei ist eine einfache Textdatei mit der Dateinamenserweiterung *.ini*. Sie ist eingeteilt in Sektoren, in denen sich Schlüssel mit den zugehörigen Werten befinden. Nachfolgend beispielhaft der Aufbau, wie Sie ihn in einem Texteditor sehen würden:

```
[Sektion1]
Schlüssel1=Wert1
Schlüssel2=Wert2

[Sektion2]
Schlüssel1=Wert1
Schlüssel2=Wert2
```

Erheblich leichter funktioniert das Auslesen und Speichern von Schlüsselinhalten aber mit den Betriebssystemfunktionen `WritePrivateProfileString` und `GetPrivateProfileString`, die Sie mit der `Declare`-Anweisung auch unter VBA benutzen können.

3.6.2 Das Benutzen der Klasse clsIni

In einem Standardmodul – in diesem Beispiel ist es das Modul `mdlCreate` – erstellen Sie sich erst einmal eine Prozedur, die eine Instanz der Klasse `clsIni` anlegt und welche die gewünschten Eigenschaften und Methoden der Klasse benutzt. Dazu muss die Klasse noch nicht einmal existieren.

Die wichtigste Eigenschaft der zukünftigen Klasse ist sicherlich der Pfad zu der Datei, die als Container für die zu speichernden Werte dienen soll. Diese Eigenschaft nennen wir sinnigerweise `PfadZurIni`.

Des Weiteren benötigt man die Angabe der Sektion, in welcher der Schlüssel gespeichert oder aus der er ausgelesen werden soll. Wichtig ist auch der Name des Schlüssels, der den Wert aufnehmen soll, und schließlich der Wert selber. Diese drei Eigenschaften bekommen die Namen `Sektion`, `Schlüssel` und `Wert`.

Die vorher angesprochenen Eigenschaften sollen gesetzt und anschließend in die Datei geschrieben werden. Man könnte schon beim Setzen einer dieser Eigenschaften das Schreiben anstoßen, das ist aber keine gute Idee, denn es werden zum Speichern erst alle Eigenschaften benötigt.

Also brauchen Sie eine Methode, die erst dann aufgerufen wird, wenn die benötigten Eigenschaften gesetzt sind. Diese Methode nennen wir `AnlegenÄndern`, denn eine Sektion, ein Schlüssel und ein Wert kann neu angelegt oder geändert werden. In diesem Beispiel benutzte ich statt einer Prozedur eine Funktion, denn mit einer Funktion kann man zusätzlich noch einen Wahrheitswert zurückgeben, der Auskunft über den Erfolg der Aktion gibt.

Schlüssel und Sektionen sollte man auch löschen können, also werden dafür auch zwei Methoden benötigt. Hier sind das die Methoden `SchlüsselLöschen` und `SektionLöschen`, die auch wieder als Funktionen mit Erfolgsmeldung ausgeführt sind.

Schön wäre es noch, wenn man alle Sektionen einer Datei oder alle Schlüssel einer Sektion auflisten könnte. Also werden auch dafür zwei Eigenschaften mit den Namen `AlleSektionen` und `AlleSchlüssel` angelegt, die in der Klasse als Funktion ausgeführt sind und jeweils ein Array mit dem Ergebnis liefern sollen.

In dem vorliegendem Beispiel wird im Temp-Pfad eine Initialisierungsdatei angelegt. Zum Ermitteln des Temp-Pfades, das ist das temporäre Verzeichnis des jeweiligen Benutzers, wird die API-Funktion `GetTempPath` benutzt. Anschließend werden in der Initialisierungsdatei zwei Sektionen mit jeweils zwei Schlüsseln und den zugehörigen Werten gespeichert. Zur Demonstration werden alle Sektionen dieser Datei, danach alle Schlüssel einer Sektion ausgelesen und angezeigt.

Anschließend wird einer dieser angelegten Schlüssel gelöscht und alle noch vorhandenen Schlüssel dieser Sektion werden ausgelesen und angezeigt. Danach wird eine Sektion gelöscht und alle Sektionen der Datei werden angezeigt.

Hier also der Code (Listing 3.3), mit dem Sie die noch anzulegende Klasse benutzen:

Listing 3.3
Beispiele\03_Klassen\
03_01_Klassen.xls\mdlCreate

```
Private Declare Function GetTempPath _
   Lib "kernel32" Alias "GetTempPathA" ( _
   ByVal nBufferLength As Long, _
   ByVal lpBuffer As String _
   ) As Long

Public Sub TestEnviron()
   Dim lngRet     As Long
   Dim strPath    As String
   Dim myIni      As New clsIni
   Dim varSection As Variant
   Dim varKey     As Variant
   Dim varItem    As Variant
   Dim strMsg     As String

   strPath = String(255, 0)
   lngRet = GetTempPath(Len(strPath), strPath)
   strPath = Left(strPath, lngRet)
```

Listing 3.3 (Forts.)
Beispiele\03_Klassen\
03_01_Klassen.xls\mdlCreate

```
With myIni
   .PfadZurIni = strPath & "Test.ini"

   ' Neue Sektion [Sektion]
   .Sektion = "Sektion"
      .Schlüssel = "Schlüssel1"
      .Wert = "Wert1"
      ' Anlegen : Schlüssel1=Wert1
      .AnlegenÄndern

      .Schlüssel = "Schlüssel2"
      .Wert = "Wert2"
      ' Anlegen : Schlüssel2=Wert2
      .AnlegenÄndern

   ' Neue Sektion [Sektion1]
   .Sektion = "Sektion1"
      .Schlüssel = "Schlüssel11"
      .Wert = "Wert11"
      ' Anlegen : Schlüssel11=Wert11
      .AnlegenÄndern

      .Schlüssel = "Schlüssel12"
      .Wert = "Wert12"
      ' Anlegen : Schlüssel12=Wert12
      .AnlegenÄndern

   ' Alle Sektionen auslesen
   varSection = .AlleSektionen
   strMsg = ""
   For Each varItem In varSection
      strMsg = strMsg & varItem & vbCrLf
   Next
   MsgBox strMsg, , "Alle Sektionen"

   ' Alle Schlüssel auslesen
   .Sektion = varSection(1)
   varKey = .AlleSchlüssel

   strMsg = ""
   For Each varItem In varKey
      strMsg = strMsg & varItem & vbCrLf
   Next
   MsgBox strMsg, , "Alle Schlüssel in " & .Sektion

   ' Wert aus Sektion1 löschen
   .Schlüssel = varKey(2)
   .SchlüsselLöschen

   ' Alle Schlüssel auslesen
   varKey = .AlleSchlüssel
   strMsg = ""
   For Each varItem In varKey
      strMsg = strMsg & varItem & vbCrLf
   Next
```

Listing 3.3 (Forts.)
Beispiele\03_Klassen\
03_01_Klassen.xls\mdlCreate

```
      MsgBox strMsg, , "Alle Schlüssel in " & .Sektion

      .Sektion = varSection(0)
      .SektionLöschen

      ' Alle Sektionen auslesen
      varSection = .AlleSektionen
      strMsg = ""
      For Each varItem In varSection
         strMsg = strMsg & varItem & vbCrLf
      Next
      MsgBox strMsg, , "Alle Sektionen"
   End With
End Sub
```

3.6.3 Die Klasse clsIni

Bis jetzt haben Sie noch keine Zeile Code der eigentlichen Klasse geschrieben. Steht fest, welche Eigenschaften und Methoden die Klasse offen legen soll, können Sie sich an die Programmierung der Klasse machen.

Fügen Sie dazu in Ihr Projekt eine leere Klasse ein und vergeben Sie den Klassennamen `clsIni`. Nachfolgend der Code der Klasse (Listing 3.4):

Listing 3.4
Beispiele\03_Klassen\
03_01_Klassen.xls\clsIni

```
Private Declare Function GetPrivateProfileString _
   Lib "kernel32" Alias "GetPrivateProfileStringA" ( _
   ByVal lpApplicationName As String, _
   ByVal lpKeyName As Any, _
   ByVal lpDefault As String, _
   ByVal lpReturnedString As String, _
   ByVal nSize As Long, _
   ByVal lpFileName As String _
   ) As Long
   ' Funktion, um Einträge aus .ini-Datei zu lesen

Private Declare Function WritePrivateProfileString _
   Lib "kernel32" Alias "WritePrivateProfileStringA" ( _
   ByVal AppName As String, _
   ByVal KeyName As String, _
   ByVal keydefault As String, _
   ByVal FileName As String _
   ) As Long
   ' Funktion, um Einträge in .ini-Datei zu schreiben

Private mstrFilepath As String
Private mstrSection  As String
Private mstrKey      As String
Private mstrValue    As String

Public Function AnlegenÄndern() As Boolean
   If WritePrivateProfileString( _
      mstrSection, mstrKey, _

      mstrValue, mstrFilepath _
      ) <> 0 Then AnlegenÄndern = True
End Function
```

Listing 3.4 (Forts.)
Beispiele\03_Klassen\
03_01_Klassen.xls\clsIni

```
Public Function SektionLöschen() As Boolean
   If WritePrivateProfileString( _
      mstrSection, vbNullString, _
      vbNullString, mstrFilepath _
      ) <> 0 Then SektionLöschen = True
End Function

Public Function SchlüsselLöschen() As Boolean
   If WritePrivateProfileString( _
      mstrSection, mstrKey, _
      vbNullString, mstrFilepath _
      ) <> 0 Then SchlüsselLöschen = True
End Function

Public Function AlleSektionen() As Variant
   Dim strBuffer  As String
   Dim lngRet     As Long

   strBuffer = String(1000, 0)

   lngRet = GetPrivateProfileString( _
      vbNullString, vbNullString, _
      "Schlüssel fehlt", strBuffer, _
      999, mstrFilepath)

   AlleSektionen = Split(Left(strBuffer, lngRet - 1), Chr(0))

End Function

Public Function AlleSchlüssel() As Variant
   Dim strBuffer  As String
   Dim lngRet     As Long

   strBuffer = String(1000, 0)

   lngRet = GetPrivateProfileString( _
      mstrSection, vbNullString, _
      "", strBuffer, _
      999, mstrFilepath)

   AlleSchlüssel = Split(Left(strBuffer, lngRet - 1), Chr(0))

End Function

Public Property Get PfadZurIni() As String
   PfadZurIni = mstrFilepath
End Property
Public Property Let PfadZurIni(ByVal vNewValue As String)
   mstrFilepath = vNewValue
End Property

Public Property Get Sektion() As String
   Sektion = mstrSection
End Property
Public Property Let Sektion(ByVal vNewValue As String)
   mstrSection = vNewValue
End Property
```

Listing 3.4 (Forts.)
Beispiele\03_Klassen\
03_01_Klassen.xls\clsIni

```
Public Property Get Schlüssel() As String
   Schlüssel = mstrKey
End Property
Public Property Let Schlüssel(ByVal vNewValue As String)
   mstrKey = vNewValue
End Property

Public Property Get Wert() As String
   Dim strBuffer  As String
   Dim lngRet     As Long

   strBuffer = String(1000, 0)

   lngRet = GetPrivateProfileString( _
      mstrSection, mstrKey, _
      "Schlüssel fehlt", strBuffer, _
      999, mstrFilepath)

   mstrValue = Left(strBuffer, lngRet)
   Wert = mstrValue
End Property
Public Property Let Wert(ByVal vNewValue As String)
   mstrValue = vNewValue
End Property
```

AnlegenÄndern

Die als öffentliche Methode benutzte Funktion `AnlegenÄndern` legt eine *.ini*-Datei an, falls diese noch nicht existiert. Das entsprechende Verzeichnis muss aber bereits existieren. Zuständig dafür ist die API-Funktion `WritePrivateProfile-String`. Dieser Funktion wird als Parameter der Dateipfad inklusive Dateiname, Sektion, Schlüsselname und zugehörigem Wert übergeben. Die API-Funktion `WritePrivateProfileString` liefert als Ergebnis einen Longwert zurück, der ungleich null ist, wenn die Funktion erfolgreich war. Dieser wird dazu benutzt, den Wert `WAHR` als Funktionsergebnis zurückzugeben, wenn alles geklappt hat.

SchlüsselLöschen

Die als öffentliche Methode benutzte Funktion `SchlüsselLöschen` löscht einen existierenden Schlüssel. Zuständig dafür ist wiederum die API-Funktion `WritePrivateProfileString`. Die an diese Funktion übergebenen Parameter sind der Dateipfad inklusive Dateiname, Sektion und Schlüsselname. Damit der Schlüssel gelöscht wird, muss als Parameter für den Wert ein `vbNullString` übergeben werden. Wenn alles geklappt hat, was Sie am Rückgabewert von `WritePrivateProfileString` erkennen können, wird als Funktionsergebnis `WAHR` zurückgegeben.

SektionLöschen

Die als öffentliche Methode benutzte Funktion `SektionLöschen` löscht eine existierende Sektion in der angegebenen Datei. Zuständig für die eigentliche Arbeit ist, wie nicht schwer zu erraten ist, die API-Funktion `WritePrivateProfileString`. Dieser wird diesmal als Parameter der Dateipfad inklusive dem Dateinamen und die zu löschende Sektion übergeben. Damit die Sektion auch tatsächlich gelöscht wird, muss als Parameter für den Schlüsselnamen und für den Wert ein `vbNullString` übergeben werden. Wie bei den anderen Methoden wird als Funktionsergebnis `WAHR` zurückgegeben, wenn alles geklappt hat.

Wert

Die Property `Get Wert` liest den Wert eines Schlüssels aus. Zuständig dafür ist die API-Funktion `GetPrivateProfileString`. Dieser Funktion wird als Parameter der Dateipfad inklusive Dateiname, Sektion und Schlüsselname übergeben. Sie liefert als Ergebnis einen Longwert zurück, der ungleich null ist, wenn die Funktion erfolgreich war. Der eigentliche Wert wird im Parameter `lpReturnedString` zurückgeliefert. Dazu wird an dieser Stelle ein ausreichend großer Puffer übergeben, der den String aufnimmt. Der Parameter `nSize` gibt die Länge des Puffers an und `lpDefault` ist ein String, der geliefert wird, wenn der Schlüssel nicht existiert.

In der `Let`-Prozedur bekommt lediglich die interne Variable `mstrValue` den übergebenen Wert zugewiesen. Das Schreiben übernimmt die Methode `Anlegen-Ändern`.

AlleSektionen

Diese Property ist schreibgeschützt, da nur eine `Get`-Prozedur angelegt wurde. Der Code ist fast der gleiche wie in der Property `Get Wert`, lediglich der Parameter für die Sektion muss auf `vbNullString` gesetzt werden. In diesem Fall werden alle Sektionen durch ein `chr(0)` getrennt im Parameter `lpReturnedString` zurückgeliefert. Mit der `Split`-Funktion wird daraus ein Array gemacht und an den Aufrufer geliefert.

AlleSchlüssel

Diese Property ist schreibgeschützt, da nur eine `Get`-Prozedur angelegt wurde. Der Code ist fast der gleiche wie in `AlleSektionen`, lediglich der Parameter für den Schlüssel muss auf `vbNullString` gesetzt werden. In diesem Fall werden alle Schlüssel der Sektion durch ein `chr(0)` getrennt im Parameter `lpReturnedString` zurückgeliefert. Mit der `Split`-Funktion wird daraus ein Array gemacht und an den Aufrufer geliefert.

PfadZurIni, Sektion, Schlüssel

In diesen Property-Prozeduren werden lediglich die internen Variablen ausgelesen und gesetzt.

3.7 Ereignisprozeduren WithEvents

Eine weitere wichtige Eigenschaft von Klassen ist, dass sie Ereignisse auslösen und auch Ereignisse von Objekten, die mit `WithEvents` deklariert wurden, empfangen können. In Klassenmodulen von UserForms oder Tabellenblättern stehen Ihnen schon vordefinierte Ereignisprozeduren zur Verfügung. Wenn Sie dort Objekte einfügen, können Sie auch deren Ereignisse benutzen.

Wollen Sie aber zum Beispiel Command-Buttons dynamisch in UserForms einfügen, d.h. während der Laufzeit und nicht während der Entwicklungszeit, haben Sie das Problem, dass Ereignisprozeduren nicht als Arrays ausgeführt werden können. Sie benötigen also für jedes Objekt eine eigene Ereignisprozedur.

Mit `WithEvents` und einer Klasse können Sie das ohne programmgesteuertes Einfügen von Code realisieren. Das Konzept ist anfangs nicht einfach zu verstehen, wenn Sie es aber ein paarmal durchgespielt hat, ist es gar nicht mehr so schwer.

3.7.1 Kontrollkästchen und Tabellenblatt

An folgendem Beispiel, das Schritt für Schritt aufgebaut wird, kann man sich die Vorgehensweise am besten verdeutlichen. Es sollen programmgesteuert Kontrollkästchen (CheckBox) in ein Tabellenblatt eingefügt werden und die Klick-Ereignisse empfangen und ausgewertet werden:

1. Sie legen fest, von welchem Objekttyp Sie ein Ereignis empfangen wollen. Wir nehmen an, dass es sich dabei um eine `CheckBox` von `MSForms` handeln soll.
2. Fügen Sie zu Ihrem Projekt eine Klasse hinzu, als Name dafür wird hier `clsSheetEvent` vergeben.
3. Im Deklarationsbereich dieser Klasse legen Sie eine öffentliche Objektvariable des gewünschten Objekttyps an, in diesem Fall `MSForms.CheckBox`, und zwar mit dem Schlüsselwort `WithEvents`. Das Schlüsselwort `WithEvents` legt fest, dass dieses Objekt in dieser Klasse ein Ereignis (Event) empfangen kann.

   ```
   Public WithEvents objCheckBox As MSForms.CheckBox
   ```

4. Im Deklarationsbereich der Klasse legen Sie eine weitere Objektvariable an. In diesem von außen übergebenen Objekt soll später eine Prozedur aufgerufen werden.

   ```
   Private mobjOwner As Object
   ```

5. Der Objektvariablen `mobjOwner` muss man von außen ein Objekt zuweisen können, also legen Sie dafür eine Eigenschaftsprozedur (`Property`) an. Da es sich um ein Objekt handelt, müssen Sie `Property Set` benutzen.

   ```
   Public Property Set Owner(ByVal vNewValue As Object)
      Set mobjOwner = vNewValue
   End Property
   ```

6. Jetzt erstellen Sie in der Klasse die Ereignisprozedur des Objektes. Das Ereignis ist dabei durch einen Unterstrich vom Objekt getrennt. Die angelegte Prozedur wird in diesem Fall durch das Ändern der mit der Objektvariablen `objCheckBox` verbundenen `CheckBox` ausgeführt.

   In dieser Prozedur rufen Sie die Zielprozedur `myCheckbox_KlickEvent` des als `mobjOwner` übergebenen Objekts auf und übergeben dabei noch den Status der Checkbox.

```
Private Sub objCheckBox_Change()
   ' Klickevent des Buttons
   On Error Resume Next

   ' Eine öffentliche Prozedur in der als Owner übergebenen
   ' Klasse ausführen
   mobjOwner.myCheckbox_KlickEvent objCheckBox.Value
End Sub
```

7. Fügen Sie das Objekt, das ein Ereignis auslösen soll, in ein Tabellenblatt ein.
8. Legen Sie im Klassenmodul des Tabellenblattes eine öffentliche Prozedur an, die bei einem Ereignis aufgerufen werden soll.

```
Public Sub myCheckbox_KlickEvent(strValue As String)
```

9. In einer Prozedur erstellen Sie für jede Checkbox in dem Tabellenblatt eine neue Instanz der Klasse `clsCheckBox`. Damit die Instanz nicht nach dem Beenden der Prozedur gelöscht wird, haben Sie vorher eine Collection auf Klassenebene deklariert, der Sie die Instanz als neues Element übergeben.

   Während der Lebensdauer der Collection, also bis die Codezeile `Set myCol = New Collection` oder `Set myCol = Nothing` ausgeführt wird, bleibt die hinzugefügte Klasseninstanz am Leben.

   Die Eigenschaft `Owner` der angelegten Klasse bekommt einen Verweis auf das eigene Tabellenblatt.

```
Public Sub MakeEvents()
   Dim x As OLEObject
   Set mobjEvents = New Collection
   For Each x In ActiveSheet.OLEObjects
      If TypeName(x.Object) = "CheckBox" Then
         Set mobjCheck = New clsSheetEvent
         Set mobjCheck.objCheckBox = x.Object
         Set mobjCheck.Owner = Me
         mobjEvents.Add mobjCheck
      End If
   Next
End Sub
```

Und hier noch einmal der zusammenhängende Code. In eine Klasse mit Namen `clsSheetEvent` (Listing 3.5):

Listing 3.5
Beispiele\03_Klassen\
03_02_Events.xls\clsSheetEvent

```
' Events von diesem Objekt können hier ausgeführt werden
Public WithEvents objCheckBox As MSForms.CheckBox

' Objektvariable zur Aufnahme einer Referenz
Private mobjOwner As Object

Private Sub objCheckBox_Change()
    ' Changeevent der Checkbox
   On Error Resume Next

    ' Eine öffentliche Prozedur in der als Owner übergebenen
    ' Klasse ausführen
   mobjOwner.myCheckbox_KlickEvent objCheckBox.Value
End Sub

Public Property Set Owner(ByVal vNewValue As Object)
    ' Besitzer übergibt eine Referenz auf das Objekt,
    ' damit dort eine Prozedur aufgerufen werden kann
   Set mobjOwner = vNewValue
End Property
```

In das Klassenmodul eines Tabellenblattes (Listing 3.6):

Listing 3.6
Beispiele\03_Klassen\
03_02_Events.xls\Tabelle1

```
' Collection, damit die Objekte am Leben bleiben
Private mobjEvents As Collection

Private mobjCheck As clsSheetEvent

Public Sub myCheckbox_KlickEvent(strValue As String)
    ' Diese öffentliche Prozedur wird von jeder geladenen
    ' Klasse clsTestEvents nach einem Klick aufgerufen
    ' Meldung ausgeben
   MsgBox strValue
End Sub
Public Sub MakeEvents()
   Dim x As OLEObject

   Set mobjEvents = New Collection

   For Each x In ActiveSheet.OLEObjects

      If TypeName(x.Object) = "CheckBox" Then

         Set mobjCheck = New clsSheetEvent
         Set mobjCheck.objCheckBox = x.Object
         Set mobjCheck.Owner = Me
         mobjEvents.Add mobjCheck

      End If

   Next
End Sub
```

3.7.2 CommandButton und UserForms

Nachfolgend ein Beispiel zum dynamischen Einfügen von Buttons und Empfangen des Klickereignisses in einer Klasse. In der Klasse wiederum kann ein Ereignis ausgelöst werden, das dann in der Besitzerklasse ausgeführt wird.

Erst einmal der Code der Klasse `clsTestEvents` (Listing 3.7):

Listing 3.7
Beispiele\03_Klassen\
03_02_Events.xls\clsTestEvents

```
' Events von diesem Objekt können hier ausgeführt werden
Public WithEvents objCommandButton As MSForms.CommandButton

' Dieses Event kann in anderen Klassen ausgelöst werden
Public Event KlickEvent(strName As String)

' Objektvariable zur Aufnahme einer Referenz
Private mobjOwner As Object

Private Sub objCommandButton_Click()
    ' Klickevent des Buttons

   On Error Resume Next
    ' Eine öffentliche Prozedur in der als Owner übergebenen
    ' UserForm ausführen
    mobjOwner.myButton_KlickEvent objCommandButton.Name

    ' Event in der Besitzerklasse auslösen und
    ' Name übergeben
   RaiseEvent KlickEvent(objCommandButton.Name)

End Sub

Public Property Set Owner(ByVal vNewValue As Object)
    ' Besitzer übergibt eine Referenz auf die UserForm,
     'damit dort eine Prozedur aufgerufen werden kann
   Set mobjOwner = vNewValue
End Property
```

Hier der Code, der in das Klassenmodul einer UserForm gehört (Listing 3.8). In diese UserForm gehört noch ein Button mit dem Namen `cmdCreateNew`.

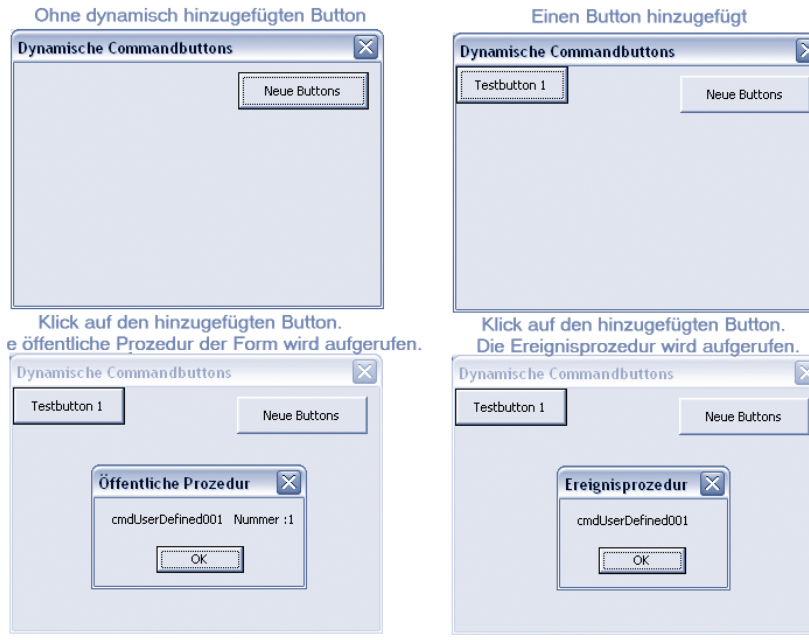
Listing 3.8
Beispiele\03_Klassen\
03_02_Events.xls\frmEventAll

```
' Anzahl der UserForms
Private lngControlCount As Long

' Y Position der UserForm
Private lngYPos As Long

' Der gemeinsame Namensteil der Buttons
Private Const ButtonName = "cmdUserDefined"

' Collection, damit die Objekte am Leben bleiben
Public objEvents As New Collection

' Objektvariable anlegen und festlegen, dass Ereignisse
' empfangen werden können
Private WithEvents objButton As clsTestEvents
```

Listing 3.8 (Forts.)
Beispiele\03_Klassen\
03_02_Events.xls\frmEventAll

```
Private Sub AddCommand()

    ' Anzahl der Buttons speichern
    lngControlCount = lngControlCount + 1

    ' Mehr als vier Buttons wollen wir nicht
    If lngControlCount > 4 Then Exit Sub

    ' Neues Klassenobjekt anlegen
    Set objButton = New clsTestEvents

    ' Referenz auf die UserForm übergeben
    Set objButton.Owner = Me

    ' Button anlegen
    Set objButton.objCommandButton = _
       Me.Controls.Add _
       ("Forms.CommandButton.1", _
       ButtonName & Format(lngControlCount, "000"))

    ' Position und Name festlegen
    With objButton.objCommandButton
       .Top = lngYPos
       .Caption = "Testbutton " & lngControlCount
    End With

    lngYPos = lngYPos + 25

    'Klassenobjekte im Speicher halten
    objEvents.Add objButton
End Sub

Public Sub objButton_KlickEvent(strObjektname As String)
    ' Nur das letzte Klassenobjekt feuert dieses
    ' Event, denn objButton hält nur eine Referenz auf
    ' den letzten Button. Man bräuchte für jedes Objekt
    ' solch eine Ereignisprozedur, also auch mehrere
    ' Objektvariablen. Arrays sind nicht erlaubt.
    MsgBox strObjektname, , "Ereignisprozedur"
End Sub

Public Sub myButton_KlickEvent(strObjektname As String)
    'Diese öffentliche Prozedur wird von jeder geladenen
    'Klasse clsTestEvents nach einem Klick aufgerufen
    Dim lngIndex As Long

    'Steuerelementindex extrahieren
    lngIndex = CLng(Right(strObjektname, 3))

    'Meldung ausgeben
    MsgBox strObjektname & "   Nummer :" & lngIndex, , _
       "Öffentliche Prozedur"
End Sub

Private Sub cmdCreateNew_Click()
    AddCommand
End Sub
```

Abbildung 3.3
Verschiedene Ansichten der UserForm



Die Funktionalität der UserForm

Bei einem Klick auf den Button `cmdCreateNew` wird die Prozedur `AddCommand` aufgerufen. Wenn bereits mehr als vier Buttons angelegt wurden, wird diese aber sofort wieder verlassen.

Die auf Klassenebene angelegte Objektvariable `objButton` nimmt anschließend mit `Set` und `New` eine neue Instanz der Klasse `clsTestEvents` auf. Da diese mit dem Schlüsselwort `WithEvents` deklariert wurde, können Ereignisse des damit verbundenen Objektes empfangen werden.

Anschließend können Sie eine Objektreferenz der UserForm an die soeben instanzierte Klasse übergeben. Damit wird erreicht, dass die Klasse eine öffentliche Prozedur der UserForm aufrufen kann.

Zum Einfügen eines Buttons wird die `Add`-Methode der Controls-Auflistung in der UserForm benutzt. Gleichzeitig bekommt die Klasse über die als öffentlich deklarierte Variable `objCommandButton` eine Objektreferenz auf den soeben hinzugefügten Button. In der Klasse `clsTestEvents` ist diese Variable mit `WithEvents` deklariert worden, also kann dort das Objekt Ereignisse auslösen. Anschließend wird die Position des Buttons festgelegt werden, ich erhöhe aber jeweils nur die Y-Position.

Würde man an dieser Stelle die Prozedur einfach beenden, hätte man beim nächsten Einfügen eines Buttons ein Problem: Wenn Sie mit `Set` und `New` eine neue Instanz der Klasse `clsTestEvents` anlegen und sie der Objektvariablen `objButton` zuweisen, wird die ältere, mit der Objektvariablen verbundene Klasseninstanz aus dem Speicher gelöscht.

Also benutzen Sie eine klassenweit gültige Collection und fügen die Objektvariable als Element hinzu. Dadurch wird erreicht, dass noch zusätzlich eine Referenz auf das Klassenobjekt angelegt wird, die so lange bestehen bleibt, bis sie aus der Collection entfernt wird oder die Objektvariable der Collection auf `Nothing` gesetzt wird. Wird die Objektvariable `objButton` für eine neue Klasseninstanz benutzt, bleibt wegen der noch bestehenden Referenz auch die ältere Instanz noch im Speicher.

Die Prozedur `objButton_KlickEvent` ist eine Ereignisprozedur, die von dem mit der Objektvariablen `objButton` verbundenen Objekt ausgelöst wird. Deshalb wurde auch bei der Deklaration das Schlüsselwort `WithEvents` benutzt.

Die Prozedur `myButton_KlickEvent` ist eine öffentliche Prozedur. Diese wird bei einem Klick auf einen Button von der damit verbundenen Klasse aufgerufen.

clsTestEvents

Die Objektvariable `objCommandButton` vom Typ `MSForms.CommandButton` ist mit dem Schlüsselwort `WithEvents` deklariert worden.

Im Deklarationsteil dieser Klasse wird noch das Event `KlickEvent` angelegt, das man innerhalb der Klasse feuern kann. Feuern bedeutet, dass Sie in der Klasse, die eine Instanz auf diese Klasse hält, eine Ereignisprozedur auslösen können. Dazu muss die Objektvariable, die eine Instanz auf die feuernde Klasse hält, also in der UserForm die Variable `objButton` mit dem Schlüsselwort `WithEvents`, deklariert sein.

Wenn Sie in der UserForm als Prozedurnamen den Objektnamen benutzen, einen Unterstrich und den Ereignisnamen dranhängen, kann die feuernde Klasse diese Prozedur aufrufen und ihr sogar Parameter übergeben. Ausgelöst wird das durch die Anweisung `RaiseEvent KlickEvent(objCommandButton.Name)`.

Umgekehrt kann das mit der Objektvariablen `objCommandButton` verbundene Objekt – in diesem Beispiel ein eingefügter Button – eine Ereignisprozedur in der verbundenen Klasse aufrufen. Dazu wird wieder der Objektname mit dem Ereignisnamen durch einen Unterstrich verbunden und als Prozedurname benutzt (`objCommandButton_Click`).

In der Property-Prozedur `Property Set Owner` wird eine Referenz auf die UserForm an die Klasse übergeben und in der Variablen `mobjOwner` gespeichert. Damit ist es möglich, eine öffentliche Prozedur des aufrufenden Objektes ausführen zu lassen. Die Zeile `mobjOwner.myButton_KlickEvent` ruft dann die entsprechende Prozedur in der UserForm auf.

Durch diesen kleinen Kniff können Sie in der UserForm eine einzige Prozedur für alle Klickereignisse realisieren. Damit unterschieden werden kann, welcher Button gerade angeklickt wurde, übergeben Sie zusätzlich den Namen des Buttons.

4

Datenbanken

4.1 Was Sie in diesem Kapitel erwartet

Über Datenbanken und die Zugriffe darauf gibt es so viel zu berichten, dass man ganze Bücher darüber schreiben könnte. Die sind natürlich auch schon geschrieben worden, dabei ist die Zielgruppe aber meist nicht der Excel-Anwender, der mal eben auf ein paar Daten aus einer Access-Datenbank zugreifen will. Aber genau das braucht man ab und zu, sei es, um die Daten weiterzuverarbeiten oder mit Ihnen Kalkulationen durchzuführen.

Das gesamte Thema Datenbanken ist aber dermaßen umfangreich, dass man im Rahmen dieses Buches nicht so umfassend darauf eingehen kann, dass Sie anschließend das Konzept von relationalen Datenbanken verstehen oder alle Feinheiten von SQL beherrschen. Aber eine kleine Einführung und ein paar Beispiele will ich Ihnen in diesem Kapitel geben, damit Sie ohne großen Aufwand auf verschiedene Datenbanken zugreifen können.

4.2 Excel ist keine Datenbank

Recht häufig wird Excel als Datenbank missbraucht, dabei ist Excel ein reines Tabellenkalkulationsprogramm. Nicht mehr, aber auch nicht weniger.

Wenn man die ganze Sache einmal nüchtern betrachtet, muss man sich fragen, warum in aller Welt überhaupt jemand auf die Idee kommen kann, ein Tabellenkalkulationsprogramm für die Datenspeicherung zu benutzen. Es gibt ja schließlich Datenbankprogramme, die extra für die Aufgabe geschrieben wurden, Daten zu speichern und zu verwalten. Sogar das Professional Paket von Office enthält mit Access ein ganz hervorragendes Datenbankprogramm.

Die Antwort für den Grund des Missbrauchs ist offensichtlich. Excel verführt den unbedarften Benutzer förmlich dazu. Startet man das Programm mit einem leeren Tabellenblatt, bietet es sich sofort an, Daten dort zu speichern. Dabei

repräsentieren dann die Zeilen einzelne Datensätze und die Spalten dessen Felder. Das ist genau das, was sich der Laie unter dem natürlichen Aufbau einer Datenbank vorstellt.

Denken Sie bei der Speicherung von größeren Datenmengen am besten gar nicht an Excel. Wenn Sie Daten dauerhaft speichern und vernünftig verwalten wollen, benutzen Sie eine Datenbank wie z.B. Access. Dafür sind Datenbanken schließlich gemacht!

Excel bietet zudem noch kleinere Datenbankfeatures wie die Eingabemaske und ein paar datenbankspezifische Tabellenfunktionen, die den Benutzer noch zusätzlich dazu verleiteten, Excel mit der Datenverwaltung zu quälen. Wenn Sie eine Hand voll Telefonnummern oder Adressen verwalten, wird das alles sicherlich ganz gut klappen. Ich gestehe sogar, dass ich so etwas auch schon gemacht habe.

Sobald aber eine größere Menge Daten verwaltet werden soll, bekommt man schneller als man denkt Probleme mit doppelten oder nicht abgeglichenen Daten. Außerdem neigt man unwillkürlich dazu, Daten redundant, also doppelt, vorzuhalten. Die Eigenschaften einer relationalen Datenbank lassen sich mit Excel nur sehr schwer nachbilden. Vieles, was in Excel gar nicht oder nur sehr umständlich zu lösen ist, ist bei einer Datenbank Standard.

Viele Anfragen in Newsgroups wären überflüssig, wenn man für die Aufgabe der Datenspeicherung das richtige Programm benutzen würde. Suchen Sie spaßeshalber mal unter *groups.google.com* nach den Stichworten »`Excel doppelte Daten`«, vielleicht werden Sie die Problematik dann erahnen können.

Oracle, SQLSERVER oder ähnliche große Datenbanken sind meiner Ansicht nach für reine Privatanwender überzogen und erfordern zudem ein enormes Fachwissen. Ein Datenbankadministrator ist schließlich noch nicht vom Himmel gefallen.

Ich will damit keinesfalls zum Ausdruck bringen, dass Access kein Fachwissen erfordert. Das Programm ist aber wesentlich einfacher zu bedienen und es lassen sich mithilfe von Assistenten sehr einfach neue Datenbanken erstellen. Natürlich sollte man sich vorher mit den Grundkonzepten von relationalen Datenbanken vertraut machen.

Ein nicht zu unterschätzender Vorteil von Access gegenüber seinen großen Brüdern ist die unkomplizierte Datensicherung. Einfach die *.mdb*-Datei wegsichern und das war es. In dieser Datei steckt alles Wichtige drin. Das sollte bei der Größe der heutigen Datenträger auch hinsichtlich des Speicherbedarfs keine Probleme mehr bereiten.

Sind die Daten erst einmal in einer Datenbank gespeichert, haben Sie mit Excel einige Möglichkeiten, auf die Daten zuzugreifen. Eine von mir favorisierte Möglichkeit ist, unter Zuhilfenahme von VBA und `ADO` eine Datenbankabfrage zu starten. Sie können sich dabei mittels einer SQL-Abfrage die notwendigen Daten holen und die eigentliche Auswertung in Excel erledigen. Das hat den Vorteil, dass Sie durch eine geeignete SQL-Abfrage die Datenmenge auf das wirklich absolut notwendige reduzieren und dabei schon nach gewissen Kriterien vorsortieren können.

Wenn Sie die Daten in einer Access-Datenbank gespeichert haben, könnten Sie sicherlich die anschließende Auswertung auch gleich in Access machen und

sich somit den Weg über Excel sparen. Das ist auch vollkommen richtig, wenn Sie nur einfache Berichte erstellen wollen (ich höre schon jetzt den Aufschrei von Accesslern). Wer aber mit den Daten weiterrechnen, damit kalkulieren und möglicherweise auch verschiedene Szenarien durchspielen will, ist mit der kombinierten Lösung allemal besser bedient. Access ist nun mal keine Tabellenkalkulation. Es gilt auch hier, für die jeweilige Aufgabe das richtige Programm zu wählen.

Ein weiterer Vorteil von ADO und einer Access-Datenbank ist der, dass Sie für den Zugriff auf diese Daten nicht einmal im Besitz von Access sein müssen, was ja in vielen Fällen auch ein gewichtiges Kriterium ist.

4.3 ADOX (ActiveX Data Objects Extension)

Bei *ADOX* hat man es mit einer Schnittstelle zu tun, die den Zugriff auf die Datenbankstruktur erlaubt. *ADOX* steht dabei für *Microsoft ADO Ext. 2.X for DDL and Security*. Damit kann man Access-Datenbanken vollständig vom VBA-Code aus verwalten, dazu ist auch kein installiertes Access notwendig.

Um *ADOX* zu benutzen, können Sie in der VBE unter EXTRAS | VERWEISE einen Verweis auf die `Microsoft ADO Ext. 2.X for DDL and Security (MSADOX)` setzen.

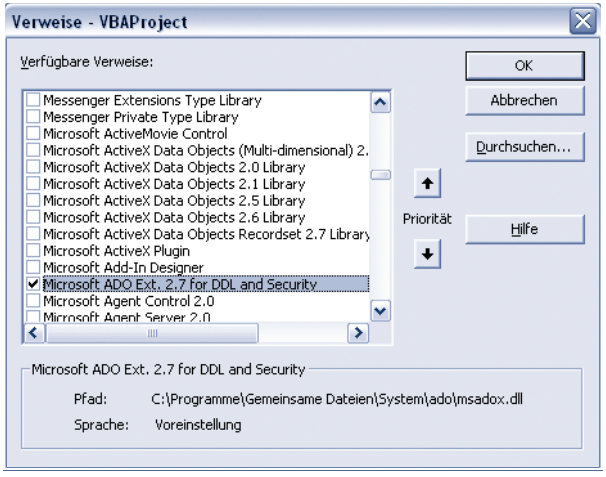


Abbildung 4.1
Verweis auf MSADOX

Normalerweise erstellen Sie sich eine Access-Datenbank direkt in MS Access, Sie können dabei Assistenten benutzen und sehr komfortabel Beziehungen zwischen Tabellen herstellen. Bei größeren Datenbankstrukturen ist das sicherlich der einzig vernünftige Weg.

Es ist aber auch mit ADOX möglich, Datenbanken zu erstellen, Tabellen und Felder hinzuzufügen, zu löschen und zu verwalten, Primärschlüssel anzulegen und noch sehr vieles mehr. Auf die meisten Themen im Zusammenhang mit

ADOX kann aber in diesem Buch nicht näher eingegangen werden, wie Sie auf die Schnelle eine Datenbank mit einer Tabelle anlegen, will ich Ihnen mit folgendem Beispiel (Listing 4.1) aber nicht vorenthalten.

Listing 4.1
Beispiele\04_Datenbanken\04_01_ADO.xls\mdlADOX

```
Public Sub CreateNewAccessCatalog()
   Dim adoxCatalog      As New ADOX.Catalog
   Dim adoxTable        As ADOX.Table
   Dim adoxColumn       As ADOX.Column
   Dim adoxKey          As ADOX.Key
   Dim strProviderADOX  As String

   Dim adoConnection    As New ADODB.Connection
   Dim adoRecordset     As New ADODB.Recordset

   Dim strTable         As String
   Dim strFile          As String
   Dim objSource        As Worksheet
   Dim lngRow           As Long

   ' Dateiort und Name der Zieldatei
   strFile = ActiveWorkbook.Path & "\myADOX.mdb"
   strTable = "myFirstTable"
   Set objSource = Worksheets("Quelldaten")

   ' Access 2000 Datenbank erzeugen
   strProviderADOX = "Provider=Microsoft.Jet.OLEDB.4.0;"

   ' Access 97 Datenbank erzeugen
   ' strProviderADOX = "Provider=Microsoft.Jet.OLEDB.4.0;" _
   '   & "Jet OLEDB:Engine Type=4;"

   If Dir(strFile) <> "" Then Kill strFile
   adoxCatalog.Create strProviderADOX & "Data Source=" & strFile
   adoxCatalog.ActiveConnection = strProviderADOX & _
      "Data Source=" & strFile

   ' Neue Tabelle erzeugen
   Set adoxTable = New ADOX.Table
   adoxTable.Name = strTable
   adoxCatalog.Tables.Append adoxTable

   With adoxTable

      ' Ein neues Feld anlegen, einmal so:
      Set adoxColumn = New ADOX.Column
      With adoxColumn
         .Name = "Monat"
         .Type = adVarWChar
         .DefinedSize = 10
         .SortOrder = adSortDescending
      End With
      .Columns.Append adoxColumn

      ' Und zweimal so:
      .Columns.Append "Wert", adDouble
      .Columns.Append "Geburtstag", adDate
   End With
```

```
    ' Etwas Zeit lassen, bevor Datenbank gefüllt wird
    Application.Wait Now + TimeValue("0:00:01")

    ' Verbindung zur Datenbank Access herstellen
    adoConnection.Provider = "Microsoft.Jet.OLEDB.4.0"

    adoConnection.Open _
      "Data Source=" & strFile, UserId:="", Password:=""

    ' Datenbank auslesen
    adoRecordset.Open "SELECT * FROM " & strTable, _
       adoConnection, adOpenKeyset, adLockOptimistic

    ' Mit Daten aus Blatt Quelldaten füllen
    With adoRecordset
        ' Bis zur letzten Reihe in Spalte 1
       For lngRow = 2 To objSource.Cells(65536, 1).End(xlUp).Row
          .AddNew
          .Fields(0) = objSource.Cells(lngRow, 1)
          .Fields(1) = objSource.Cells(lngRow, 2)
          .Fields(2) = objSource.Cells(lngRow, 3)
          .Update
       Next
    End With
End Sub
```

Listing 4.1 (Forts.)
Beispiele\04_Datenbanken\
04_01_ADO.xls\mdlADOX

4.4 ADO (ActiveX Data Objects)

Die Grundlage des Datenbankzugriffs bilden heutzutage die *Microsoft ActiveX Data Objects*. Damit können Clientanwendungen – in diesem Fall Excel – auf Daten eines Datenbankservers zugreifen und diese sogar verändern. *ADO* ist recht einfach zu bedienen, schnell und verbraucht wenig Speicherplatz. ADO ist auf nahezu allen Rechnern vorhanden, Sie brauchen also meistens nichts nachinstallieren.

Um ADO zu benutzen, können Sie in der VBE unter EXTRAS | VERWEISE einen Verweis auf die `Microsoft ActiveX Data Objects` (`MSADO`) setzen.

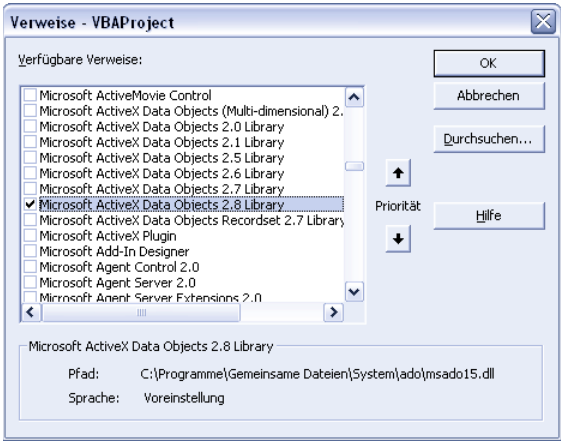


Abbildung 4.2
Verweis auf MSADO

Ein Verweis hat den Vorteil, dass Excel über eine Typbibliothek die Schnittstellen und vordefinierten Konstanten im Voraus kennt. Deshalb funktioniert auch Intellisense. Außerdem erhöht sich theoretisch auch die Ausführungsgeschwindigkeit. Das nennt man eine frühe Bindung (`Early Binding`).

Es ist aber auch die späte Bindung (`Late Binding`) möglich, dabei wird mit `CreateObject` ein Zugriff auf die gleichen Funktionalitäten ermöglicht. Die vordefinierten Konstanten sind dabei aber nicht verfügbar, im Fall der späten Bindung müssen diese selbst angelegt oder es müssen die entsprechenden Werte (meistens Longwerte) direkt benutzt werden.

Aber auch die späte Bindung bietet unter Umständen Vorteile. Wenn Sie Ihre Mappe häufig weitergeben, kann es sein, dass auf manchen Rechnern bestimmte Verweise nicht funktionieren. So haben Sie beispielsweise einen Verweis auf `MSADO 2.7` gesetzt, auf dem Zielrechner ist diese Version aber nicht vorhanden. Das führt dann zu einem Fehler.

In vielen Fällen verfügen aber auch die Vorgängerversionen über die gleichen Schnittstellen und funktionieren sogar mit den gleichen Datenbanken auf die gleiche Weise. Benutzen Sie `Late Binding`, wird die Registry unter *HKEY_LOCAL_MACHINE\Software\Classes* nach dem gewünschten Objekt durchsucht.

Wollen Sie beispielsweise ein Connection-Objekt benutzen, sollten Sie mit `CreateObject` nach dem Objekt `ADODB.Connection` suchen. Unter diesem Eintrag ist die `CLSID` der jeweils neusten Version auf dem aktuellen Rechner eingetragen. Wollen Sie eine andere Version verwenden, muss diese sich in dem Registryzweig `Classes` mit einem eigenen Eintrag verewigt haben, beispielsweise unter dem Schlüssel `ADODB.Connection.2.1`. Ist die Bibliothek verfügbar und registriert, können Sie auch diese benutzen.

Oft ist es von Vorteil, wenn Sie beim Entwickeln `Early Binding` einsetzen und später den Verweis löschen und mit `CreateObject` und `Late Binding` weiterarbeiten. So kombinieren Sie die Vorteile von Intellisense mit der Unabhängigkeit von Verweisen.

4.4.1 Das Connection-Objekt

Um eine Verbindung zu einer Datenbank herzustellen, benötigen Sie im Allgemeinen ein `Connection`-Objekt. Das können Sie zwar umgehen, indem Sie in der `Open`-Methode des `Recordset`-Objekts einen Connectionstring benutzen, das macht den Code aber schwerer verständlich und sollte deshalb vermieden werden. Außerdem haben Sie bei einem `Connection`-Objekt die Möglichkeit, schon beim Verbinden ein mögliches Problem zu erkennen.

Um solch ein Objekt zu erhalten, können Sie, wenn ein Verweis auf `MSADO` gesetzt ist, bei der Deklaration eine Objektvariable mit diesem Typ deklarieren. Anschließend wird mit `Set` und `New` das Objekt tatsächlich angelegt.

```
Dim myConnection As ADODB.Connection
Set myConnection = New ADODB.Connection
```

Wie auch beim Anlegen von anderen Objekten können Sie bei der Deklaration auch gleich New benutzen, dann wird das Objekt aber erst beim ersten Zugriff darauf tatsächlich angelegt.

```
Dim myConnection As New ADODB.Connection
```

Beim Late Binding wird die Methode CreateObject eingesetzt:

```
Dim myConnection As Object
Set adoConnection = CreateObject("ADODB.Connection")
```

Dieses Connection-Objekt wird anschließend durch die Open-Methode mit einer Datenquelle verbunden und stellt dann eine geöffnete Verbindung zu einer Datenquelle dar.

Open-Methode

Mit der Open-Methode wird die Verbindung zu einer Datenbank hergestellt. Bei dieser Methode können Sie verschiedene Parameter mit übergeben. Der Aufbau der Methode Open ist folgender:

```
myConnection.Open ConnectionString, UserID, Password, Options
```

- Connectionstring

  Der wichtigste Parameter ist der Connectionstring. Dieser enthält verschiedene Verbindungsinformationen als String. In diesem String kann der Provider angegeben werden, das ist bei Access-Datenbanken der Provider Microsoft.Jet.OLEDB.4.0. Dieser Teil des Connectionstrings würde dann so aussehen:

  ```
  Provider=Microsoft.Jet.OLEDB.4.0
  ```

  Eine häufige Fehlerquelle ist die, dass es bei älteren Versionen von ADO die Version 4.0 des OLEDB-Treibers noch nicht gibt, weshalb man im Fehlerfall die Version 3.51 benutzen muss. Dann haben Sie aber keinen Zugriff mehr auf neuere Datenbanken. Als Ausweg laden Sie sich dann bei Microsoft die neuste ADO-Version (Microsoft Data Access Components (MDAC)) herunter und installieren diese.

  Mit dem Connectionstring übergeben Sie auch die Information, wo diese Datenbank zu finden ist. Hinter die Providerinformation, abgetrennt durch ein Semikolon, können Sie die Datenquelleninformation an den String hängen. Bei einer Access-Datenbank geben Sie den Pfad inklusive dem Dateinamen in folgender Form an:

  ```
  Data Source = LW:\Verzeichnis\Dateiname.mdb
  ```

  Getrennt durch je ein Semikolon kann in den String noch die UserID und das Passwort eingefügt werden. Das sieht dann folgendermaßen aus:

  ```
  ;User ID = "UserID" ; Password = "MyPass"
  ```

Und so kann ein Connectionstring aussehen, wenn er nach dem Öffnen der Datenbank ausgelesen wird:

```
Provider=Microsoft.Jet.OLEDB.4.0;
Password="";
User ID="Admin";
Data Source=c:\nordwind2003.mdb;
Mode=Share Deny None;
Extended Properties="";
Jet OLEDB:System database="";
Jet OLEDB:Registry Path="";
Jet OLEDB:Database Password="";
Jet OLEDB:Engine Type=5;
Jet OLEDB:Database Locking Mode=1;
Jet OLEDB:Global Partial Bulk Ops=2;
Jet OLEDB:Global Bulk Transactions=1;
Jet OLEDB:New Database Password="";
Jet OLEDB:Create System Database=False;
Jet OLEDB:Encrypt Database=False;
Jet OLEDB:Don't Copy Locale on Compact=False;
Jet OLEDB:Compact Without Replica Repair=False;
Jet OLEDB:SFP=False
```

- UserID

  Bei der Open-Methode des Connection-Objekts kann optional als zweiter Parameter die UserID mit übergeben werden. Ist auch im Connectionstring die ID angegeben, wird die UserID, die im Connectionstring steht, ignoriert.

- Password

  Der dritte Parameter kann optional für das Passwort benutzt werden. Ist auch im Connectionstring das Passwort angegeben, wird die Information im Connectionstring ignoriert.

- Options

  Der vierte Parameter ist ein ConnectOptionEnum-Wert. Dieser optionale Wert legt fest, ob die Verbindung synchron oder asynchron geöffnet wird.

4.4.2 Das Recordset-Objekt

Mit einem verbundenen Connection-Objekt besitzen Sie aber noch keinen Zugriff auf die Datensätze. Erst durch die Open oder OpenSchema-Methode auf das Connection-Objekt wird der Recordset mit Daten gefüllt. In den folgenden Abschnitten werden einige interessante Methoden und Eigenschaften grob erklärt.

Open-Methode

Mit der Open-Methode wird ein Cursor geöffnet. Ein Cursor ist so etwas wie ein Positionszeiger auf einen Datensatz. Je nachdem, was bei der Verbindung zur Datenbank als CursorLocation angegeben wurde (adUseClient oder adUseServer) wird das Bewegen durch die Datensätze von der Client- oder der serverseitigen Cursorbibliothek beeinflusst. Nachfolgend die Syntax der Open-Methode:

```
Recordset.Open Source,ActiveConnection,CursorType,LockType,Options
```

- `Source`

  Das kann der Variablenname eines `Command`-Objekts, eine SQL-Anweisung, der Aufruf einer gespeicherten Prozedur oder der Dateiname eines permanenten Recordsets sein.

- `ActiveConnection`

  Dieser Parameter kann die Objektvariable eines `Connection`-Objektes mit einer gültigen, aktiven Verbindung zu einer Datenbank sein. Sie haben aber auch die Möglichkeit, direkt einen Connectionstring anzugeben und sparen sich dadurch ein `Connection`-Objekt. Der Weg über ein separates `Connection`-Objekt ist aber meiner Ansicht nach besser. Ihr Code wird durchsichtiger und Sie merken schon beim Verbinden mit der Datenbank, ob bereits an dieser Stelle ein Fehler aufgetreten ist.

- `CursorType`

  Sie haben die Möglichkeit, einen Cursortyp anzugeben.

  Standard ist der Typ `adOpenForwardOnly`. Damit können Sie sich nur in Vorwärtsrichtung durch die Datensätze bewegen. Wenn Sie diese Datensätze nur einmal durchlaufen wollen, ist das der richtige Typ, denn die Geschwindigkeit ist am höchsten.

  Der Cursortyp `adOpenKeyset` ähnelt einem dynamischen Cursor. Von anderen Benutzern der Datenbank aktuell hinzugefügte Datensätze werden aber nicht angezeigt. Aktuelle Datensatzänderungen anderer Benutzer werden dagegen angezeigt.

  Der dynamische Cursortyp `adOpenDynamic` gestattet es, dass vorgenommene Änderungen, Hinzufügungen und Löschvorgänge durch andere angezeigt werden. Der Cursor kann in alle Richtungen durch die Datensätze bewegt werden.

  Der statische Cursor `adOpenStatic` erstellt eine Kopie der Datensätze. Sie bekommen quasi einen Schnappschuss der Daten zum Zeitpunkt der `Open`-Methode.

- `LockType`

  Damit wird die Art der Sperrung einzelner Datensätze festgelegt.

  Standard ist der Locktyp `adLockReadOnly`. Mit diesem Typ ist nur das Lesen von Datensätzen gestattet.

  Wird der Typ `adLockPessimistic` benutzt, wird der aktuelle Datensatz beim Bearbeiten für andere vollständig gesperrt.

  Wenn Sie den Typ `adLockOptimistic` benutzen, wird der Datensatz nur beim Aufrufen der `Update`-Methode gesperrt.

  Die optimistische Stapelaktualisierung `adLockBatchOptimistic` wird verwendet, wenn der Stapelaktualisierungsmodus benutzt wird. In diesem Modus werden die Änderungen an Datensätzen gemeinsam an die Datenbank übertragen. Erst dann werden die Datensätze zum Bearbeiten gesperrt.

- `Options`

  Dieser optionale Wert gibt an, wie der Provider das Source-Argument auswertet, wenn es sich nicht um ein `Command`-Objekt handelt.

OpenSchema-Methode

Mit der `OpenSchema`-Methode erhalten Sie Informationen über die Datenquelle. Dabei können Sie beispielsweise Informationen über die Tabellen oder die Felder in den Tabellen anfordern. Es gibt insgesamt über dreißig `QueryType`-Werte und für jeden einzelnen davon mehrere `Criteria`-Werte. Ich gehe hier aber nur auf zwei Stück ein, das sind die für den Normalbürger wohl nützlichsten Typen `adSchemaColumns` und `adSchemaTables`. Zum einen, weil ich mit den meisten anderen noch nie etwas zu tun hatte und zum anderen, weil es sonst den Rahmen des Buches sprengen würde. Es folgt die Syntax der `OpenSchema`-Methode:

```
Set Recordset= myConnection.OpenSchema (QueryType,Criteria,SchemaID)
```

- `QueryType, Criteria`

  Der Query-Typ `adSchemaColumns` liefert Informationen über die Felder einer Datenbank. Mögliche Kriterien sind `TABLE_CATALOG`, `TABLE_SCHEMA`, `TABLE_NAME` und `COLUMN_NAME`.

  Der Typ `adSchemaTables` liefert Informationen über die Tabellen einer Datenbank. Mögliche Kriterien sind `TABLE_CATALOG`, `TABLE_SCHEMA`, `TABLE_NAME` und `TABLE_TYPE`.

- `SchemaID`

  Die `SchemaID` ist ein global eindeutiger Bezeichner (`GUID`) für die Schemaabfrage eines Providerschemas. Dieser Parameter wird aber nur benötigt, wenn der `QueryType` auf `adSchemaProviderSpecific` gesetzt ist. In allen anderen Fällen wird er sowieso ignoriert. Ich persönlich habe mich noch nie damit befassen müssen.

AddNew-Methode

Diese Methode erstellt einen neuen Datensatz für ein aktualisierbares `Recordset`-Objekt. Der Datensatz wird der Datenbank hinzugefügt, wenn Sie den Datensatz wechseln oder die `Update`-Methode aufrufen. Es folgt die Syntax der `AddNew`-Methode

```
Recordset.AddNew Feldliste, Werte
```

- `Feldliste`

  Dieser optionale Parameter kann ein einzelner Feldname oder ein Array mit Feldnamen sein. Statt Feldnamen können Sie auch die Ordinalpositionen der Felder benutzen.

- `Werte`

  Dieser optionale Parameter kann ein einzelner Wert oder ein Array mit Werten sein. Wenn Sie in der Feldliste Arrays benutzen, müssen Sie für jeden Wert im Array auch einen entsprechenden Eintrag in der Feldliste haben.

Delete-Methode

Diese Methode löscht den aktuellen Datensatz oder eine Gruppe von Datensätzen.

```
Recordset.Delete AffectRecords
```

- `AffectRecords`

  Dieser Wert bestimmt, wie viele Datensätze durch die `Delete`-Methode gelöscht werden. Standard ist der Wert `adAffectCurrent`, damit wird der aktuelle Datensatz gelöscht. Wird der Wert `adAffectGroup` benutzt, löschen Sie alle Datensätze, die den aktuellen Filtereigenschaften entsprechen.

MoveFirst-, MoveLast-, Move`Next`-, MovePrevious-Methoden

Mit diesen Methoden bewegen Sie sich durch die Datensätze. Beim Cursortyp `adOpenForwardOnly` funktionieren die Methoden `MoveFirst` und `MovePrevious` nicht.

```
Recordset.{MoveFirst | MoveLast | MoveNext | MovePrevious}
```

- `MoveFirst`

  Der Cursor wird an den ersten Datensatz bewegt und dieser wird dann als der aktuelle benutzt.

- `MoveLast`

  Der Cursor wird an den letzten Datensatz bewegt und dieser wird dann als der aktuelle benutzt.

- `MoveNext`

  Der Cursor wird an den nächsten Datensatz bewegt und dieser wird dann als der aktuelle benutzt.

- `MovePrevious`

  Der Cursor wird an den vorherigen Datensatz bewegt und dieser wird dann als der aktuelle benutzt.

Update-Methode

Diese Methode speichert Änderungen an einem Datensatz für ein aktualisierbares `Recordset`-Objekt in der Originaldatenbank.

```
Recordset.Update Feldliste, Werte
```

- `Feldliste`

  Dieser optionale Parameter kann ein einzelner Feldname oder ein Array mit Feldnamen sein. Statt Feldnamen können Sie auch die Ordinalpositionen der Felder benutzen.

- `Werte`

  Dieser optionale Parameter kann ein einzelner Wert oder ein Array mit Werten sein. Wenn Sie in der Feldliste Arrays benutzen, müssen Sie für jeden Wert im Array auch einen entsprechenden Eintrag in der Feldliste haben.

UpdateBatch-Methode

Diese Methode speichert den aktuellen Datensatz oder eine Gruppe von Datensätzen in der Originaldatenbank.

```
Recordset.UpdateBatch AffectRecords
```

- `AffectRecords`

  Dieser Wert bestimmt, wie viele Datensätze durch die `UpdateBatch`-Methode gespeichert werden. Benutzen Sie den Wert `adAffectCurrent`, wird nur der aktuelle Datensatz gespeichert. Wird der Wert `adAffectGroup` benutzt, speichern Sie alle Datensätze, die den aktuellen Filtereigenschaften entsprechen. Standard ist der Wert `adAffectAll`, damit werden alle geänderten Datensätze gespeichert.

Filtereigenschaft

Die Filtereigenschaft ermöglicht es, Datensätze eines Recordsets, die nicht den Filterkriterien entsprechen, auszublenden.

```
Recordset.Filter=Kriteriun
```

Das Kriterium kann eine Zeichenfolge sein, die aus einer oder mehreren einzelnen Klauseln besteht. Diese können durch die Operatoren `AND` oder `OR` verkettet sein. Als Wert ist auch eine Konstante möglich.

Benutzen Sie die Konstante `adFilterNone`, wird der aktuelle Filter entfernt und es sind wieder alle Datensätze des Recordsets verfügbar.

Im Stapelaktualisierungsmodus bewirkt `adFilterPendingRecords`, dass nur die Datensätze angezeigt werden, die sich geändert haben, aber noch nicht an den Server gesendet wurden.

Die Konstante `adFilterAffectedRecords` lässt nur zu, dass die Datensätze angezeigt werden, die von dem letzten `Delete`-, `Resync`-, `UpdateBatch`- oder `CancelBatch`-Aufruf betroffen sind.

Wenn nur die Datensätze im aktuellen Zwischenspeicher angezeigt werden sollen – das sind die Ergebnisse des letzten Abrufs von Datensätzen –, benutzen Sie `adFilterFetchedRecords`.

Wollen Sie nur die Datensätze angezeigt bekommen, die bei der letzten Stapelaktualisierung nicht verarbeitet werden konnten, verwenden Sie `adFilterConflictingRecords`.

Beim Bewegen durch die Datensätze sind nur die Sätze verfügbar, die den Filterkriterien entsprechen.

Ein Filterstring ist prinzipiell folgendermaßen aufgebaut:

```
Feldname-Operator-Wert
```

- `Feldname`

  Der Feldname muss gültig sein. Falls er Leerstellen enthält, muss der Name in eckige Klammern eingeschlossen werden.

- `Operator`

  Folgende Operatoren sind möglich: `=`, `LIKE`, `<>`, `<`, `<=`, `>`, `>=`

- `Wert`

  Wert kann ein Vergleichswert oder eine Zeichenfolge sein. Eine Zeichenfolge wird in einfache Anführungszeichen eingeschlossen. Wenn `LIKE` benutzt wird, sind auch die Platzhalter `%` und `*` erlaubt.

- `Verknüpfungsoperatoren`

  Die Operatoren `AND` und `OR` sind gleichwertig. Klauseln können Sie in Klammern zusammenfassen. Es funktioniert aber nicht, mehrere mit `OR` verknüpfte Klauseln in einer Klammer zusammenzufassen und das Ergebnis mit dem einer anderen Klausel über den Operator `AND` zu verbinden.

BOF- oder EOF-Eigenschaft

Wenn Sie beim Wechseln der Datensätze die Grenzen des Recordsets überschreiten, wird eine der beiden Eigenschaften wahr. `BOF` wird `True`, wenn Sie sich vor dem ersten und `EOF` wird `True`, wenn Sie sich hinter dem letzten Datensatz befinden. Wird ein `Recordset`-Objekt geöffnet, das keine Datensätze enthält, werden beide Eigenschaften auf den Wert `True` gesetzt.

GetRows

Die `GetRows`-Methode ist dafür gedacht, Datensätze aus einem `Recordset` in ein zweidimensionales Array zu kopieren. Der erste Index kennzeichnet das Feld und der zweite die Datensatznummer.

```
Array = Recordset.GetRows( Rows, Start, Fields )
```

- `Rows`

  Mit diesem optionalen Parameter kann die Anzahl der Datensätze festgelegt werden, die in das Array kopiert werden sollen. Wird nichts oder die Konstante `adGetRowsRest (-1)` übergeben, werden alle verfügbaren Datensätze kopiert.

- `Start`

  Mit diesem optionalen Parameter kann der Startpunkt in Lesezeichen festgelegt werden, ab dem Daten kopiert werden sollen. Wenn vom Provider keine Lesezeichen unterstützt werden, bringt die Verwendung dieses Parameters einen Laufzeitfehler.

- `Fields`

  Mit diesem optionalen Parameter können die Felder festgelegt werden, die in das Array kopiert werden. Dabei können Sie den Index oder den Namen eines Feldes angeben. Sie können auch ein Array von Feldern übergeben, diese Felder werden dann mit in das Ergebnisarray kopiert. Das übergebene Array (2, 1, 3) liefert die Felder zwei, eins und drei. Die Indizierung beginnt mit dem Feld null.

Die Zeile

```
varRecord = adoRecordset.GetRows(2, 1, Array(0, 1, 3))
```

kopiert in den Variant `varRecord` zwei Datensätze ab Lesezeichen eins, dabei werden die Felder null, eins und drei benutzt.

GetString

Die `GetString`-Methode ist dafür gedacht, Datensätze aus einem Recordset in eine Zeichenfolge zu kopieren.

```
Variant = Recordset.GetString(StringFormat, NumRows, _
ColumnDelimiter, RowDelimiter, NullExpr)
```

- `StringFormat`

  Dieser Parameter gibt das Format an. Sollte immer `adClipString` sein, wenn die Parameter `ColumnDelimiter`, `RowDelimiter` und `NullExpr` benutzt werden.

- `NumRows`

  Mit diesem optionalen Parameter kann die Anzahl der Datensätze festgelegt werden, die in den String kopiert werden sollen. Wird nichts übergeben, werden alle verfügbaren Datensätze kopiert.

- `ColumnDelimiter`

  Mit diesem optionalen Parameter können Sie die Trennzeichen zwischen den Feldern festlegen. Wird nichts angegeben, wird das Tabulatorzeichen benutzt.

- `RowDelimiter`

  Mit diesem optionalen Parameter können Sie die Trennzeichen zwischen den Datensätzen festlegen. Wird nichts angegeben, wird das Wagenrücklaufzeichen benutzt.

- `NullExpr`

  Mit diesem optionalen Parameter legen Sie die Zeichenfolge für leere Felder fest. Wird nichts angegeben, wird ein leerer String benutzt.

Mithilfe dieser Methode können Sie sehr schön Datenbankabfragen als *.CSV*-Dateien (Comma Separated Values) speichern. In Deutschland werden die Daten durch ein Semikolon separiert, da dort das Komma als Dezimaltrennzeichen dient, in anderen Ländern durch ein Komma. Mit der `GetString`-Methode können Sie das oder die Trennzeichen frei wählen. Auch die Trennzeichen der einzelnen Datensätze können beliebig festgelegt werden. Außerdem können Sie für leere Felder einen beliebigen String vorgeben.

In manchen Fällen können Sie aber immer noch Probleme mit den Ländereinstellungen bekommen, beispielsweise bei der Darstellung von Datums-, Zeit- und Zahlenformaten. Wenn beispielsweise Dateien mit dem Trennzeichen Komma, mit dem Tausendertrennzeichen Hochkomma und dem Dezimaltrennzeichen Punkt erzeugt werden sollen, bekommen Sie in Deutschland ohne die

Änderung der Ländereinstellung Schwierigkeiten. Mit etwas Fantasie können Sie aber auch diese Hürde nehmen.

Die einfachste Möglichkeit der Zahlenkonvertierung besteht wohl darin, den erzeugten Text mit der `Replace`-Funktion zu bearbeiten. Dazu benutzen Sie erst einmal eine andere, eindeutige identifizierbare Zeichenfolge als Feldtrenner. Sollen anschließend Kommas durch Punkte und Punkte durch Hochkommas ersetzt werden, tauschen Sie erst einmal die Punkte durch eindeutige Zeichenfolgen. Jetzt können Sie mit `Replace` die Kommas in Punkte und die eindeutige Zeichenfolge, die die Punkte maskiert, in Hochkommas umwandeln. Bleibt nur noch, den maskierten Feldtrenner in die gewünschte Zeichenfolge zu verwandeln.

Zumindest unter Access können Sie aber schon beim Erzeugen des Recordsets die Daten in das gewünschte Zahlenformat verwandeln. Dazu können Sie im SQL-String die `Format`-Funktion benutzen und kann damit beliebige Datums- und Zeitformate erzeugen. Leider funktioniert die `Replace`-Funktion in SQL-Abfragen nicht.

Sie können aber einen Wert trotzdem so auseinander nehmen und ihn als Text wieder zusammensetzen, dass Sie schon beim Erzeugen des Recordsets das gewünschte Textformat erhalten. Hier eine SQL-Abfrage, mit der Sie einen Wert unter den Ländereinstellungen mit deutschen Zahlenformaten in einen String mit dem Dezimaltrennzeichen Punkt bekommen. Das umzuwandelnde Feld hat hier den Feldnamen `Wert`:

```
adoRecordset.Open _
   "SELECT " & _
   "cstr(FIX(Wert))& '.' & " & _
   "format( " & _
   "ABS(Wert-FIX(Wert))" & _
   "*100,'00')" & _
   " FROM [" & strRange & "];", _
   adoConnection, _
   adOpenKeyset, _
   adLockOptimistic
```

Dabei wird der Ganzzahlenanteil, der mit der `Fix`-Funktion erzeugt wurde, in einen String umgewandelt. An diesen wird ein Punkt und der Nachkommaanteil als Absolutwert gehängt. Der Nachkommaanteil wird vorher durch die Multiplikation mit 100 und der `Format`-Funktion in eine zweistellige Ganzzahl verwandelt.

Das folgende Beispiel (Listing 4.2) zeigt, wie Sie einen bestimmten Bereich eines Tabellenblattes mit einer Abfrage auslesen und diesen erzeugten Recordset anschließend als *.csv*-Datei speichern. Leider wird die erste Zeile nicht beachtet, da dort von ADO die Feldnamen erwartet werden.

Dazu wird dem benutzten Bereich eines Tabellenblatts ein Name zugewiesen. Dann wird der Pfad der aktuellen Arbeitsmappe ermittelt und es wird eine ADO-Verbindung hergestellt. Daraufhin wird ein Recordset aus dem benannten Bereich erzeugt. Mit der `GetString`-Methode erzeugen Sie anschließend einen String mit einem Semikolon als Trennzeichen und einen Zeilenvorschub und

Wagenrücklauf (vbCrLf) als Datensatztrenner. Dieser String wird dann als Datei mit der Endung *.csv* im Verzeichnis der aktuellen Mappe gespeichert. Das funktioniert natürlich auch mit anderen Datenbanken.

Listing 4.2
Beispiele\04_Datenbanken\
04_01_ADO.xls\mdlExportCSV

```
'Verweis auf Microsoft ActiveX Data Objects
Sub ExportCSV()
   Dim adoConnection    As New ADODB.Connection
   Dim adoRecordset     As New ADODB.Recordset

   Dim strResult        As String
   Dim strFile          As String
   Dim strAddress       As String
   Dim strRange         As String

   ' Dateiort und Name der Quelldatei
   strFile = ActiveWorkbook.FullName

   ' Bereichsname
   strRange = "MyRange"

   ' Bereichsname bezieht sich auf diesen Range
   ' Hier der UsedRange
   strAddress = "=Quelldaten!" & _
      Worksheets("Quelldaten").UsedRange.Address( _
      ReferenceStyle:=xlR1C1)

   ' Benannten Bereich setzen
   ActiveWorkbook.Names.Add Name:="MyRange", _
      RefersToR1C1:=strAddress

   ' Verbinden mit eigener Datei
   adoConnection.Open _
      "Provider=Microsoft.Jet.OLEDB.4.0;" & _
      "Data Source=" & strFile & _
      ";Extended Properties=Excel 8.0;"

   ' Alles vom Bereich "MyRange" in den Recordset
   adoRecordset.Open _
      "SELECT * FROM [" & strRange & "];", _
      adoConnection, _
      adOpenKeyset, _
      adLockOptimistic

   ' Den Inhalt des Recordsets in einen String bringen.
   ' Feldtrenner ist ";"
   ' Trennung zwischen den Sätzen ist CrLf (chr(13)& chr(10))
   ' Nullfelder werden mit einem Leerstring gefüllt
   strResult = adoRecordset.GetString( _
      adClipString, , ";", vbCrLf, "")

   ' Zielname festlegen
   strFile = ActiveWorkbook.Path & "\" & "Export.csv"

   ' Eventuell vorhandene Datei löschen
   Kill strFile
```

Listing 4.2 (Forts.)
Beispiele\04_Datenbanken\
04_01_ADO.xls\mdlExportCSV

```
    ' Ausgabe in Datei
    Open strFile For Binary As 1
       Put 1, , strResult
    Close

    ' Schließen
    adoRecordset.Close
    adoConnection.Close

End Sub
```

RecordCount-Eigenschaft

Diese Eigenschaft gibt die aktuelle Anzahl von Datensätzen zurück. Kann diese nicht bestimmt werden, wird der Wert -1 zurückgegeben.

4.5 SQL

In diesem Abschnitt behandele ich ein paar der gebräuchlichsten SQL-Argumente. Ich habe aber bewusst stark vereinfacht und viele Möglichkeiten werden erst gar nicht behandelt. `Aliase` und `Joins` habe ich zum Beispiel ganz weggelassen. Die Dialekte sind aber auch abhängig von der jeweiligen Datenbank und können von dem hier beschriebenen abweichen.

```
SELECT [ALL | DISTINCT] [TOP Anzahl [PERCENT]]
FROM Table
[WHERE Bedingung [AND | OR Bedingung]]
[GROUP BY Feld [, Feld ...]]
[HAVING Bedingung]
[ORDER BY Feld [ASC | DESC] [,Feld [ASC | DESC] ...]]
```

- `SELECT`

  Hier werden die Felder angegeben, die in dem Abfrageergebnis vorkommen sollen. Wird ein Stern benutzt, werden alle Felder angezeigt.

  An dieser Stelle können auch Rechenoperationen oder Aggregatfunktionen wie SUM (Summe), MIN (Minimum), MAX (Maximum), AVG (Mittelwert) und COUNT (Anzahl) benutzt werden.

- `ALL`

  Alle Datensätze werden angezeigt. Ist Standard, braucht also nicht angegeben werden.

- `DISTINCT`

  Gleiche Datensätze werden nur einmal angezeigt.

- `TOP` Anzahl [PERCENT]

  Es werden nur so viele Datensätze in der Reihenfolge der ORDER BY-Klausel angezeigt wie angegeben werden. Wird PERCENT benutzt, gilt die Angabe als auf eine ganze Zahl aufgerundeter Prozentwert.

- `FROM`

  Hier werden die Tabellen aufgelistet, die die abzurufenden Daten enthalten. Tabellennamen mit Leerzeichen werden in eckige Klammern gesetzt.

- `WHERE`

  Hier werden die Bedingungen festgelegt, die die Datensätze erfüllen müssen, damit sie in das Abfrageergebnis aufgenommen werden. Mit den Operatoren `OR`, `AND` und `NOT` können die Bedingungen kombiniert werden. Als Vergleichsoperatoren können `=`, `LIKE`, `<>`, `<`, `<=`, `>`, `>=` benutzt werden. Platzhalter für beliebig viele Zeichen ist in SQL-Abfragen das Prozentzeichen `%`.

- `GROUP BY` Feld [,Feld...]

  Gruppiert das Abfrageergebnis entsprechend den Werten einer oder mehrerer Felder.

- `HAVING` Bedingung

  Gibt eine Filterbedingung an. `HAVING` sollte mit `GROUP BY` verwendet werden. Sie können beliebig viele Filterbedingungen mit den Operatoren `OR`, `AND` und `NOT` kombinieren.

- `ORDER BY` Feld

  Damit wird das Abfrageergebnis entsprechend den Daten sortiert, die sich in den angegebenen Feldern befinden.

  `ASC` ist Standard, braucht somit nicht angegeben werden und gibt eine aufsteigende Reihenfolge an.

  `DESC` gibt eine absteigende Reihenfolge an.

4.6 Excel-Tabellen

Mit ADO ist es möglich, auf Daten einer Excel-Tabelle zuzugreifen. Dabei ist es aber nicht erforderlich, dass auf Ihrem Rechner Excel installiert ist. Sogar mit einer Scriptdatei können Sie auf Exceldateien zugreifen und unter Umständen sogar Daten verändern. Ein paar Einschränkungen gibt es allerdings und es müssen auch noch einige andere Voraussetzungen erfüllt sein.

Zunächst einmal muss die Excel-Tabelle wie eine Datenbank aufgebaut sein. Dazu ist es notwendig, dass ein rechteckiger Bereich mit Feldnamen als Überschrift existiert, damit so etwas wie eine Datenbankstruktur vorliegt. Am einfachsten ist es, wenn Sie Bereichsnamen definieren, diese Namen können Sie bei einer SQL-Abfrage als Tabellennamen benutzen. Sie benötigen dann auch keine Überschriften, steht in der ersten Zeile eine Zahl oder befindet sich darin eine leere Zelle, vergibt ADO Feldnamen in der Form »`F`« + `Feldindex`.

In den Spalten sollten auch nur gleiche Datentypen stehen, also beispielsweise nur Strings oder Werte. Nach meinen Tests legt der erste Datensatz das Format fest. Im Recordset selbst bewirken abweichende Datentypen oder leere Zellen ein leeres Feld im Datensatz. Sie können auch nur Daten gleichen Typs zu einem Recordset hinzufügen.

ADO verwaltet nur maximal 255 Felder, deshalb dürfen Sie auch nur bis zu 255 Spalten in den benannten Bereich mit einschließen. Im Fall des benannten Bereichs benötigen Sie aber noch nicht einmal eine Überschrift. Schließen Sie

beispielsweise die ersten 255 Spalten komplett in diesen Bereich mit ein, können Sie im Prinzip auf jede Zelle dieses Bereichs zugreifen.

Dazu muss nur die Zelladresse so umgerechnet werden, dass die Zeile auf den richtigen Datensatz und die Spalte auf das entsprechende Feld zeigt. Die Zelle A2 wäre in diesem Fall der Datensatz Nummer null, da die Zeile eins für die Feldnamen reserviert ist. Da es sich um die erste Spalte handelt, müssen Sie im Datensatz null auf das Feld null zugreifen.

Diese Werte können verändert und auch mit der `Update`-Methode des `Recordset`-Objekts zurückgeschrieben werden, wenn es sich bei den Daten um den für dieses Feld festgelegten Datentyp handelt.

Um auch an die Werte der Zeile eins zu kommen, müssen Sie die Feldnamen auslesen, die ja dem Inhalt der Zeile eins entsprechen, aber auch nur, wenn sich darin Text befindet, ansonsten übernimmt ADO eigenmächtig die Namensvergabe. Die Feldnamen, somit auch die Daten der Zeile eins, können nicht geändert werden.

Folgendermaßen stellen Sie eine Verbindung zu einer Excel-Datei her, wobei `adoConnection` ein `ADODB.Connection`-Objekt ist:

```
adoConnection.Open _
   "Provider=Microsoft.Jet.OLEDB.4.0;" & _
   "Data Source=" & strFile & _
   ";Extended Properties=Excel 8.0;"
```

Mit dem folgenden Beispiel (Listing 4.3) können Sie per ADO auf Excel-Daten zugreifen. Der Zellinhalt von Zelle `A3` wird nach der Abfrage in einer Messagebox ausgegeben und anschließend wird der Zellinhalt in die Zeichenfolge »B23« geändert. Dazu ist aber eine gewisse Vorarbeit notwendig.

Die Tabelle, aus der Daten ausgelesen werden sollen, muss einen benannten Bereich mit dem Namen `MeinBereich` besitzen. In diesem Bereich müssen alle Spalten enthalten sein, auf die irgendwann einmal zugegriffen werden soll, und zwar angefangen bei der Spalte 1 bis zur letzten benutzten Spalte.

Es dürfen aber nur maximal 255 Spalten enthalten sein, die letzte Spalte muss deshalb davon ausgenommen werden. Bei Bedarf können Sie sich ja noch zusätzlich einen Bereich anlegen, der diese Spalte mit einschließt. Es muss anschließend die Quelladresse nur anders in Datensatz und Feld umgerechnet werden.

Listing 4.3
Beispiele\04_Datenbanken\04_01_ADO.xls\mdlXLS

```
' Verweis auf Microsoft ActiveX Data Objects
Sub XLAbfrage()
   Dim adoConnection    As New ADODB.Connection
   Dim adoRecordset     As New ADODB.Recordset

   Dim lngRow           As Long
   Dim lngColumn        As Long

   Dim strResult        As String
   Dim strFile          As String
```

Listing 4.3 (Forts.)
Beispiele\04_Datenbanken\
04_01_ADO.xls\mdlXLS

```
Dim strAddress       As String
Dim strRange         As String
Dim rngSourceAddress As Range

' Dateiort und Name
strFile = ActiveWorkbook.FullName

' Quellzelle
Set rngSourceAddress = Application.InputBox( _
   prompt:="Zelle zur Abfrage wählen", Type:=8)
strAddress = rngSourceAddress.Address(0, 0)

' Vorher festgelegter, benannter Bereich
' Spalte 1-255
strRange = "MeinBereich"

With Worksheets(1).Range(strAddress)

   'Zeile 1 ist Datensatz null
   lngRow = .Row - 2

   'Spalte 1 ist Feld null
   lngColumn = .Column - 1

End With

adoConnection.Open _
   "Provider=Microsoft.Jet.OLEDB.4.0;" & _
   "Data Source=" & strFile & _
   ";Extended Properties=Excel 8.0;"

adoRecordset.Open _
   "SELECT * FROM [" & strRange & "];", _
   adoConnection, _
   adOpenKeyset, _
   adLockOptimistic

On Error Resume Next

If lngRow = -1 Then

   'Zeile 1 kann nur gelesen werden, da diese
   'Zellen als Feldnamen behandelt werden
   strResult = adoRecordset.Fields(lngColumn).Name

Else

   'Zum entsprechenden Datensatz gehen
   adoRecordset.MoveFirst
   adoRecordset.Move lngRow

   'Das entsprechende Feld auslesen
   strResult = adoRecordset.Fields(lngColumn)

   MsgBox strResult

End If
```

Listing 4.3 (Forts.)
Beispiele\04_Datenbanken\04_01_ADO.xls\mdlXLS

```
    'Zum Schreiben in die Datei
    adoRecordset.Fields(lngColumn) = "Zelle " & strAddress
    adoRecordset.Update

    adoRecordset.Close
    adoConnection.Close

End Sub
```

Ähnliches funktioniert auch als VB Script. Da bei Scriptdateien aber keine Verweise gesetzt werden können und somit keine Typenbibliotheken der benutzten Objekte zur Verfügung stehen, sind auch die verwendeten Konstanten unbekannt. Statt die reinen Werte zu benutzen, legen Sie sich am besten die Konstanten mit den entsprechenden Werten an.

Aus der Zelladresse den richtigen Datensatz und das benötigte Feld zu extrahieren erfordert einen höheren Aufwand als unter Excel-VBA. Dafür sind die zwei kleinen Funktionen `AddressToColumn` und `AddressToRow` zuständig.

Speichern Sie die Textdatei mit der Endung *.vbs* und führen diese durch einen Doppelklick aus. Ein installierter Script Host wird dabei vorausgesetzt.

Listing 4.4
Beispiele\04_Datenbanken\04_01_ADO.xls\mdlXLSScript

```
Dim strResult
Dim strFile
Dim strAddress
Dim lngRow
Dim lngColumn
Dim strRange

Const adOpenDynamic = 2
Const adOpenForwardOnly = 0
Const adOpenKeyset = 1
Const adOpenStatic = 3

Const adLockBatchOptimistic = 4
Const adLockOptimistic = 3
Const adLockPessimistic = 2
Const adLockReadOnly = 1

Set adoConnection = CreateObject("ADODB.Connection")
Set adoRecordset = CreateObject("ADODB.Recordset")

Call Extract

Sub Extract()

   strFile = "c:\XL_ADO.xls"
   strAddress = "A3"
   strRange = "MeinBereich"

   'Zeile 2 ist Datensatz null
   lngRow = AddressToRow(strAddress) - 2

   'Spalte 1 ist Feld null
   lngColumn = AddressToColumn(strAddress) - 1
```

Listing 4.4 (Forts.)
Beispiele\04_Datenbanken\
04_01_ADO.xls\mdlXLSScript

```
    adoConnection.Open _
       "Provider=Microsoft.Jet.OLEDB.4.0;" & _
       "Data Source=" & strFile & _
       ";Extended Properties=Excel 8.0;"

    adoRecordset.Open _
       "SELECT * FROM [" & strRange & "];", _
       adoConnection, _
       adOpenKeyset, _
       adLockOptimistic

    On Error Resume Next
    If lngRow = -1 Then
       'Zeile 1 kann nur gelesen werden, da diese
       'Zellen als Feldnamen behandelt werden
       strResult = adoRecordset.Fields(lngColumn).Name
    Else
       'Zum entsprechenden Datensatz gehen
       adoRecordset.MoveFirst
       adoRecordset.Move lngRow
       'Das entsprechende Feld auslesen
       strResult = adoRecordset.Fields(lngColumn)
    End If

    MsgBox strResult

    adoRecordset.Close
    adoConnection.Close

End Sub

Function AddressToRow(myAddress)
    Dim Pos
    Do
       Pos = Pos + 1
       If InStr(1, "0123456789", Mid(myAddress, Pos, 1)) > 0 Then
          AddressToRow = Mid(myAddress, Pos, 2)
          Exit Do
       End If
    Loop While Pos < Len(myAddress)
End Function

Function AddressToColumn(myAddress)
    Dim Pos
    Do
       Pos = Pos + 1
       If InStr(1, "0123456789", Mid(myAddress, Pos, 1)) > 0 Then
          AddressToColumn = _
             Asc(UCase(Mid(myAddress, Pos - 1, 1))) - 64
          If Pos = 3 Then _
             AddressToColumn = AddressToColumn + _
             (Asc(UCase(Mid(myAddress, 1, 1))) - 64) * 26
          Exit Do
       End If
    Loop While Pos < Len(myAddress)
End Function
```

4.7 Access-Datenbanken

Mit diesem Beispiel (Listing 4.5) können Sie auf Access-Daten zugreifen. In der ersten Abfrage werden die Tabellen der Datenbank mit der Methode `OpenSchema` eines Recordsets ausgelesen und in einem folgenden Dialog können Sie eine davon auswählen.

In einer zweiten Abfrage holen Sie sich die Feldnamen und auch hier können Sie ein Feld auswählen. Da in dem fertigen SQL-String der Operator `LIKE` benutzt wird, ist es nur sinnvoll, Felder auszuwählen, die auch Text enthalten.

Anschließend kann in einer Inputbox ein Suchkriterium eingegeben werden, nach dem in dem ausgewählten Feld mit dem Operator `LIKE` gesucht wird. Datensätze, die den Kriterien entsprechen, werden ausgewählt und in einem Tabellenblatt dargestellt.

Listing 4.5
Beispiele\04_Datenbanken\04_01_ADO.xls\mdlMdb

```
Public Sub MDB_Search( _
   Optional strFile As String = "c:\nordwind.mdb", _
   Optional strWorksheet As String = "Tabelle1")

Dim strSearch        As String
Dim strQuestion      As String
Dim objField         As Object
Dim strRet           As String

Dim lngColumn        As Long
Dim lngRow           As Long

Dim strField         As String
Dim astrFields()     As String
Dim lngFields        As Long

Dim strTable         As String
Dim astrTables()     As String
Dim lngTables        As Long

' Ohne Verweis die nächsten drei Zeilen auskommentieren
Dim adoConnection    As New ADODB.Connection
Dim adoRecordset     As New ADODB.Recordset
Dim adoSchema        As New ADODB.Recordset

' Ohne Verweis bei den nächsten 12 Zeilen
' die Kommentierung aufheben
' Dim adOpenKeyset        As Long
' Dim adLockOptimistic    As Long
' Dim adSchemaTables      As Long
' adOpenKeyset = 1
' adLockOptimistic = 3
' adSchemaTables = 20
' Dim adoConnection    As Object
' Dim adoRecordset     As Object
' Dim adoSchema        As Object
' Set adoConnection = CreateObject("ADODB.Connection")
' Set adoRecordset = CreateObject("ADODB.Recordset")
' Set adoSchema = CreateObject("ADODB.Recordset")
```

Listing 4.5 (Forts.)
Beispiele\04_Datenbanken\04_01_ADO.xls\mdlMdb

```
If Dir(strFile) = "" Then _
   MsgBox "Datei existiert nicht": Exit Sub

' Verbindung zur Datenbank herstellen
adoConnection.Provider = "Microsoft.Jet.OLEDB.4.0"
adoConnection.Open _
  "Data Source=" & strFile, _
   UserId:="", _
   Password:=""

' Tabellennamen aus Datenbank extrahieren
Set adoSchema = adoConnection.OpenSchema( _
   adSchemaTables, _
   Array(Empty, Empty, Empty, "TABLE"))

' Alle Sätze nacheinander durchlaufen
Do Until adoSchema.EOF

   ' Tabellenname holen
   strTable = adoSchema!TABLE_NAME

   ' Keine Systemtabellen
   If InStr(1, strTable, "MSys") = 0 Then

      ' Tabellen in ein Array
      lngTables = lngTables + 1
      ReDim Preserve astrTables(1 To lngTables)
      astrTables(lngTables) = strTable

      ' Tabellen als String aufbereiten
      strQuestion = strQuestion & lngTables & " = "
      strQuestion = strQuestion & strTable & vbCrLf

   End If
   ' Nächster Datensatz
   adoSchema.MoveNext
Loop
' Das Schema schließen
adoSchema.Close

Do
   ' Tabelle auswählen oder beenden
   strRet = InputBox(strQuestion, _
      "Aus welcher Tabelle sollen Werte extrahiert werden?", 1)
   If strRet = "" Then Exit Sub
   If IsNumeric(strRet) Then
      lngRow = CLng(strRet)
      If lngRow > 0 And lngRow <= lngTables Then Exit Do
   End If
Loop

' Gewählte Tabelle
strTable = astrTables(lngRow)
```

Listing 4.5 (Forts.)
Beispiele\04_Datenbanken\
04_01_ADO.xls\mdlMdb

```
' Felder auslesen
adoRecordset.Open "SELECT * FROM " & strTable, _
   adoConnection, _
   adOpenKeyset, _
   adLockOptimistic

strQuestion = ""

' Alle Felder nacheinander durchlaufen
For Each objField In adoRecordset.Fields

   ' Feldname holen
   strField = objField.Name

   ' Felder in ein Array
   lngFields = lngFields + 1
   ReDim Preserve astrFields(1 To lngFields)
   astrFields(lngFields) = strField

   ' Felder als String aufbereiten
   strQuestion = strQuestion & lngFields & " = "
   strQuestion = strQuestion & strField & vbCrLf

Next
' Den Recordset schließen
adoRecordset.Close

Do
   ' Feld auswählen oder beenden
   strRet = InputBox(strQuestion, _
      "Aus welchem Feld soll der Suchbegriff gesucht werden?", 1)
   If strRet = "" Then Exit Sub
   If IsNumeric(strRet) Then
      lngRow = CLng(strRet)
      If lngRow > 0 And lngRow <= lngFields Then Exit Do
   End If
Loop

' Gewählte Tabelle
strField = astrFields(lngRow)

strSearch = InputBox("Geben Sie einen Suchbegriff ein" _
      & vbCrLf & "Das Prozentzeichen % für alles.", _
      strField)
If strSearch = "" Then Exit Sub

' Wildcard für "Alles ersetzen" ist das Prozentzeichen
adoRecordset.Open _
   "SELECT * FROM [" & strTable & "] WHERE [" & _
   strField & "] LIKE '%" & strSearch & "%' ;", _
   adoConnection, _
   adOpenKeyset, _
   adLockOptimistic
```

Listing 4.5 (Forts.)
Beispiele\04_Datenbanken\04_01_ADO.xls\mdlMdb

```
If Not (adoRecordset.EOF) Then
   With Worksheets(strWorksheet)

      ' Zellinhalt löschen
      .Cells.Clear

      ' Zum ersten Datensatz
      adoRecordset.MoveFirst

      ' Feldnamen in Zeile 2 eintragen
      For lngColumn = 1 To UBound(astrFields)
         .Cells(2, lngColumn) = astrFields(lngColumn)
      Next

      lngRow = 2
      ' Alle Datensätze durchlaufen
      Do While Not (adoRecordset.EOF)

         lngRow = lngRow + 1
         lngColumn = 0

         ' Alle Felder des Datensatzes durchlaufen
         For Each objField In adoRecordset.Fields

            lngColumn = lngColumn + 1

            ' und ausgeben
            .Cells(lngRow, lngColumn) = objField.Value

         Next
         ' Nächster Datensatz
         adoRecordset.MoveNext
      Loop
   End With
End If

adoRecordset.Close
adoConnection.Close
End Sub
```

4.8 dBase

Diese Datenbankdateien sind anders zu handhaben als beispielsweise Access-Dateien. Die dBase-Datenbank ist für ADO das Verzeichnis, in dem sich die *.dbf*-Dateien befinden. Die Tabellen sind diese Dateien, der Tabellenname leitet sich aus dem Dateinamen ab.

Manchmal hat man Probleme mit den Umlauten, da manche Programme, die dBase-Dateien erzeugen, den ASCII-(OEM-)Zeichensatz benutzen. In diesem Buch finden Sie in Kapitel 15 API-Funktionen, die Strings in beide Richtungen konvertieren können.

Nachfolgend eine Möglichkeit, die Verbindung zu DBase herzustellen:

```
    ' Bei Microsoft ActiveX Data Objects 2.0 ist
    ' der Provider Microsoft.Jet.OLEDB.3.51
    .ConnectionString = "Provider=Microsoft.Jet.OLEDB.4.0;" & _
          "Data Source=" & strPath & ";" & _
          "Extended Properties=dBASE IV;"
```

Mit diesem Beispiel können Sie per ADO auf dBase-Daten zugreifen.

Listing 4.6
Beispiele\04_Datenbanken\04_01_ADO.xls\mdlDBF

```
Sub testDBF()
   Dim adoConnection    As New ADODB.Connection
   Dim adoRecordset     As New ADODB.Recordset

   Dim lngColumn        As Long
   Dim lngRow           As Long
   Dim objWS            As Worksheet
   Dim strConn          As String
   Dim strSQL           As String
   Dim varRecord        As Variant
   Dim strPath          As String
   Dim strFile          As String
   Dim i                As Long
   Dim k                As Long

   ' Extras/Verweis auf Microsoft
   ' ActiveX Data Objects 2.X Library

   On Error GoTo Fehlerbehandlung

   ' Der Pfad zu den .dbf - Dateien.
   ' Nur Verzeichnisse!
   strPath = "C:\"

   ' Zieltabelle
   Set objWS = Worksheets("Tabelle2")

   ' Dateiname der .dbf - Datei
   strFile = Dir$(strPath & "\Auftrag1.dbf")

   If strFile <> "" Then
     ' Dateiname ist auch gleichzeitig
     ' der Tabellenname in SQL-Abfrage
     strFile = Left$(strFile, InStr(1, LCase(strFile), ".dbf") - 1)
   Else
      MsgBox "Datei existiert nicht"
   End If

   With adoConnection

     ' Bei Microsoft ActiveX Data Objects 2.0 ist
     ' der Provider Microsoft.Jet.OLEDB.3.51
     .ConnectionString = "Provider=Microsoft.Jet.OLEDB.4.0;" & _
           "Data Source=" & strPath & ";" & _
           "Extended Properties=dBASE IV;"
     .Open

   End With
```

Listing 4.6 (Forts.)
Beispiele\04_Datenbanken\04_01_ADO.xls\mdlDBF

```
    ' SQL-String erzeugen
    strSQL = strSQL & "SELECT * FROM [" & strFile & "] "

    ' Tabelleninhalt löschen
    objWS.Cells.ClearContents

    ' Abfrage starten
    adoRecordset.Open strSQL, _
       adoConnection, _
       adOpenDynamic, _
       adLockOptimistic

    ' Variant Array mit den Daten erzeugen
    varRecord = adoRecordset.GetRows

    ' Anzahl der Felder
    lngColumn = adoRecordset.Fields.Count

    ' Anzahl der Datensätze
    lngRow = UBound(varRecord, 2) + 1

    For k = 0 To lngColumn - 1
       ' Feldnamen eintragen
       objWS.Cells(1, k + 1) = adoRecordset.Fields(k).Name
    Next

   ' Recordset schließen, wird nicht mehr gebraucht
   adoRecordset.Close

    For i = 1 To lngRow

       For k = 1 To lngColumn
          ' Daten eintragen
          objWS.Cells(i + 1, k) = varRecord(k - 1, i - 1)
       Next

    Next
Fehlerbehandlung:
End Sub
```

4.9 Oracle-Datenbanken

Oracle-Datenbanken sind etwas schwieriger zu handhaben. Sie besitzen keine Datenbankdatei, deren Pfad als Datenquelle angegeben werden könnte. Stattdessen lauscht an irgendeiner Stelle im Netzwerk ein Programm an einem vorher festgelegten Port.

Bei Oracle-Datenbanken ist der Datenquellenname also kein Netzwerkpfad, sondern ein symbolischer Name, der ein `Alias` der eigentlichen Adresse darstellt. Die Daten, wo dieser Server tatsächlich zu erreichen ist – also die Hostadresse und den lauschenden Port –, finden Sie unter dem Aliasnamen in der Datei `Tnsnames.ora`. Hier exemplarisch ein Eintrag in der Datei `Tnsnames.ora`, der Name der Datenbank ist dabei XYZ.

```
XYZ =
  (DESCRIPTION =
    (ADDRESS_LIST =
      (ADDRESS = (PROTOCOL = TCP)(Host = 10.22.1.2)(Port = 1526))
    )
    (CONNECT_DATA = (SID = xyz))
  )
```

Es müssen aber noch ein paar andere Voraussetzungen erfüllt sein, um auf die Daten zugreifen zu können. Sie brauchen vom Datenbankadministrator einen Benutzernamen und ein zugehöriges Kennwort. Selbstverständlich benötigen Sie auch zumindest Leserechte.

Ein zu der Datenbank passender Oracle-Client muss auf Ihrem Rechner installiert sein und in diesem Programmverzeichnis unter `network\ADMIN` muss sich die passende Datei `Tnsnames.ora` befinden.

Von Ihrem Rechner aus müssen Sie den in dieser Datei angegebenen Host erreichen können. Dazu können Sie das Befehlszeilentool `Ping` benutzen. An der Eingabeaufforderung wird dazu Folgendes eingegeben:

`ping 10.22.1.2` (IP oder der Hostname)

Die Antwort sollte positiv ausfallen, der Host muss also ähnlich wie dieser hier antworten:

```
Ping 10.22.1.2 mit 32 Bytes Daten:
Antwort von 10.22.1.2: Bytes=32 Zeit<1ms TTL=128
Antwort von 10.22.1.2: Bytes=32 Zeit<1ms TTL=128
Antwort von 10.22.1.2: Bytes=32 Zeit<1ms TTL=128
Antwort von 10.22.1.2: Bytes=32 Zeit<1ms TTL=128
Ping-Statistik für 10.22.1.2:
    Pakete: Gesendet = 4, Empfangen = 4, Verloren
Ca. Zeitangaben in Millisek.:
    Minimum = 0ms, Maximum = 0ms, Mittelwert = 0ms
```

Anschließend können Sie mit dem Tool `tnsping` von Oracle prüfen, ob der Datenbankserver überhaupt bereit ist und die Einträge in der `tnsnames.ora` stimmen.

`tnsping XYZ` (symbolischer Name)

```
TNS Ping Utility for 32-bit Windows: Versionsinfos

Attempting to contact <ADDRESS=<PRO-
TOKOL=TCP><HOST=10.22.1.2><PORT=1526>
OK <520 msec>
```

Nachfolgend ein Connectionstring für Oracle-Datenbanken:

```
"Driver = {Microsoft ODBC for Oracle}; Server = XYZ; UID = MeName;
Password = myPass"
```

Als CursorLocation beim Verbinden sollten Sie `adUseClient` verwenden.

```
myConnection.CursorLocation = adUseClient
```

Die folgenden Prozeduren setze ich seit Jahren ohne Probleme ein:

```
Private adoRecordset    As ADODB.Recordset
Private adoDatabase     As ADODB.Connection

Public Sub OracleConnect()
   Set adoDatabase = New ADODB.Connection
      With adoDatabase
           .CursorLocation = adUseClient
           .ConnectionString = _
            "Driver={Microsoft ODBC for Oracle};" & _
            "Server=XYZ;UID=MeName;Password=MyPass"
           .Open
      End With
   Exit Sub
End Sub

Public Sub RecordsetFill(strSQL As String)
   Set adoRecordset = New ADODB.Recordset
   With adoRecordset
      .LockType = adLockOptimistic
      .CursorType = adOpenDynamic
      .ActiveConnection = Datenbank
      .Source = SQL
      .Open
   End With
End Sub
```

Die SQL-Syntax ist auch etwas anders als bei Access-Datenbanken. Näheres darüber finden Sie aber im Internet.

Einen kleinen Tipp kann ich Ihnen aber an dieser Stelle geben. Wenn Sie in der SQL-Abfrage Zeiten oder ein Datum verwenden, hat es sich bewährt, eine kleine selbst geschriebene Funktion dafür zu verwenden, aus einem VBA-Datum ein Oracle-SQL-String zu machen. Damit mit den Datums- und Zeitformaten wirklich nichts schief gehen kann, wird im erzeugten String die Oracle-SQL-Anweisung `To_Date` verwendet.

```
Function ToDateORA(ByVal myDate As Date)
   ToDateORA = Format$(myDate, "YYYY-MM-DD hh:mm:ss")
   ToDateORA = "TO_DATE('" & ToDateORA & "', _
      'YYYY-MM-DD HH24:MI:SS')"
End Function
```

Das Ergebnis sollte dann in etwa so aussehen:

```
TO_DATE('2004-06-29 11:45:32', 'YYYY-MM-DD HH24:MI:SS')
```

5

API-Grundlagen

5.1 Was Sie in diesem Kapitel erwartet

In den folgenden Kapiteln werden einige API Funktionen eingesetzt, ganz einfach deshalb, weil sich manche Funktionalitäten ohne diese Funktionen nicht realisieren lassen. Und dazu ist es wichtig, etwas mehr als der gewöhnliche VBA Anwender zu wissen.

VBA ist eine sehr mächtige Sprache, sie verbirgt aber auch fast all das, was sich tatsächlich bei einer Programmausführung ereignet. Die Mechanismen auf Betriebssystemebene sind für den Anwender einfach nicht sichtbar. Das entlastet den Programmierer ganz enorm und ist neben der leicht zu erlernenden Syntax auch ein Grund, dass VB(A) zu einer der beliebtesten Programmiersprachen in der Windows-Welt geworden ist.

VBA Programmierer brauchen dabei noch nicht einmal die Grundkonzepte eines Computers verstehen, geschweige denn, sich in die Tiefen von Prozessen, Threads und Heaps auszukennen.

Aber genau darin liegen eben auch die Schwierigkeiten beim Umgang mit der Windows-API. Die API, als eine sehr große Menge von offen gelegten Funktionen und Prozeduren, ist nicht in VB geschrieben und ist auch eigentlich nicht dafür gedacht, von VB(A) benutzt zu werden.

Allein den Entwicklern von VB(A) ist es zu verdanken, dass trotzdem ein großer Teil davon benutzt werden kann. Aber es ist nun einmal so, dass einige Parameter oder ganze Funktionen für Funktionalitäten da sind, die sich unter VBA gar nicht oder nur mit sehr großem Aufwand einsetzen lassen.

In diesem Kapitel bekommen Sie also eine kleine Einführung in die Benutzung der API. Es werden die Fallstricke angesprochen, die auf Sie lauern und es wird beschrieben, wie Sie diese umgehen können.

5.2 Was ist überhaupt die API?

***API* ist die Abkürzung für *Application Programming Interface*, frei übersetzt ins Deutsche bedeutet das »Programmierschnittstelle für Anwendungen«. Dabei handelt es sich um offen gelegte Schnittstellen von Standard-Dlls, damit andere Anwendungen auf die Funktionen und Prozeduren darin zugreifen können.**

Selbst gestandene Excel-Anwender mit Erfahrung in der VBA-Programmierung werden ganz schnell still, wenn es um das Thema API-Funktionen geht und Newbies erzittern geradezu vor Ehrfurcht. Das Wort API hat anscheinend etwas Mystisches und Unnahbares an sich. Das ist schade, denn an der API gibt es eigentlich nichts Geheimnisvolles. Nahezu alle Anwendungen bedienen sich im Hintergrund dieser Funktionen.

Im weiteren Verlauf dieses Kapitels wird der Einfachheit halber von API-Funktionen gesprochen, obwohl es vereinzelt auch API-Prozeduren gibt. Diese sind aber dünn gesät und der Unterschied zwischen Prozeduren und Funktionen ist sowieso bis auf das Funktionsergebnis bedeutungslos.

5.3 Datenspeicher

Ihrer Anwendung stehen im Speicher exklusiv 4.294.967.296 Bytes zur Verfügung, das sind 2^{32} unterschiedliche Speicheradressen. Aber nicht nur Ihre eigene Anwendung kann über so viel Speicher verfügen, sondern jede einzelne, die gerade auf Ihrem Rechner läuft. Möglicherweise wundern Sie sich jetzt, dass Sie über so viel Speicher verfügen, Sie als Hardwarebastler wissen ja schließlich, dass in Ihrem Rechner nur 512 MB an Speicherriegeln verbaut sind.

Tatsächlich muss dieser Speicher hardwaremäßig gar nicht existieren. Genauer gesagt handelt es sich bei der Angabe von 4,2 GB um verfügbare Speicheradressen und nicht um physikalischen Speicher in Form von RAM oder Festplattenspeicher. Jeder gestarteten Anwendung steht ein eigener, ca. 4,2 GB großer Adressraum zur Verfügung, wobei nur die erste Hälfte tatsächlich zum Speichern von Anwendungsdaten benutzt werden kann. Der Speicherraum oberhalb von etwa 2 GB ist für gemeinsam benutzte Objekte, Bibliotheken, System-DLLs und Gerätetreiber reserviert. Ausgenommen von diesem frei verfügbaren Bereich sind zudem noch die ersten 4 MB, wobei das erste Megabyte davon für die virtuelle DOS-Maschine benutzt wird.

Die eigentliche Abbildung des virtuellen in den physikalischen Speicher und der Zugriff darauf wird vom Betriebssystem geregelt. Die gestartete Anwendung bekommt davon rein gar nichts mit, aus der Sicht der Anwendung steht ihr sogar ganz allein der gesamte Rechner mit all seinen Ressourcen zur Verfügung.

Daten, d.h. Inhalte von Variablen und Datenfeldern, werden im virtuellen Adressraum der Anwendung abgelegt. Eine Variable, der Sie Daten zugewiesen haben, ist im Prinzip ein Zeiger auf die Adresse im virtuellen Speicherraum, ab welcher die Daten zu finden sind. Der gesamte Adressraum ist 32 Bit breit, deshalb handelt es sich bei einem Zeiger um einen Longwert und da es keine negativen Adressen gibt (außer vielleicht im realen Leben), um den Datentyp `unsigned-Long` (nicht vorzeichenbehaftet).

Wird einer Funktion eine Variable als Referenz übergeben, so landet diese virtuelle Speicheradresse auf dem Stack oder Stapelspeicher (siehe Abschnitt 5.4). Die aufgerufene Prozedur oder Funktion holt sich von dort diese Information und hat nun die Möglichkeit, diesen Speicherbereich zu manipulieren, da Sie ja die Adresse und den zugehörigen Datentyp kennt. Eine Übergabe als Wert `ByVal` würde den Inhalt der Variablen auf den Stack legen, also eine Kopie des Wertes. Dies kann zu gefährlichen Fehlern führen, wenn eine Funktion einen Zeiger erwartet.

Nehmen wir an, eine API-Funktion will den Speicher bearbeiten, dessen Adresse sie bekommen hat. Sie übergeben aber keine Adresse, wenn Sie die Variable als Wert, also `ByVal`, übergeben. In diesem Fall legen Sie auf den Stack leichtsinnigerweise eine Kopie des Wertes der Variablen. Ist dieser gerade 10.000.000, wird die aufgerufene Funktion versuchen, den Speicher an Adresse 10.000.000 zu manipulieren. Entweder wird dort schon benutzter Speicher überschrieben oder aber dieser angesprochene Speicherbereich wird noch gar nicht in den physikalischen Speicher abgebildet. Das führt in beiden Fällen fast immer zu einem sofortigen Absturz der Anwendung. Und das ist gut so!

Was Sie sich einprägen sollten, ist die Tatsache, dass gleiche Speicheradressen auch von mehreren Anwendungen gleichzeitig benutzt werden können. D.h., die Speicherstelle 1.000 der einen Anwendung hat rein gar nichts mit der Speicherstelle 1.000 der anderen zu tun. Es ist auch absolut nicht vorhersehbar, an welcher Stelle im physikalischen Speicher die Daten tatsächlich abgelegt werden. Ja, es ist noch nicht einmal sichergestellt, dass diese an dem einmal zugewiesenen Platz auch bleiben. Je nach Speicherauslastung können die Daten sogar vorübergehend oder dauerhaft in die *Swapdatei* (Auslagerungsdatei) ausgelagert werden. Die Prozessarbeitsräume der einzelnen Anwendungen sind zumindest bei NT und deren Nachfolgern strikt voneinander getrennt, und zwar so gut, dass es gar nicht so einfach ist, Daten zwischen zwei Anwendungen auszutauschen.

5.4 Stack

Wenn aus VBA eine Funktion oder ein Unterprogramm aufgerufen wird, passiert dabei aus der Sicht eines VBA Programmierers nicht allzu viel. Man benutzt die Funktion, wie man es irgendwann einmal gelernt hat. Erst wird die Variable eingetippt, die das Funktionsergebnis aufnehmen soll, dann kommt ein Gleichheitszeichen und darauf folgt der Funktionsname. Parameter, die mit an die Funktion übergeben werden sollen, stehen in Klammern dahinter, wobei mehrere Parameter durch Kommas getrennt sind.

Aus der Perspektive des Systems sieht die ganze Sache aber schon etwas anders aus. Steht der Aufruf eines Unterprogramms an, muss erst einmal die Adresse des Hauptprogramms gespeichert werden, an welcher der Prozessor sich gerade befindet. Das ist wichtig, damit nach Beendigung des Unterprogramms der Programmablauf genau an dieser Stelle weitergehen kann. Weiterhin müssen verschiedene Register und der Akkumulatorinhalt des Prozessors gesichert werden. Es wird quasi ein Schnappschuss des aktuellen Zustands gemacht, um nach der Rückkehr den Zustand, der vor dem Aufruf geherrscht hat, wiederherstellen zu können.

Diese Daten müssen nun irgendwo im Prozessarbeitsraum der Anwendung zwischengespeichert werden und dafür reserviert sich das Anwendungsprogramm einen Speicherbereich, der sich *Stack* oder auf Deutsch *Stapel* nennt. Jede Anwendung benutzt solch einen Stapelspeicher, wobei sichergestellt sein muss, dass dieser Speicherbereich nicht vom Anwendungsprogramm selbst verwaltet wird, sondern ausschließlich von der CPU.

Der Zugriff auf diesen Bereich wird unter Zuhilfenahme eines Registers gesteuert, dass sich `Stack-Pointer` oder Stapelzeiger (SP) nennt. Dieses Prozessorregister enthält einen Zeiger auf den letzten Eintrag im Stack, wobei der `Stack`, wie der Name schon sagt, als Stapel ausgeführt ist.

Wie bei einem echten Stapel kann ohne Probleme nur etwas ganz oben auf den Stapel gelegt werden. Beim Entnehmen sollte tunlichst dort oben auch wieder begonnen werden. Auf den Speicher übertragen haben Sie es bei einem Stack mit einem `FiLo`-Speicher (`First in, Last out`) zu tun. Interessant ist in diesem Zusammenhang vielleicht noch, dass der Stapel paradoxerweise in Richtung kleinere Speicheradressen höher wird, aber das spielt in unseren weiteren Betrachtungen keine Rolle.

Wichtig zu wissen ist, dass der Stapel auch zur Parameterübergabe an Funktionen und Prozeduren benutzt wird. Sie können unter der maschinennahen Programmiersprache Assembler zwar auch verschiedene Register zur Übergabe benutzen, aber bei der Verwendung von VB(A) wird der Weg über den Stack gegangen. Die aufgerufene Prozedur weiß, wie groß ihre einzelnen Parameter sein müssen und an welcher Stelle der Parameterliste diese stehen müssen. Somit kann sie sich unter Zuhilfenahme des `Stack-Pointers` die Position im Stack ausrechnen, an der diese Parameter zu finden sind.

Von dort werden diese Parameter eingelesen, unabhängig davon ob dort etwas Sinnvolles drinsteht oder nicht. Unter der schützenden Hülle von VBA wird dafür gesorgt, dass an diesen Stellen nichts hineinkommt, was die Anwendung gefährden könnte. Sie verlassen aber die schützende Hand von VBA, sobald Sie mit API-Funktionen arbeiten.

VBA prüft zwar noch, ob gemäß der Deklarationsanweisung die richtigen Datentypen übergeben werden, kann aber nicht erkennen, ob die Deklaration überhaupt stimmt. VBA nimmt Sie beim Wort, wenn Sie bei der Deklaration angeben, an welcher Stelle auf dem Stack sich die API-Funktion die Parameter abholen soll und legt diese Daten dementsprechend auch dort ab.

Achtung

Sie haben beim Einsatz der API die freie Auswahl, wie Sie Ihre Anwendung abstürzen lassen wollen. Sie können falsch deklarieren und legen auf dem Stack den falschen Datentyp ab oder Sie ziehen es vor, richtig zu deklarieren und übergeben beispielsweise einen frei gewählten `Long`-Wert statt eines gültigen Zeigers. Nichts und niemand wird Sie daran hindern. Der aufgerufenen API-Funktion ist die von Ihnen geschriebene Deklarationsanweisung völlig gleichgültig.

5.5 Parameterübergabe

Immer ein Quell zur Freude ist die Parameterübergabe an API-Funktionen. Diese Funktionen sind eben in einer anderen Sprache als VB(A) geschrieben und auch nicht extra dafür gemacht, von VBA benutzt zu werden.

Es fängt damit an, dass viele Funktionen als Argument einen Zeiger erwarten. Ein Zeiger ist ein 32-Bit-Wert, der die Adresse eines Speicherblocks enthält. Bei einem API-Funktionsaufruf werden von VBA die Argumente auf den Stack gelegt, die aufgerufene Funktion kann diese dann dort lesen und/oder manipulieren.

Erwartet jetzt die aufgerufene Funktion einen Zeiger, holt sie sich diesen vom Stack. Ihr ist es aber ganz egal, ob dort tatsächlich ein Zeiger oder irgendein anderer Wert steht. Die vier Bytes an dieser Stelle sind aus Sicht der Funktion ein Zeiger. Wenn Sie einen Wert übergeben haben, wird dieser als Adresse interpretiert, weil er sich gerade dort auf dem Stack befindet.

Wenn Sie Pech haben, wird nur von dieser Speicherstelle gelesen und Sie bekommen von Ihrem Fehler nichts mit. Mit etwas Glück stürzt die Anwendung ohne Umschweife ab, weil Speicher dann ab dieser Adresse überschrieben wird. In diesem Fall merken Sie wenigstens sofort, dass Sie etwas falsch gemacht haben.

Umgekehrt ist es mindestens genauso schlimm. Die Funktion erwartet einen Wert, beispielsweise die Länge eines Puffers, der überschrieben werden darf, Sie übergeben aber die Variable mit der Länge als Referenz. In VBA ist diese Variable einfach eine Black Box, welche Daten aufnimmt, aber auf dem Stack offenbart sich das wahre Gesicht. Es ist nämlich ein Zeiger auf einen Speicherbereich.

Solch ein Zeiger kann ziemlich hohe Werte annehmen und die Funktion interpretiert diesen dann als Länge des Puffers. Sie können mir ruhig glauben, dass dieser Wert meist größer ist als die Länge des angelegten Puffers. Auch hier gilt: Wenn Sie Pech haben, wird nur der Puffer oder ein Teil davon überschrieben. Mit etwas Glück stürzt die Anwendung sofort ab, weil Speicher außerhalb des Puffers überschrieben wird. Der erste Fall ist schlechter, weil es sehr lange dauern kann, bis es so richtig schön knallt. In dieser Zeit haben Sie aber vielleicht schon sehr viele fehlerhafte Berechnungen durchgeführt und möglicherweise schon abgespeichert.

API-Funktionen liefern anstatt Strukturen (in VBA benutzerdefinierte Typen) oft nur einen Zeiger auf die Speicherstelle, ab der diese Struktur beginnt. Sie müssen sich dann erst eine Variable von solch einem Typ anlegen. In diese müssen Sie dann mittels `CopyMemory` den Speicherinhalt kopieren.

5.6 Die Declare-Anweisung

Eine große Menge von Funktionen, mit denen Sie sehr viele nützliche, aber zum Teil auch ungemein schädliche Sachen anstellen können, befinden sich leider in Standard-Dlls und nicht in Com- oder ActiveX-Komponenten. Deshalb können Sie diese Funktionen auch nicht über einen Verweis in der VBE verfügbar machen. Visual Basic und VBA bieten aber mithilfe der `Declare`-Anweisung die Möglichkeit, Funktionen einer Standard-Dll für die Benutzung durch VBA verfügbar zu machen.

Hier folgt die Syntax für die Deklarationsanweisung einer API-Funktion (optionale Parameter sind in eckige Klammern eingeschlossen, das Zeichen »|« steht für ein `ODER`):

Für Prozeduren:

```
[Public|Private] Declare Sub SubName Lib "Dll-Name" _
   [Alias "Aliasname"] ([Argumente])
```

Für Funktionen:

```
[Public|Private] Declare Function FuncName Lib " Dll-Name " _
[Alias "Aliasname"] ([Argumente]) [As Typ]
```

- `[Public | Private]`

  Die `Declare`-Anweisung beginnt mit dem optionalen Parameter der Gültigkeit. Er regelt den Gültigkeitsbereich der deklarierten Prozedur oder Funktion. Bei `Public` steht die Funktion allen Modulen zur Verfügung, bei `Private` ist sie nur in dem Modul verfügbar, in dem auch die Deklaration steht. In Klassenmodulen muss dieser Parameter auf `Private` gesetzt werden, ein `Public` oder gar kein Parameter ist an dieser Stelle nicht möglich. Auf Modulebene wird ohne Angabe der Gültigkeit automatisch `Public` als Standardwert benutzt.

- `Declare Sub|Function`

  Nach der Angabe der Gültigkeit muss das Wort `Declare` folgen und je nachdem, ob es sich um eine Prozedur oder eine Funktion handelt, eines der Worte `Sub` oder `Function`.

- `FuncName`

  An dieser Stelle wird der Name der Prozedur oder Funktion eingegeben, wie er später verwendet werden soll. Verwenden Sie keinen Alias, muss dieser Name mit dem tatsächlichen Funktionsnamen in der Bibliothek exakt übereinstimmen und es wird dabei grundsätzlich auf Groß- und Kleinschreibung geachtet. Dieser Name darf auch nicht in Anführungszeichen stehen.

- `Lib "Dll-Name"`

  Danach folgt der so genannte `Lib`-Abschnitt. Er beginnt mit der Zeichenfolge `Lib`, auf den der Name der Bibliothek folgt, in der diese Funktion vorhanden ist. Falls sich die Bibliothek nicht in einem Hauptsuchpfad befindet, müssen Sie auch den Pfad mit angeben.

- `[Alias "Aliasname"]`

  Wenn Sie als Namen einen anderen als den tatsächlichen Funktionsnamen in der DLL verwendet haben, ist der optionale `Alias` wichtig. Hinter das Schlüsselwort `Alias` gehört dann der eigentliche Funktionsname, diesmal aber in Anführungszeichen. Auch hier wird wie bei allen Funktionsnamen in DLLs auf Groß- und Kleinschreibung geachtet. In Verbindung mit dem `Alias` ist es möglich, gleichen Funktionen verschiedene Namen zu geben.

  Manchmal kollidiert nämlich ein Funktionsname in einer Bibliothek mit einem VBA-Schlüsselwort und es wird deshalb notwendig, einen anderen Namen zu benutzen. Viel häufiger aber wird ein `Alias` eingesetzt, um typensichere Deklarationen zu schreiben. Anstatt einen typenlosen Parameter `As Any` zu deklarieren, deklarieren Sie den Parameter mit dem gewünschten Typ und denken sich einen speziellen Namen für die Funktion aus. Am besten wird dem verwendeten Funktionsnamen der geänderte Parametertyp angehängt. Aus dem Namen XYZ wird dann beispielsweise der Funktionsname XYZLong.

  Ein `Alias` wird häufig bei Funktionen verwendet, die mit Strings arbeiten. In vielen Fällen gibt es in der API nämlich zwei Varianten der gleichen Funktion. Eine für `ANSI` (A), und eine für `Unicode` (W), jeweils mit einem anderen Buchstaben am Ende. Ohne den Buchstaben am Ende sieht der Code anscheinend schöner aus. Anders ist es meiner Ansicht nach nicht zu erklären, warum bei den meisten Deklarationen darauf verzichtet wird.

- Parameterliste

  Nach dem `Lib`-Abschnitt und dem optionalen `Alias` folgt die Parameterliste in Klammern. Es ist extrem wichtig, hier die korrekten Datentypen anzugeben, die übergeben werden sollen. Bei API-Funktionen sind das meistens `Long`-Datentypen, Strukturen oder nullterminierte (`LPSTR`) Strings. Die Datentypen `Double` oder `Single` habe ich persönlich noch nicht erlebt. `Currency` wird ab und zu missbraucht, da VB(A) keine 64-Bit-Ganzzahlen kennt.

  Mindestens genauso wichtig ist es, anzugeben, ob der Parameter als Wert, d.h. mit dem Wort `ByVal`, übergeben werden soll. Ohne Angabe oder mit dem Schlüsselwort `ByRef` wird der Parameter als Referenz übergeben. Hier ist die häufigste Fehlerquelle im Umgang mit API-Funktionen zu suchen. Wichtig ist es, zu erwähnen, dass Strings immer `ByVal` deklariert sein müssen. Verfallen Sie ja nicht auf den Gedanken, dies zu ändern.

- Rückgabetyp

  Wenn es sich um eine Funktion handelt, die Sie mit der `Declare`-Anweisung verfügbar machen, geben Sie unbedingt am Ende den Rückgabetyp an. Dieser ist in nahezu allen Fällen ein `Long`. Machen Sie das nicht, wird ein Variant angenommen, was sehr böse Folgen haben kann.

Nachfolgend eine Deklarationsanweisung, in der auch ein `Alias` benutzt wird:

```
Private Declare Function CharLowerBuff Lib "user32.dll"
   Alias "CharLowerBuffA" ( _
   ByVal lpsz As String, _
   ByVal cchLength As Long _
   ) As Long
```

Diese Deklaration bedeutet Folgendes:

1. Durch das Schlüsselwort `Private` ist diese Funktion nur in dem Modul gültig, in dem diese Anweisung steht.
2. Es handelt sich um eine Funktion, da das Wort `Function` benutzt wurde.
3. Diese Funktion ist in dem Modul unter dem Namen `CharLowerBuff` verfügbar.
4. Die Funktion befindet sich in der DynamicLinkLibrary `user32.dll`. Der Pfad wurde nicht mit angegeben, da die Datei in einem der Standardpfade gespeichert ist.
5. Der tatsächliche Funktionsname in der DynamicLinkLibrary ist `CharLowerBuffA`.
6. Der Parameter `lpsz` wird wie alle Strings als Wert (`ByVal`) übergeben.
7. Der Parameter `cchLength` ist ein Long und wird als Wert (`ByVal`) übergeben.
8. Das Funktionsergebnis ist ein Longwert.

5.7 Datentypen

Wenn Sie dieses Buch lesen, sollten Sie die unterschiedlichen Datentypen kennen, die VBA Ihnen bietet. Sie gehen täglich damit um, kennen die Wertebereiche, Grenzen und Genauigkeiten. Das reicht vollkommen, wenn Sie mit VBA arbeiten. Sobald Sie aber die schützende Hand von VBA verlassen und mit Funktionen arbeiten, die für und mit einer anderen Sprache geschrieben wurden, ist es wichtig, etwas mehr darüber zu wissen. Beginnen wir deshalb etwas früher.

Informationen werden bekanntlich als Einsen und Nullen im Speicher eines Rechner gespeichert. Genauer gesagt, wird mit unterschiedlichen Spannungspegeln gearbeitet. Eine einzelne Speicherzelle kann dabei eine Eins oder eine Null aufnehmen und repräsentiert ein *Bit*. Mit den zwei Zuständen eines Bits könnten Sie zwei unterschiedliche Buchstaben, die Zahlen von null bis eins oder auch einen Wahrheitswert speichern.

Das ist aber in den meisten Fällen etwas wenig. Wenn Sie Text speichern oder übertragen wollen, sind 128 Kombinationen, die auch genauso viel unterschiedliche Buchstaben und Zeichen aufnehmen können, sicher nicht zu viel. Das entspricht den Kombinationsmöglichkeiten der Zustände von 7 Bits (2^7=128). Und auf genau diese 7 Bits stützt sich heute noch das gesamte E-Mail-System. Um 8 Bit Binärdaten mit 256 Kombinationen via E-Mail zu übertragen, wird dort auch heute noch die `Base64`-Codierung eingesetzt, die diese 8 Bit in entsprechend mehr 7 Bit Zeichen übersetzt.

Diese 7 Bit mit den 128 Kombinationsmöglichkeiten reichen aber auch noch nicht sehr weit. Der gebräuchliche *ANSI*- oder *ASCII*-Zeichensatz benötigt schon mal 256 Kombinationsmöglichkeiten. Deshalb werden 8 Bits (2^8=256) zu einem *Byte* zusammengefasst.

Aber auch diese 256 Zeichen sind manchmal noch etwas wenig, denkt man zum Beispiel an asiatische Schriftzeichen. Es wird daher immer häufiger *Unicode* benutzt, der 2^{16} Kombinationen benötigt und somit zwei Bytes pro Zeichen lang ist.

In diesem Unicode-Zeichensatz sind alle Zeichen definiert, die zurzeit weltweit auf Rechnern verwendet werden, momentan sind das etwa 40.000 Stück. Darunter sind auch mathematische Formelzeichen, Steuerzeichen, ja sogar Einzelteile von Zeichen wie zum Beispiel die zwei Pünktchen über den Ü, die sich mit dem U zu einem Ü kombinieren lassen. Übrigens verwendet VBA intern generell Unicode. Unter NT und deren Nachfolgern wie 2000 und XP werden auch außerhalb von VBA alle Zeichen im Arbeitsspeicher als Unicode abgelegt.

Ein Byte ist 8 Bit lang. Das ist so festgelegt. Jedes dieser 8 Bits hat eine bestimmte Wertigkeit. Das niederwertigste Bit hat die Wertigkeit 2^0=1 und wird als Bit Nummer 0 bezeichnet, das höchstwertigste ist das Bit 7. Siehe dazu Tabelle 5.1:

Tabelle 5.1
Bitwertigkeit

Bitnummer	2^x	Hex	Bitfolge	Wertigkeit
0	2^0	01	00000001	1
1	2^1	02	00000010	2
2	2^2	04	00000100	4
3	2^3	08	00001000	8
4	2^4	10	00010000	16
5	2^5	20	00100000	32
6	2^6	40	01000000	64
7	2^7	80	10000000	128
	Summe	**FF**	**11111111**	**255**

Ein gesetztes Bit wird mit seiner Wertigkeit multipliziert. Die Summe der einzelnen Bits, die vorher mit der Wertigkeit multipliziert wurden, ergeben den Wert des Bytes. Wenn für ein gesetztes Bit eine Eins und für ein nicht gesetztes eine Null geschrieben wird, ergibt ein `Byte` eine Zeichenfolge aus acht Einsen oder Nullen. Dabei kommt Bit Nummer null ans Ende. Bei dem Wert 15 ergibt sich also die Zeichenfolge:

```
00001111
```

Rechnung:

```
(0 mal 27=0) + (0 mal 26=0) + (0 mal 25=0) + (0 mal 24=0) + (1 mal
23=8) + (1 mal 22=4) + (1 mal 21=2) + (1 mal 20=1)=15
```

Um das Arbeiten mit Bytes und Bits zu vereinfachen, werden jeweils 4 Bits zusammengefasst. Diese 4 Bits werden auch als ein *Nibbles* bezeichnet. Um besser damit zu arbeiten, wird jede Kombination durch ein Zeichen dargestellt. 4 Bits sind 16 Kombinationen, also werden 16 unterschiedliche Zeichen benötigt,

die deshalb auch als Hexziffer (Hexa=16) bezeichnet werden. Für die Werte 0 bis 9 werden die Zahlen benutzt, für den Wert 10 der Buchstabe A verwendet, für 11 den Buchstaben B, und so wird fortgefahren bis zum Zeichen F für den Wert 15. Der Buchstabe F steht also für die Bitfolge 1111. Nachfolgend eine Tabelle (Tabelle 5.2) mit den Hexziffern:

Tabelle 5.2 Hexziffern

Dezimalwert	Hex	Binär	Dezimalwert	Hex	Binär
0	0	0000	8	8	1000
1	1	0001	9	9	1001
2	2	0010	10	A	1010
3	3	0011	11	B	1011
4	4	0100	12	C	1100
5	5	0101	13	D	1101
6	6	0110	14	E	1110
7	7	0111	15	F	1111

Zwei dieser Hex-Zeichen ergeben ein Byte. Hat also das Byte den Wert 15, können Sie den Hex-Code 0F benutzen. Eine Umrechnung in die Binärdarstellung mit Nullen und Einsen geht dabei sehr flott, da man es immer nur mit 4 Bits und 16 Kombinationen zu tun hat. Ein Wort, (Word) besitzt 2 Bytes und ein Doppelwort (DWORD), Sie werden es kaum glauben, besitzt 4 Bytes.

Um ganze Zahlen zu speichern, werden die Kombinationsmöglichkeiten von gesetzten und nicht gesetzten Bits genutzt. Für Ganzzahlentypen wie Byte, Integer und Long brauchen Sie für jede Zahl eine eindeutige Kombination davon. Mit 8 Bits – oder einem Byte – können Sie 256 Zahlen darstellen, das entspricht dem Zahlentyp Byte. Der Wert wird nach dem zuvor vorgestellten Schema berechnet. D.h., die Summe der einzelnen Bits, die vorher mit der Wertigkeit multipliziert wurden, ergeben den Wert.

Legen Sie fest, dass auch negative Zahlen erlaubt sind, dann handelt es sich um eine signed (vorzeichenbehaftete) Ganzzahl. Wenn das Bit 7 nicht gesetzt ist, entsprechen die Bits 0 bis 6 den Werten 0 bis 127. Bit 7 stellt das Vorzeichenbit dar. Ist dieses Bit gesetzt, ist die Zahl negativ.

Den niedrigsten darstellbaren Wert bekommen Sie bei einem vorzeichenbehafteten Datentyp, wenn nur das höchstwertige Bit gesetzt ist. Beim Integer ist das dann –32768 und beim Long der Wert –2.147.483.648. Zum Umrechnen der negativen Zahlen können Sie so vorgehen, dass alle Bits *invertiert* werden und eins hinzugezählt wird. Das Ergebnis ist die negative Zahl. Einige Beispiele negativer Ganzzahlen (Tabelle 5.3) folgen.

Tabelle 5.3
Negative Ganzzahlen

Datentyp	Bitfolge	Hex	Invertiert	Invertiert+1	Wert
Signed-Byte	10001111	8F	01110000	01110001	–113
Signed-Integer	11111111 11111111	FFFF	00000000 00000000	00000000 00000001	–1
Signed-Integer	10000000 00000001	8001	01111111 11111110	10000000 00000001	–32767
Signed-Integer	10000000 00000000	8000	01111111 11111111	10000000 00000000	–32768
Signed-Long	11111111 11111111 11111111 11111111	FFFFFFFF	00000000 00000000 00000000 00000000	00000000 00000000 00000000 00000001	–1
Signed-Long	10000000 00000000 00000000 00000000	80000000	01111111 11111111 11111111 11111111	10000000 00000000 00000000 00000000	–2.147.483.648

Beim Datentyp `Integer` sind 16 Bits, also 2 Bytes oder 4 Nibbles, und beim Datentyp `Long` 32 Bits, also 4 Bytes oder 8 Nibbles, beteiligt.

Manche API-Funktionen verlangen vorzeichenlose Parameter, meistens `unsigned Long`. VB kennt aber diesen Datentyp nicht, ein `Long`-Wert ist dort immer vorzeichenbehaftet. Trotzdem sind beide Datentypen prinzipiell gleich, sie werden nur unterschiedlich interpretiert.

Jetzt taucht natürlich sofort die Frage auf, wie man einen hohen `unsigned` (nicht vorzeichenbehafteten) `Integer`- oder `Long`-Wert in solch eine Variable bekommt, ein einfaches Zuweisen ist wegen eines Überlaufs nicht möglich. Nun, es gibt verschiedene Wege, die ans Ziel führen.

`LSet` kopiert unter anderem eine Variable eines benutzerdefinierten Datentyps in eine Variable eines anderen benutzerdefinierten Datentyps. Damit ist es möglich, Daten aus einer Variable, die aus einem Datentyp mit einem Longelement besteht, in eine Variable zu kopieren, die aus einem Datentyp mit einem Integerelement besteht.

Mit `CopyMemory` können Sie das Gleiche ohne benutzerdefinierte Datentypen erledigen, indem Sie direkt den Speicher manipulieren. Sie kopieren ab Speicherstelle `pScr` so viele Bytes, wie im Parameter `ByteLen` angegeben sind, an die Speicherstelle `pDest`. Die Übergabe einer Variablen ist, wenn sie als Referenz erfolgt, die Übergabe einer Speicheradresse. Nachfolgend (Listing 5.1) eine kleine Demonstration, wie Sie so etwas erledigen können.

Listing 5.1
Beispiele\02_Grundlagen_API\
02_01_ToSigned.xls\mdlToSigned

```
Private Declare Sub CopyMemory Lib "kernel32" Alias _
    "RtlMoveMemory" ( _
    pDst As Any, _
    pSrc As Any, _
    ByVal ByteLen As Long)

Private Type MyTCur
    C As Currency
End Type

Private Type MyTLong
    L As Long
End Type

Private Type MyTInt
    i As Integer
End Type

Sub ToSigned()
   Dim curMyCurrency As Currency
   Dim lngLong       As Long
   Dim udtCur        As MyTCur
   Dim udtLong       As MyTLong
   Dim udtInt        As MyTInt

   ' Zu groß für signed Long
   curMyCurrency = 2147483648@

   ' Zu groß für signed Integer
   lngLong = 65535

   ' Currency ist eine skalierte Ganzzahl mit
   ' 4 Nachkommastellen. Deshalb /10000
   udtCur.C = curMyCurrency / 10000

   ' LSet kopiert die rechte Struktur in die Linke
   LSet udtLong = udtCur
    'Ausgabe:
   MsgBox "Currency = " & curMyCurrency & vbCrLf & _
       "Long = " & udtLong.L & vbCrLf & _
       "Hex = &H" & Hex(udtLong.L) & vbCrLf & _
       "Bin = " & HexInBinStr(Hex(udtLong.L)), , _
       "Currency to signed Long with LSet"
   ' Currency = 2147483648
   ' Long = -2147483648
   ' Hex = &H80000000
   ' Bin = 10000000000000000000000000000000

   ' Ein Array mit zwei Long Elementen anlegen
   Dim ArrLong(1 To 2) As Long
   ' Mit CopyMemory Speicher an andere Stelle kopieren
   CopyMemory ArrLong(1), curMyCurrency, 8
   ' Ausgabe:
   MsgBox "Currency = " & curMyCurrency * 10000 & vbCrLf & _
       "Long = " & ArrLong(1) & vbCrLf & _
       "Hex = &H" & Hex(ArrLong(1)) & vbCrLf & _
       "Bin = " & HexInBinStr(Hex(ArrLong(1))), , _
       "Currency to signed Long with CopyMemory"
```

Listing 5.1 (Forts.)
Beispiele\02_Grundlagen_API\
02_01_ToSigned.xls\mdlToSigned

```
    ' Currency = 2147483648
    ' Long = -2147483648
    ' Hex = &H80000000
    ' Bin = 10000000000000000000000000000000

    ' Dem Element L des benutzerdefinierten Typs
    ' einen Longwert zuweisen
    udtLong.L = lngLong

    ' LSet kopiert die rechte Struktur in die Linke
    LSet udtInt = udtLong
    MsgBox "Long = " & lngLong & vbCrLf & _
        "Integer = " & udtInt.i & vbCrLf & _
        "Hex = &H" & Hex(udtInt.i) & vbCrLf & _
        "Bin = " & HexInBinStr(Hex(udtInt.i)), , _
        "Long to signed Integer with LSet"
    ' Ausgabe:
    ' Long = 65535
    ' Integer = -1
    ' Hex = &HFFFF
    ' Bin = 1111111111111111

    ' Ein Array mit zwei Integer Elementen anlegen
    Dim ArrInt(1 To 2) As Integer
    ' Mit CopyMemory Speicher an andere Stelle kopieren
    CopyMemory ArrInt(1), lngLong, 4
    ' Ausgabe:
    MsgBox "Long = " & lngLong & vbCrLf & _
        "Integer = " & ArrInt(1) & vbCrLf & _
        "Hex = &H" & Hex(ArrInt(1)) & vbCrLf & _
        "Bin = " & HexInBinStr(Hex(ArrInt(1))), , _
        "Long to signed Integer with CopyMemory"
    ' Long = 65535
    ' Integer = -1
    ' Hex = &HFFFF
    ' Bin = 1111111111111111

End Sub

Public Function HexInBinStr(ByVal HexZahl As String) As String
   Dim i       As Long
   Dim bytBin  As Byte

   On Error GoTo Fehler

   ' Hexstring in Großbuchstaben umwandeln
   HexZahl = UCase(HexZahl)

   Do While Len(HexZahl) > 0 ' Verlassen, wenn String leer

      ' Jedes Nibble (4 Bits) in ein Byte umwandeln
      bytBin = CByte("&H" & Right$(HexZahl, 1))
```

Listing 5.1 (Forts.)
Beispiele\02_Grundlagen_API\02_01_ToSigned.xls\mdlToSigned

```
        For i = 0 To 3
            ' 0, wenn Bit nicht gesetzt, 1, wenn Bit gesetzt ist
            ' Ergebnis vor den Hexstring setzen
            HexInBinStr = ((bytBin And 2 ^ i) > 0) * -1 & HexInBinStr
        Next

        ' Den Hexstring um das gerade bearbeitete Nibble kürzen
        HexZahl = Left$(HexZahl, Len(HexZahl) - 1)

    Loop ' Nächstes Nibble

Fehler:
End Function
```

Wenn Sie sich Deklarationsanweisungen anschauen, werden Sie feststellen, dass die einzelnen Parameter meistens ein bis drei Zeichen lange *Präfixe* haben. Das sind die kleingeschriebenen Buchstaben am Anfang eines Parameternamens und damit werden die Datentypen gekennzeichnet.

Hier eine kleine Liste der gebräuchlichsten API-Präfixe (Tabelle 5.4):

Tabelle 5.4
Präfixe API-Parameter

Präfix	Typ	Beschreibung
b	Boolean	32-Bit-Wahrheitswert. Nur der Wert null entspricht dem Wahrheitswert Falsch. Alles andere bedeutet Wahr.
ch	Char, Byte, Tchar	8-Bit-Wert. Bei Tchar kann es, wenn es sich um Unicode handelt, auch ein 16-Bit-Wert (nicht vorzeichenbehaftet) sein.
lpfn	FARPROC	32-Bit-Funktionszeiger. Damit werden Sie in Excel wahrscheinlich nie konfrontiert werden. Ein paar schöne Sachen lassen sich aber nur damit realisieren. Versionen > Excel 97 kennen AddressOf, bei Excel 97 kann die Funktionalität nachgebaut werden.
h	Handle	32-Bit-Wert. Nicht vorzeichenbehaftet. Ermöglicht in Verbindung mit API-Funktionen den Zugriff auf Windows-Objekte.
n	Integer	32-Bit-Wert. Vorzeichenbehaftet.
l	Long	32-Bit-Wert. Vorzeichenbehaftet.
lp	Long Pointer	32-Bit-Zeiger auf einen Speicherbereich.
lpi	Long Pointer	32-Bit-Zeiger auf einen 32 Bit vorzeichenbehafteten Wert.
w	Word	16-Bit-Wert. Nicht vorzeichenbehaftet.
dw	DoubleWord	32-Bit-Wert. Nicht vorzeichenbehaftet.
f	Flag	Ein Bit in einem 32-Bit-Wert, das eine besondere Bedeutung hat.

5.8 Strings

Der String stellt bei der Übergabe an API-Funktionen eine Besonderheit dar. In VB(A) handelt es sich bei einem String um einen BSTR-Unicodestring (Tabelle 5.5).

2	0	0	0	65	0	66	0	0	0
Präfix Stringlänge (hier 2 Zeichen)				Zeichen »A«		Zeichen »B«		Stringende \0	

Tabelle 5.5
BSTR-Unicodestring im Speicher

Ein BSTR-String ist ein (Long-)Zeiger auf den Beginn des ersten Zeichens im Speicher. Vor diesem ersten Zeichen befindet sich noch ein Longwert, der die Anzahl der Zeichen im Speicher angibt. Hinter dem letzten Zeichen finden Sie ein Nullbyte, welches das Ende des Strings markiert.

***ASCII* ist eigentlich ein 7-Bit-Zeichensatz, der irgendwann auf 256 Zeichen erweitert wurde. Manche DOS-Programme wie zum Beispiel dBase verwenden ASCII. Windows verwendet den vom American National Standards Institute genormten *ANSI*-Zeichensatz. Die zwei Zeichensätze stimmen nur von Position 32 bis 127 überein.**

Handelt es sich dabei um einen Unicodestring, belegt jedes Zeichen zwei Bytes und auch die Kennung für das Stringende wird durch zwei Nullbytes markiert. VBA beachtet die Kennung für das Stringende aber nicht, sondern richtet sich ausschließlich nach der Längenangabe vor dem eigentlichen String. Nur deshalb sind bei einem BSTR auch Zeichen mit dem ANSI-Code null zulässig.

Die meisten API-Funktionen wollen keinen BSTR-, sondern einen LPSTR-String (Tabelle 5.6).

65	0	66	0	0	0
Zeichen »A«		Zeichen »B«		Stringende \0	

Tabelle 5.6
LPSTR-Unicodestring im Speicher

Ein LPSTR ist genauso wie BSTR ein (Long-)Zeiger auf das erste Zeichen eines Strings, der mit einem Nullbyte abgeschlossen wird. Bei Unicode ist das Ende des Strings an zwei Nullbytes hintereinander im Speicher zu erkennen. Da bei LPSTR das oder die Nullbytes das Ende des Strings markiert, dürfen in solch einer Zeichenkette auch keine Zeichen mit dem Wert null vorkommen. Wenn Sie BSTR mit LPSTR vergleichen, können Sie feststellen, dass beide bis auf das Präfix für die Längenangabe (bei BSTR) absolut gleich sind.

Strings an API-Funktionen zu übergeben ist eine Sache für sich. Wie beschrieben, sind Zeichenketten in VBA vom Typ BSTR, die API möchte aber LPSTR. Das ist erst einmal nicht weiter schlimm, denn die Zeiger der Stringvariablen zeigen bei BSTR sowie bei LPSTR auf das erste Zeichen im Speicher und beide sind sogar nullterminiert.

Der Knackpunkt ist, dass bei VBA alle Strings im Unicodeformat vorliegen, die meisten API-Funktionen aber keine Unicodestrings mögen. Die Entwickler von VB(A) haben sich deshalb nun etwas ganz Besonderes einfallen lassen. Wenn Sie einen String an eine API-Funktion übergeben, wird vorher eine ANSI-Kopie im Speicher erstellt. An die API-Funktion wird also kein Zeiger auf den Originalstring, sondern ein Zeiger auf diese Kopie übergeben.

Sie werden bald feststellen, dass viele API-Funktionen ein A am Ende des Funktionsnamens haben und nur einige wenige ein W. Das A am Ende bedeutet ANSI, das W (*Wide*) steht für Unicode. Wollen Sie eine Unicodefunktion der API benutzen, müssen Sie mit anderen Typen – wie Bytearrays – arbeiten oder den String mit StrConv so umwandeln, dass er beide Bytes des Unicodezeichens als gesonderte Zeichen enthält. Im Direktfenster der Entwicklungsumgebung von Excel (VBE) können Sie sich das Ergebnis anschauen:

```
Print StrConv("asd",vbUnicode)
```

API-Funktionen, die als String übergebene Parameter manipulieren, manipulieren – wie vorher angesprochen – eine Kopie des VBA-Strings. Deshalb wandelt VBA den String nach Beendigung der API-Funktion wieder in einen Unicodestring um und kopiert ihn anschließend an die Originaladresse im Speicher zurück. Das Zurückkopieren passiert auch, wenn Sie diese Stringvariable explizit mit dem Wort ByVal als Wert übergeben.

In Basic kann, wie schon zuvor erwähnt, ein String auch Zeichen mit dem Wert null enthalten, wenn Sie beispielsweise einen Stringpuffer anlegen und diesen mit dem Inhalt einer Binärdatei füttern. Wenn Sie diesen String an eine API-Funktion übergeben, ist für diese Funktion der String beim ersten Auftreten einer Null beendet. Ich habe einmal Stunden damit verbracht, herauszufinden, warum die Ausgabe eines zusammengesetzten Strings in einer Messagebox nicht so richtig funktioniert. Ich hatte ganz einfach vergessen, ein CHR(0) am Ende des ersten Strings zu löschen. Für die Messagebox war der Text an dieser Stelle zu Ende. Im Direktfenster der VBE können Sie das jederzeit nachvollziehen.

```
MsgBox "1234" & Chr(0) & "5678"
```

5.8.1 Kommandozeile auslesen

Manchmal liefert eine API-Funktion auch nur einen Pointer auf einen String. Als Beispiel ist hier die Funktion GetCommandLineA anzuführen (Listing 5.2). Damit ist es möglich, die Kommandozeile auszulesen, mit der Excel gestartet wurde.

Listing 5.2
Beispiele\02_Grundlagen_API\02_02_Commandline.xls\mdlCommandline

```
Private Declare Function GetCommandLine Lib "kernel32" _
   Alias "GetCommandLineA" () As Long

Private Declare Function lstrlen Lib "kernel32" ( _
   ByVal str As Long _
   ) As Long

Private Declare Function lstrcpy Lib "kernel32" ( _
   ByVal dest As String, _
   ByVal src As Long _
   ) As Long

Private Function StringVonPointer(plngAnsi As Long)
   Dim lngAnzahl  As Long
   Dim strName    As String
```

Listing 5.2 (Forts.)
Beispiele\02_Grundlagen_API\
02_02_Commandline.xls\
mdlCommandline

```
    ' Länge des Strings ermitteln
    lngAnzahl = lstrlen(plngAnsi)

    ' Puffer anlegen
    strName = String(lngAnzahl, 0)

    ' String ab dem Pointer in den Puffer kopieren
    lstrcpy strName, plngAnsi

    If InStr(1, strName, Chr(0)) <> 0 Then
        ' Beim ersten Auftreten von chr(0) kürzen
       strName = Left$(strName, InStr(1, strName, Chr(0)) - 1)
    End If

    ' String als Funktionsergebnis zurückgeben
    StringVonPointer = strName
End Function

Public Function ExtractCommandline() As String
    ' Kommandozeile auslesen
    ExtractCommandline = StringVonPointer(GetCommandLine())
End Function
```

Wichtig ist bei diesem Listing die Funktion `StringVonPointer`, die aus dem als Funktionsergebnis von `GetCommandLine` gelieferten Longwert einen Basicstring macht. Die API-Funktion `lstrlen` ermittelt dabei die Länge des Strings im Speicher ab dem Pointer `plngAnsi`. Ist die Länge des Strings bekannt, können Sie anschließend einen Stringpuffer in der entsprechenden Größe erzeugen, der zum Schluss den String aufnimmt. Dazu wird mit `lstrcpy` der String ab dem Pointer (Zeiger) `plngAnsi` in den Speicherbereich kopiert, an dem VBA eine ANSI-Kopie des Puffers abgelegt hat. Nach Beendigung von `lstrcpy` kopiert VBA den String aus diesem Bereich wieder als Unicode in den ursprünglich angelegten Puffer zurück.

Wenn Sie an Excel einen eigenen Parameter übergeben und auswerten wollen, können Sie folgendermaßen vorgehen: Unter START | AUSFÜHREN geben Sie die Befehlszeile nach folgenden Schema ein (funktioniert auch aus einer Batchdatei):

```
»Pfad zu Excel.exe« /e/EigenerParameter »Pfad zur Arbeitsmappe«
```

- `»Pfad zu Excel.exe«`

  Wenn nur eine Version von Excel auf Ihrem Rechner existiert, kann der komplette Pfad weggelassen werden, in diesem Fall reicht das Wort EXCEL. Der Pfad muss in Anführungszeichen eingeschlossen sein.

- `/e/EigenerParameter`

  Hinter `/e/` können Sie einen eigenen Startparameter übergeben. Leerzeichen sind darin nicht erlaubt.

- `»Pfad zur Arbeitsmappe«`

  An dieser Stelle wird der Pfad zur Arbeitsmappe angegeben. Dieser muss in Anführungszeichen eingeschlossen sein.

Zum Extrahieren des übergebenen Parameters aus der Kommandozeile können Sie dann folgenden Code (Listing 5.3) benutzen:

Listing 5.3
Beispiele\02_Grundlagen_API\02_02_Commandline.xls\mdlCommandline

```
Private Sub cmdCommandline_Click()
   Dim strStartparam As String
   Dim strCommand    As String
   Dim lngPos        As Long
   Dim lngLen        As Long
   ' Aufrufen mit
   ' "Pfad zu Excel.exe" /e/EigenerParameter "Pfad zur Mappe"

   ' Kommandozeile holen
   strCommand = ExtractCommandline()

   ' Hinter den Ziffern /e/ steht der eigene Startparameter.
   ' Die Position der Kennung /e/ ermitteln
   lngPos = InStr(1, strCommand, "/e/")

   If lngPos > 0 Then
   ' Die Kennung /e/ für den eigenen Startparameter ist vorhanden

      ' Die Länge des Parameters bis zum Auftreten
      ' eines Leerzeichens oder bis zum Ende ermitteln
      lngLen = InStr(lngPos, strCommand, " ") - 3 - lngPos
      If lngLen < 0 Then lngLen = Len(strCommand)

      ' Den eigenen Übergabeparameter aus der Kommandozeile holen
      strStartparam = Mid(strCommand, lngPos + 3, lngLen)

   End If

   ' Ausgabe
   MsgBox "Kommandozeile : " & strCommand & vbCrLf & vbCrLf & _
      "Eigener Startparameter : " & strStartparam, , _
      "Commandline"
End Sub
```

Den eigenen Startparameter können Sie beispielsweise in der `Workbook_Open`-Prozedur im Klassenmodul `DieseArbeitsmappe` auswerten. Diese Ereignisprozedur wird beim Öffnen der Arbeitsmappe abgearbeitet und Sie haben dort die Möglichkeit, je nach übergebenen Parameter beliebigen Code auszuführen.

5.9 CopyMemory

Die wohl wichtigste Prozedur zur Speichermanipulation ist `CopyMemory`, deshalb habe ich dieser einen eigenen Abschnitt gewidmet. Diese Prozedur kopiert Daten von einem Datenblock im Speicher an eine andere Stelle im Speicher. Das gilt ausschließlich für Daten im gleichen Prozessarbeitsraum. Über Prozessgrenzen hinweg können damit aber keine Daten ausgetauscht werden.

Nachfolgend die Deklarationsanweisung für diese Prozedur:

```
Private Declare Sub CopyMemory Lib "kernel32"
   Alias " RtlMoveMemory" ( _
   ByVal pDst As Any, _
   ByVal pScr As Any, _
   ByVal ByteLen As)
```

Diese Deklaration bedeutet Folgendes:

1. Durch das Schlüsselwort `Private` ist diese Funktion nur in dem Modul gültig, in dem diese Anweisung steht.
2. Es handelt sich um eine Prozedur, da das Wort `Sub` benutzt wurde.
3. Diese Funktion ist in dem Modul unter dem Namen `CopyMemory` verfügbar.
4. Die Funktion befindet sich in der DynamicLinkLibrary `kernel32`. Der Pfad wurde nicht mit angegeben, da die Datei in einem der Standardpfade gespeichert ist.
5. Der tatsächliche Funktionsname in der DynamicLinkLibrary ist `RtlMoveMemory`.
6. Der Parameter `pDest` ist ein Zeiger auf den Zieldatenblock, der mit den Daten ab dem Zeiger `pScr` überschrieben werden soll. Stellen Sie sicher, dass der Zieldatenblock größer oder gleich der Länge ist, die in `ByteLen` angegeben wurde.
7. Der Parameter `pScr` ist Zeiger auf den Quelldatenblock, von dem Daten in den Zieldatenblock ab dem Zeiger `pDest` geschrieben werden sollen. Stellen Sie sicher, dass der Quelldatenblock größer oder gleich der Länge ist, die in `ByteLen` angegeben wurde.
8. Der Parameter `ByteLen` gibt die Anzahl der Bytes an, die kopiert werden sollen.

`CopyMemory` ist zwar einer der wichtigsten, aber andererseits auch eine der gefährlichsten API-Funktionen überhaupt, jedenfalls was die Ursache für Programmabstürze betrifft. Es ist damit relativ leicht, Speicherbereiche zu überschreiben, die eigentlich Tabu sein sollten. Die Verwechslung der Parameterübergabe `ByVal` mit `ByRef` ist dabei die häufigste Ursache für einen Crash.

Der Grund dafür ist, dass diese Prozedur für die ersten beiden Parameter einen Longwert auf dem Stack erwartet, der die Ziel- bzw. die Quelladresse angibt. Übergeben Sie das Element eines Arrays als Referenz, wird genauso wie gewollt, ein 4 Byte breiter `Long`-Wert mit der Adresse auf den Stack gelegt, selbst wenn es sich bei den Daten um ein Bytearray handelt.

Die Übergabe `ByVal` führt dagegen in fast allen Fällen zum sofortigen Absturz der Anwendung, da der Wert der Variablen auf den Stack gelegt wird. Und dieser Wert wird dann von der aufgerufenen API-Funktion/Prozedur, also in diesem Fall `CopyMemory`, vom Stack geholt und als Speicheradresse interpretiert.

Die gleiche Problematik gilt auch für alle anderen Variablen, deren Adresse übergeben werden soll. Nur die Übergabe als Referenz legt die Adresse zu dem Wert dieser Variablen auf den Stack.

Etwas anders ist der Fall, wenn es sich um eine Longvariablen handelt, die einen Zeiger als Wert enthält, dann muss die Variable `ByVal` übergeben werden, ansonsten gibt es einen Absturz. Wird in solch einem Fall die Variable als Referenz übergeben, wird die Speicheradresse des eigentlichen Wertes auf den Stack gelegt und nicht der Wert, der auf die eigentliche Adresse zeigt.

Solch einen Wert bekommen Sie beispielsweise mit den undokumentierten Funktionen `VarPtr`, `ObjPtr` und `StrPtr`. Übergeben Sie eine Variable mit solch einem Wert als Referenz, wird auf den Stack zwar ein Longwert abgelegt, dieser verweist aber auf die Speicherstelle, an welcher der eigentliche Zeiger steht. Um den eigentlichen Zeiger auf den Stack zu bekommen, müssen Sie dann die Variable `ByVal` übergeben.

Achtung

Die häufigste Ursache für Programmabstürze beim Benutzen der API-Prozedur `CopyMemory` ist der falsche Einsatz von `ByVal` und `ByRef`. Sichern Sie vor dem Testen auf jeden Fall Ihre Anwendung.

Im nachfolgenden Abschnitt über Little- und Big Endian können Sie `CopyMemory` mit ein paar erläuternden Worten im Einsatz erleben.

5.10 Little Endian, Big Endian

Die Darstellung der Bits auf dem Papier entspricht nicht der tatsächlichen Lage im physikalischen Speicher. Schaut man sich eine Bitfolge auf dem Papier an, so ist das niederwertigste Bit immer rechts außen dargestellt. Nach links werden die Ordnungszahlen der Bits immer größer (Tabelle 5.1).

Nehmen wir als Beispiel einmal die Integervariable mit dem Wert 6:

Die Bitfolge dieser Zahl ist:

```
0000000000000110
```

Das entspricht hexadezimal der Zeichenfolge `&H0006`. Das Präfix als Erkennungszeichen für eine hexadezimale Zeichenfolge ist in VBA `&H`. Könnte man sich jetzt den Speicherbereich ansehen, an dem der Wert abgelegt ist, zum Beispiel mit einem Hex-Editor, würde man stutzen. Die zwei Bytes sind augenscheinlich vertauscht. An der kleineren Speicheradresse steht das niederwertige Byte, also sieht es im Speicher in Wirklichkeit so aus:

```
&H0600
```

Wenn Sie einen Long mit dem Wert 6 nehmen, der hexadezimal folgendermaßen dargestellt wird

```
&H00000006,
```

sieht der Speicherauszug so aus:

```
&H06000000
```

Dort sind also nicht nur die Bytes getauscht, sondern auch die WORDS untereinander. Dies wird dann Little Endian genannt. Intel benutzt dieses System, aber auch die entgegengesetzte Variante, das Big Endian, gibt es im realen Leben. Das Internet überträgt sogar in diesem Format.

In diesem Beispiel (Listing 5.4) zum Sichtbarmachen des Speicherinhaltes kommt die API-Funktion CopyMemory zum Einsatz:

Listing 5.4
Beispiele\02_Grundlagen_API\02_03_Endian.xls\mdlShowMemory

```
Private Declare Sub CopyMemory Lib "kernel32" _
   Alias "RtlMoveMemory" ( _
   pDst As Any, _
   pSrc As Any, _
   ByVal ByteLen As Long)

Public Sub ShowMemory()
   Dim lngLong             As Long
   Dim abytBuffer(1 To 4)  As Byte
   Dim astrByte(1 To 4)    As String
   Dim strHex              As String
   Dim strOriginal         As String

   ' Longvariable mit Wert &HOF=15 füllen
   lngLong = 15

   ' Jetzt wird die Longvariable in den Puffer kopiert
   CopyMemory abytBuffer(1), lngLong, 4

   ' Alle 4 Bytes in Hexzeichen mit führenden Nullen
   ' (String(2 - Len(Hex(bytBuffer(1))), "0")) umwandeln
   astrByte(1) = String(2 - Len(Hex(abytBuffer(1))), "0") & _
      Hex(abytBuffer(1))
   astrByte(2) = String(2 - Len(Hex(abytBuffer(2))), "0") & _
      Hex(abytBuffer(2))
   astrByte(3) = String(2 - Len(Hex(abytBuffer(3))), "0") & _
      Hex(abytBuffer(3))
   astrByte(4) = String(2 - Len(Hex(abytBuffer(4))), "0") & _
      Hex(abytBuffer(4))

   ' Originalstring in das Hexformat mit führenden Nullen
   ' umwandeln
   strOriginal = String(8 - Len(Hex(lngLong)), "0") & _
      Hex(lngLong)

   ' Ausgabestring zusammensetzen
   strHex = astrByte(1) & astrByte(2) & astrByte(3) & astrByte(4)

   ' Ergebnis ausgeben
   MsgBox strOriginal & " = Original" & vbCrLf & _
          strHex & " = Im Speicher", , "Bytefolge im Speicher"
End Sub
```

Die Prozedur CopyMemory kopiert den Speicherinhalt ab Adresse pSrc an die Adresse pDst, und zwar so viele Bytes, wie Sie im Parameter ByteLen angegeben haben. Ein Long ist 4 Bytes lang, also erzeugen wir einen Puffer, der diese 4 Bytes auch aufnehmen kann. Wenn ein zu kleinen Puffer erzeugt wird oder der Parameter ByteLen zu groß gewählt wird, wird ohne Rückfrage Speicher überschrie-

ben. Dort kann alles Mögliche gespeichert sein. Hat man Glück, stürzt die Anwendung sofort ab, hat man Pech, so treten irgendwann die seltsamsten Fehler auf. Diese Fehler dann auf einen überschriebenen Speicherbereich zurückzuführen, ist gar nicht so einfach, besonders wenn die Anwendung schon einige Zeit im Einsatz ist.

Sehr schön kann man erkennen, wie hier Adressen der Speicherbereiche über die Parameter `pSrc` und `pDst` an die Prozedur übergeben werden. Eine Variable ist ja nichts anderes als ein Zeiger dorthin, deshalb auch die Übergabe der Variablen als Referenz. Ausgenommen ist der Parameter Länge, denn da will die Funktion ja einen Wert auf dem Stack haben.

Was macht das kleine Programm sonst noch? Die einzelnen Speicherstellen werden in der Reihenfolge, wie sie im Speicher stehen in einen `HexCode` umgewandelt. Die Konstruktion `String(2 - Len(Hex(Buffer(2))), "0")` dient lediglich dazu, den Hexcode mit führender Null darzustellen.

Sie sollten zum Verständnis einmal den Longwert der Variablen `lngLong` verändern und sich anschauen, was dann passiert.

5.11 Arrays, Puffer und Speicher

5.11.1 Arrays allgemein

Arrays sind Datenfelder, die eine Folge von gleichartigen Elementen im Speicher darstellen, wobei auch mehrdimensionale Arrays möglich sind. Laut Onlinehilfe liegt die Grenze bei 60 Dimensionen. Jedes Element besitzt dabei einen eindeutigen Index. Wenn die `Option Base`-Anweisung nichts anderes angibt, beginnt die Zählung beim Index null.

Wenn Sie bei der Deklaration die Klammer leer lassen, in der die Größe und die Dimensionen des Datenfelds angegeben werden, haben Sie ein dynamisches Datenfeld. Mithilfe der `ReDim`-Anweisung können Sie dann während der Laufzeit die Anzahl der Dimensionen, die Anzahl der Elemente und die oberen und unteren Grenzen jeder einzelnen Dimension anpassen. Mit `ReDim Preserve`, können Sie sogar die obere Grenze der letzten Dimension erweitern, ohne die vorhandenen Werte im Array zu löschen. Sie können auch die obere Grenze der letzten Dimension verkleinern, dabei fallen aber selbstverständlich die überschüssigen Daten weg, während die restlichen Daten erhalten bleiben. Bis hierher dürfte man auch alles in der Onlinehilfe nachschlagen können.

5.11.2 Arrays im Speicher

Interessanter wird die Sache, wenn man weiß, wie die Daten eines Arrays im Speicher abgelegt werden. Wird beispielsweise ein Datenfeld des Typs `Byte` mit hundert Elementen deklariert, wird dafür logischerweise Speicherplatz benötigt. Zum Ablegen der Daten werden in diesem Fall hundert Bytes reserviert, die alle

im Speicher *nebeneinander* stehen. Bei zweidimensionalen Datenfeldern stehen die Elemente der ersten Dimension dann jeweils nebeneinander.

D.h., erst stehen die Elemente des ersten Elements der zweiten Dimension nebeneinander, dann folgen die Elemente des zweiten Elements der zweiten Dimension, dann folgen die Elemente des dritten Elements der zweiten Dimension usw.

Das sieht dann so aus:

```
(1,1)(2,1)(3,1(1,2)(2,2)(3,2)(1,3)(2,3)(3,3)
```

Und etwas übersichtlicher dargestellt:

```
(1,1)(2,1)(3,1)
(1,2)(2,2)(3,2)
(1,3)(2,3)(3,3)
```

Jetzt wird auch klar, warum bei dynamischen Arrays unter Beibehaltung der vorherigen Daten nur die letzte Dimension geändert werden kann. Wollte man von der ersten Dimension die Anzahl der Elemente ändern, müsste man aus der Sicht der VB-Entwickler genauso oft Speicherinhalte verschieben, wie Elemente der zweiten Dimension vorhanden sind. Dagegen brauchen Sie bei der Änderung der zweiten Dimension nur Speicherplatz hinzuzufügen oder zu entfernen. Das Wissen um die Speicherbelegung bei Arrays können Sie sich zunutze machen, um zwei Arrays ohne eine Schleife zusammenzufügen.

Im nachfolgenden Listing (Listing 5.5) zur Manipulation von Datenfeldern mithilfe der API-Funktion `CopyMemory` haben Sie als Ausgangslage zwei verschiedene Integer-Datenfelder, die als Quelle dienen sollen. Das Datenfeld `intField1` enthält sechs Elemente und genauso viel Elemente hat auch das Datenfeld `intField2`.

Um diese beiden Datenfelder zusammenzufügen, legen Sie ein Datenfeld an, das groß genug ist, um die beiden anderen aufnehmen zu können. Das Zieldatenfeld `intField3` vom Typ Integer wird demnach mit 12 Elementen angelegt. Nun könnten Sie in einer oder zwei Schleifen die Quelldaten in das Zieldatenfeld kopieren. Wir wollen aber ohne Schleife auskommen, also benutzen wir `CopyMemory` und kopieren Speicherinhalte direkt an eine andere Stelle im Speicher. Das läuft extrem schnell ab.

Listing 5.5
Beispiele\02_Grundlagen_API\02_04_Allgemein.xls\mdlUnionArray

```
Private Declare Sub CopyMemory Lib "kernel32" _
   Alias "RtlMoveMemory" ( _
   pDst As Any, _
   pSrc As Any, _
   ByVal ByteLen As Long)

Sub UnionArray()
   Dim lngCount      As Long
   Dim intField1(5)  As Integer
   Dim intField2(5)  As Integer
   Dim intField3(11) As Integer
   Dim strMsg        As String
```

Listing 5.5 (Forts.)
Beispiele\02_Grundlagen_API\
02_04_Allgemein.xls\
mdlUnionArray

```
    ' Datenfelder mit Werten füllen, Index beginnt bei null
    For lngCount = 0 To 5
       intField1(lngCount) = lngCount
       intField2(lngCount) = lngCount + 6
    Next

    ' Vom Datenfeld intField1 ab Element null 12 Bytes
    ' in das Feld intField3 ab Element null kopieren
    CopyMemory intField3(0), intField1(0), 12

    ' Vom Datenfeld intField2 ab Element null 12 Bytes
    ' in das Feld intField3 ab Element sechs kopieren
    CopyMemory intField3(6), intField2(0), 12

    ' Message für Datenfeld 1
    strMsg = strMsg & "Feld 1 = "
    For lngCount = 0 To 5
       strMsg = strMsg & intField1(lngCount) & ";"
    Next

    ' Message für Datenfeld 2
    strMsg = Left(strMsg, Len(strMsg) - 1) & vbCrLf & "Feld 2 = "
    For lngCount = 0 To 5
       strMsg = strMsg & intField2(lngCount) & ";"
    Next

    ' Message für Datenfeld 2
    strMsg = Left(strMsg, Len(strMsg) - 1) & vbCrLf & "Feld 3 = "
    For lngCount = 0 To 11
       strMsg = strMsg & intField3(lngCount) & ";"
    Next

    ' Semikolon abschneiden
    strMsg = Left(strMsg, Len(strMsg) - 1)

    ' Ergebnis ausgeben
    MsgBox strMsg
End Sub
```

Das Zielarray `intField3` nimmt im Speicher `(12 mal Integer (12*2 Bytes)) = 24 Bytes` als Speicherplatz für die reinen Daten in Anspruch.

Die beiden anderen Arrays benötigen jeweils die Anzahl von `(6*2 Bytes)=12 Bytes` im Speicher.

Übergeben Sie eine Variable als Referenz an eine Funktion oder Prozedur, wird die Adresse der Speicherstelle auf den Stack gelegt, ab der die eigentlichen Daten stehen. Die aufgerufene Prozedur oder Funktion, hier `CopyMemory`, holt sich dann von diesem Stack die erwarteten Parameter ab, in diesem Fall also einen Zeiger auf die Daten.

Bei der ersten Aktion mit `CopyMemory` werden 12 Bytes ab der übergebenen Speicheradresse `intField1(0)` in den Speicher ab Speicheradresse `intField3(0)` kopiert.

Danach werden 12 Bytes ab Speicheradresse `intField2(0)` in den Speicher ab Speicheradresse `intField3(6)` kopiert und das Datenfeld `intField3` enthält nun die alle Quelldaten.

Beim Benutzen von `CopyMemory` ist es sehr leicht möglich, Speicherbereiche zu überschreiben, die andere Daten enthalten und eigentlich tabu sein sollten. Es werden keinerlei Rückfragen gestellt und auch ein Undo ist nicht möglich. Der Speicher ist in diesem Fall wirklich unwiederbringlich überschrieben worden. Sie sollten also immer genau wissen, was Sie machen und vor dem Ausprobieren jedes Mal die Mappe speichern.

5.11.3 Puffer

Relativ häufig stehen Sie beim Umgang mit der API vor der Aufgabe, einen Puffer bereitzustellen, der Daten aufnehmen kann. In vielen Fällen ist dabei ein Stringpuffer in der entsprechenden Größe ausreichend. Das gilt insbesondere in den Fällen, in denen ein String im `ANSI`- oder `ASCII`-Format zurückgeliefert werden soll. Ein solcher Puffer wird folgendermaßen angelegt, hier beispielsweise für die Aufnahme von fünf Zeichen:

```
Dim strPuffer As String
strPuffer=String(5,0)
```

Sie können den Puffer selbstverständlich auch mit einem anderen Zeichen füllen, ich ziehe es aber vor, den Zeichencode null zu benutzen.

VBA legt grundsätzlich bei der Übergabe an eine API-Funktion eine `ANSI`-Kopie des von VBA intern benutzten Unicodestrings im Speicher an und übergibt die Adresse, ab der diese Kopie zu finden ist. Die API-Funktion manipuliert dann den zur Verfügung gestellten Speicherbereich und nach der Rückkehr der Funktion kopiert VBA den String wieder als Unicode zurück an den eigentlichen Speicherplatz des Strings.

In einigen Fällen muss es jedoch ein bisschen mehr sein, beispielsweise wenn Sie einen Unicodestring an eine API übergeben müssen oder einen solchen zurückbekommen wollen. Wie schon angesprochen, spricht in einem solchen Fall erst einmal die automatische Umwandlung bei der Übergabe gegen einen Stringpuffer.

Im Allgemeinen lässt es sich dabei in den meisten Fällen leichter mit Bytearrays arbeiten. Aber auch der Einsatz eines Strings ist grundsätzlich noch möglich. Sie müssen nur den Stringpuffer vorher anpassen.

So wird ein String für die Übergabe an eine API-Unicodefunktion vorbereitet:

```
strBuff = StrConv(strBuff, vbUnicode)
```

Sie werden feststellen, dass der String `strBuff` jetzt doppelt so lang ist. Jedes zweite Zeichen ist ein `Chr(0)`. Aber das ist nicht alles. Da der String vorher schon Unicode war und somit im Speicher je Zeichen 2 Bytes eingenommen hat und nach der Umwandlung immer noch Unicode ist, werden also pro ursprüng-

liches Zeichen tatsächlich 4 Bytes belegt. Dass es wirklich vier Bytes pro ursprüngliches Zeichen sind, können Sie testen, indem Sie im Direktbereich der VBE folgende Zeile eingeben und mit ↵ abschließen:

```
Print LenB(StrConv("1", vbUnicode))
```

Die Ausgabe ist in diesem Fall 4. Bei der Übergabe wird von VBA daraus ein ANSI-String, bei dem aus jedem Zeichen je ein Byte eliminiert ist. Im Speicher, in dem die Kopie abgelegt ist, liegt somit das ursprüngliche Zeichen als Unicode vor. Zur Verdeutlichung eine kleine Tabelle (Tabelle 5.7).

Tabelle 5.7
Unicodeübergabe als String

Ursprüngliches Zeichen	1
Bytefolge im Speicher	49 0
Bei der direkten Übergabe an eine API als Bytefolge im Speicher	49
Zeichenfolge nach der Umwandlung mit StrConv	1 –
Bytefolge im Speicher	49 0 0 0
Bei der Übergabe an eine API als Bytefolge im Speicher. Das zweite und vierte Byte wird dabei eliminiert.	49 0

Um das Gleiche mit einem Bytearray zu machen, brauchen Sie lediglich ein dynamisches Bytearray und können sich die Umwandlung mit `StrConv` sparen. Das dynamische Bytearray wird wie folgt deklariert:

```
Dim abytBuff() As Byte
```

Sie können das Bytearray von nun an wie eine normale Stringvariable behandeln.

```
abytBuff = "1"

MsgBox abytBuff

MsgBox Left(abytBuff, 1)
```

Es ist sogar Left und Co. möglich. Zur Verdeutlichung nachfolgend eine Tabelle (Tabelle 5.8).

Tabelle 5.8
Dynamische Byte-Arrays als Stringpuffer

Ursprüngliches Zeichen	1
Das Element `abytBuff(0)` nach der Zuweisung	49
Das Element `abytBuff(1)` nach der Zuweisung	0
Ab Speicherstelle `abytBuff(0)` als Bytefolge im Speicher	49 0
Die Ausgabe mit `MsgBox abytBuff`	1
Die Ausgabe mit `MsgBox Left(abytBuff, 1)`	1

Das Element abytBuff(0) enthält also in diesem Fall den Wert 49 und abytBuff(1) den Wert null. Wie man sieht, hat das erste Element des Arrays den Index null. Das Bytearray hat jetzt insgesamt zwei Elemente und enthält demnach das zugewiesene Zeichen in Unicode. Aber wie übergeben Sie nun diesen Puffer an eine Funktion?

Das ist eigentlich ganz einfach. Sie müssen nur das erste Element als Referenz, also ByRef, übergeben. Ich wiederhole es noch einmal und werde auch an anderer Stelle immer wieder darauf eingehen, weil es so etwas Grundlegendes ist, das in Fleisch und Blut übergehen sollte. Wird eine Variable als Referenz übergeben, wird nicht etwa der Wert auf den Stack gelegt, sondern ein Zeiger auf die Speicherstelle.

Das eröffnet beim Benutzen von Byte-Arrays ganz neue Perspektiven. Da ein beliebiges Element übergeben werden kann, ist es möglich, ein größeres Bytearray in verschiedene Häppchen aufzuteilen. Zusammen mit dem Wissen über die Belegung im Speicher können Sie sich beispielsweise einen String ab dem 11. Element mittels CopyMemory in den Speicher schreiben lassen, danach einen anderen ab Index 0 und so weiter und so fort.

5.11.4 Safearrays

Vorhin habe ich geschrieben, dass es über Arrays noch mehr zu berichten gibt. Woher weiß zum Beispiel Visual Basic, um was für ein Array es sich überhaupt handelt. Die Adresse des ersten Elements langt nicht, um solch ein Array zu verwalten. VBA muss die Länge der einzelnen Elemente kennen, die oberen und unteren Grenzen und die Anzahl der Dimensionen müssen auch bekannt sein. Erst damit ist ein problemloser und schneller Zugriff auf die einzelnen Elemente möglich.

Die Informationen dazu sind in einer Safearray-Struktur abgelegt.

```
Private Type SAFEARRAY
   cDims          As Integer
   fFeatures      As Integer
   cbElements     As Long
   cLocks         As Long
   pvData         As Long
   Bounds(0 To 0) As SAFEARRAYBOUND
End Type
```

- cDims

  Dieser Integerwert gibt die Anzahl der Dimensionen des Arrays an.

- FFeatures

  Dieser Integerwert enthält Informationen über den im Array verwendeten Datentyp. Die einzelnen gesetzten Flags haben folgende Bedeutung (Tabelle 5.9).

Tabelle 5.9
Die Flags des Elements FFeatures der Struktur SAFEARRAY

Bitnummer	Bedeutung
0	(&H1) Das Array ist auf dem Stack abgelegt.
1	(&H2) Das Array ist statisch.
2	(&H4) Bedeutung unbekannt. Lt. MSDN eingebettet in eine Struktur.
3	(&H10) Die Grenzen des Arrays sind nicht veränderbar.
8	(&H100) Das Array ist ein Stringarray.
9	(&H100) Bedeutung unbekannt. Lt. MSDN ein Array der Schnittstelle IUnknown.
10	(&H400) Bedeutung unbekannt. Lt. MSDN ein Array der Schnittstelle iDispatch.
11	(&H800) Das Array ist ein Variant-Array.

- `cbElements`

  Dieser Longwert gibt die Größe eines Elements in Bytes an.

- `cLocks`

  Dieser Longwert gibt den Zustand des aktuellen Sperrzählers wieder.

- `pvData`

  Dieser Longwert ist ein Zeiger auf das erste Element der eigentlichen Daten.

- `Bounds`

  Dieses Array der Struktur `SAFEARRAYBOUND` stellt die Untergrenze und die Anzahl der Elemente jeder Dimension dar. Für jede Dimension gibt es dafür ein Element im Array. Hier die Struktur `SAFEARRAYBOUND`:

```
Private Type SAFEARRAYBOUND
   cElements   As Long
   lLbound     As Long
End Type
```

  - `cElements`

    Die Anzahl der Elemente dieser Dimension.

  - `lLbound`

    Die untere Grenze dieser Dimension.

Wie kommen Sie nun an die `Safearray`-Struktur?

Handelt es sich um einen Variant mit dem Datentyp `Array`, gehen Sie folgendermaßen vor:

```
Dim lngSafearray As Long
' Liefert einen Zeiger auf die Safearray-Struktur
CopyMemory lngSafearray, ByVal (VarPtr(varTest) + 8), 4
```

Diese Zeilen liefern in der Variablen `lngSafearray` einen Zeiger auf die Safearray-Struktur, vorausgesetzt der Variant `varTest` besitzt auch tatsächlich den

Untertyp Array. Andernfalls kann die Anwendung abstürzen. Deshalb sollten Sie vorher die Funktion `VarType` bemühen, um sicherzustellen, dass Sie es mit dem richtigen Typ zu tun haben. Die Funktion `VarType` ist in Kapitel 1 näher erläutert.

Um die Funktion zu verstehen, müssen Sie erst einmal wissen, wie ein Variant überhaupt aufgebaut ist. Ab der Speicherstelle, welche die undokumentierte Funktion `VarPtr` liefert, befinden sich die Verwaltungsinformationen (Tabelle 5.10) vom Variant-Datentyp.

Tabelle 5.10
Verwaltungsinformationen Variant

Bytenummer	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**	**15**	**16**
Doppelwort	**DWord #1**				**DWord #2**				**DWord #3**				**DWord #4**			
	Datentyp, wie er von der Funktion `VarType` geliefert wird.				Leer				Ab Byte 9 findet man die Daten, zum Teil als Zeiger auf die eigentlichen Speicheradressen. Die Anzahl der Bytes ist abhängig vom Datentyp.							

- Doppelwort #1

  Das erste Doppelwort enthält die Informationen über den Datentyp, und zwar in Form eines Werts, der auch von der Funktion `VarType` geliefert wird.

- Doppelwort #2

  Leer

- Ab Doppelwort #3

  Ab Bytenummer 9 ist der eigentliche Wert zu finden. Bei den Datentypen String und Array finden Sie dort einen Zeiger. Beim Datentyp String verweist dieser Zeiger auf das erste Zeichen des Strings. Das Doppelwort vor diesem ersten Zeichen, also die Speicheradresse des Strings minus 4, enthält noch die Länge des Strings in Bytes. Diese Längenangabe erfolgt ohne die Einbeziehung der zwei abschließenden Nullen, die das Ende des Strings kennzeichnen. Beim Variant-Array steht am dritten Doppelwort die gesuchte Speicherstelle der `Safearray`-Struktur, welche die eigentlichen Informationen zum Array enthält.

Das Element `pvData` in der gefundenen `Safearray`-Struktur zeigt dann auf den Speicherbereich, der die Daten des Arrays enthält. Bei Arrays mit festen Datenlängen finden Sie dort auch tatsächlich die Daten in der Anordnung, wie sie schon unter Arrays im Speicher (Kapitel 5.11.2) beschrieben wurde.

Bei einem Array mit Elementen des Datentyps `Variant` sind dort je 16 Bytes für die Variant-Verwaltungsinformationen der einzelnen Elemente gespeichert. Noch etwas anders verhält es sich bei `String`-Arrays. Dort zeigt das Element `pvData` auf einen Speicherbereich, bei dem für jedes Element des Arrays 4 Bytes reserviert sind. Es stehen dort aber seltsamerweise weder Daten noch irgendwelche Zeiger auf Speicherbereichen. Ich habe mir sagen lassen, dass man ohne eine TypeLibrary gar nicht an die Speicherstellen der eigentlichen Daten herankommt.

Um bei einem normalen Array an die Safearray-Struktur zu kommen, müssen Sie etwas tricksen und folgendermaßen vorgehen:

Listing 5.6
Beispiele\02_Grundlagen_API\02_04_Allgemein.xls\mdlUnionArray

```
Private Declare Function VarPtrArray Lib "msvbvm60.dll" _
   Alias "VarPtr" ( _
   ArrPtr() As Any _
   ) As Long

Private Declare Sub CopyMemory Lib "kernel32" _
   Alias "RtlMoveMemory" ( _
   pDst As Any, _
   pSrc As Any, _
   ByVal ByteLen As Long)

Private Sub ArrayPtr()
   Dim alngArray(1 To 10) As Long
   Dim lngPointerToPointer As Long
   Dim lngSafearray As Long

   ' Zeiger auf Zeiger auf die Safearray-Struktur
   lngPointerToPointer = VarPtrArray(alngArray)

   ' Den eigentlichen Zeiger in die Longvariable kopieren
   CopyMemory lngSafearray, ByVal lngPointerToPointer, 4
End Sub
```

Über die API-Funktion VarPtrArray aus der Bibliothek msvbvm60.dll erhalten Sie einen Zeiger auf einen Zeiger auf die Safearray-Struktur. Um den eigentlichen Zeiger auf die Safearray-Struktur zu erhalten, müssen Sie CopyMemory einsetzen. Hat man durch die Speicherschiebereien schließlich den Zeiger zur Safearray-Struktur in der Variablen lngSafearray stehen, gilt alles Weitere wie auch beim Variant-Array. Mit einem kleinen feinen Unterschied: Anders als bei Variant-Arrays können bei einem normalen Array auch String-Arrays fester Länge vorkommen. In diesem Fall zeigt das Element pvData der Safearray-Struktur auf einen Speicherbereich, der die tatsächlichen Stringdaten enthält, und zwar in Unicode.

Excel bietet ein Feature, das wenig bekannt ist, aber meiner Ansicht nach enorme Vorteile bietet. Sie können sich nämlich in Excel-VBA einen rechteckigen Bereich samt Inhalt aus einem Tabellenblatt in ein Array »beamen«. Dazu brauchen Sie nur einer Variant-Variablen einen Bereich zuweisen und können somit den Einsatz einer Schleife vergessen.

```
Dim MyRange As Variant
MyRange = Worksheets(1).Range("A1:C3")
```

Und schon haben Sie auf einen Rutsch ein zweidimensionales Variantarray mit dem Zellinhalt des Bereichs. Die erste Dimension spiegelt die Zeile, die zweite die Spalte wieder. Das Array können Sie bearbeiten und genauso leicht wieder in ein Tabellenblatt in einen entsprechend großen Bereich einfügen.

```
Worksheets(2).Range("C3:E5")= MyRange
```

Der Vorteil liegt in der erheblich höheren Verarbeitungsgeschwindigkeit. Anstatt aus einem `Range` laufend die Daten auszulesen, kopieren Sie sich diesen Bereich einmal in den Speicher und greifen dann nur noch auf das Array zu.

Mit dem Wissen um die `Safearray`-Struktur könnten Sie ohne eine einzige Schleife aus einem mehrdimensionalen Array ein eindimensionales machen. Wenn es nicht auf die Reihenfolge der Daten ankommt, können Sie die Dimensionen mit ein paar Zeilen Code tauschen, also beispielsweise ein Datenfeld(5,7) in ein Datenfeld(7,5) verwandeln.

Wenn Sie richtig Transponieren wollen, müssen Sie aber auch hier Schleifen benutzen. Bei Variant-Arrays mit dem Untertyp `Variant` müssen aber keine Daten, sondern nur die Variantheader umkopiert werden. Solche Arrays bekommen Sie beispielsweise, wenn Sie einer `Variant`-Variablen einen Bereich (`Range`) zuweisen.

Folgendes Listing (Listing 5.7) transponiert einen Bereich im Speicher und fügt ihn wieder in das Blatt ein.

Listing 5.7
Beispiele\02_Grundlagen_API\
02_04_Allgemein.xls\
mdlTransponieren

```
Private Declare Sub CopyMemory Lib "kernel32" _
   Alias "RtlMoveMemory" ( _
   pDst As Any, _
   pSrc As Any, _
   ByVal ByteLen As Long)

Private Sub TransponseRange()
   Dim varTrans As Variant

   With Worksheets("Transponieren")

      ' Zielbereich löschen
      .Range("C12:H17").ClearContents

      ' Quellbereich füllen
      .Range("A12") = "A12"
      .Range("B12") = "B12"
      .Range("A12:B12").AutoFill Destination:=Range("A12:B17")

      ' Bereich in ein Array kopieren
      varTrans = MyTransArray(.Range("A12:B17"))

      ' Variant-Array in den Zielbereich einfügen, dabei
      ' die Größe des Arrays beachten.
      .Range(.Cells(12, 3), .Cells( _
         12 - 1 + UBound(varTrans, 1), _
          3 - 1 + UBound(varTrans, 2)) _
         ) = varTrans
   End With
End Sub
```

Listing 5.7 (Forts.)
Beispiele\02_Grundlagen_API\02_04_Allgemein.xls\mdlTransponieren

```
Private Function MyTransArray(rngSource As Range) As Variant
    Dim i             As Long
    Dim k             As Long
    Dim m             As Long
    Dim acurOld()     As Currency
    Dim acurNew()     As Currency
    Dim lngCount      As Long
    Dim varRange      As Variant
    Dim lngSafeArray  As Long
    Dim lngData       As Long
    Dim lngCountX     As Long
    Dim lngCountY     As Long
    Dim lngLboundX    As Long
    Dim lngLboundY    As Long

    ' Range in ein zweidimensionales Array umwandeln
    varRange = rngSource

    ' Zeiger auf Safearray ermitteln
    CopyMemory lngSafeArray, ByVal (VarPtr(varRange) + 8), 4

    ' Zeiger auf Daten ermitteln
    CopyMemory lngData, ByVal lngSafeArray + 12, 4

    ' Für beide Dimensionen die Untergrenze und
    ' die Anzahl der Elemente ermitteln
    CopyMemory lngCountX, ByVal lngSafeArray + 16, 4
    CopyMemory lngLboundX, ByVal lngSafeArray + 20, 4
    CopyMemory lngCountY, ByVal lngSafeArray + 24, 4
    CopyMemory lngLboundY, ByVal lngSafeArray + 28, 4

    ' Gesamte Anzahl der Elemente errechnen
    lngCount = (lngCountX * lngCountY)

    ' Mehr als 256 Spalten sind ungewöhnlich
    If lngCountY > 256 Then MsgBox "Zu viele Zeilen!": Exit Function

    ' ######################################################
    ' Wenn die Reihenfolge der Daten egal ist, kann Folgendes
    ' weggelassen werden.

    ' Zwischenpuffer für die Daten schaffen.
    ' Jedes Element braucht 16 Byte, also zweimal Currency.
    ' An dieser Stelle stehen nicht die eigentlichen Daten,
    ' sondern nur Variantinfos mit den Zeigern auf die Daten
    ReDim acurOld(1 To 2, 1 To lngCount)
    ReDim acurNew(1 To 2, 1 To lngCount)

    ' Die Verwaltungsinfos der Daten in den Zwischenpuffer kopieren
    CopyMemory acurOld(1, 1), ByVal lngData, lngCount * 16

    ' Die Verwaltungsinfos der Daten tauschen und in den
    ' Zwischenpuffer acurNew schreiben
```

Listing 5.7 (Forts.)
Beispiele\02_Grundlagen_API\02_04_Allgemein.xls\mdlTransponieren

```
    For i = 1 To lngCountY
        For k = 1 To lngCountX
            m = m + 1
            acurNew(1, m) = acurOld(1, (k - 1) * lngCountY + i)
            acurNew(2, m) = acurOld(2, (k - 1) * lngCountY + i)
        Next
    Next

    ' Die getauschten Verwaltungsinfos der Daten an die
    ' Originalposition zurückkopieren
    CopyMemory ByVal lngData, acurNew(1, 1), lngCount * 16

    ' Wenn die Reihenfolge der Daten egal ist, kann bis hierhin
    ' alles weggelassen werden.
    ' ########################################################

    ' Jetzt wird gelogen und behauptet, dass die erste Dimension
    ' die Anzahl der Elemente der zweiten hat, und umgekehrt
    CopyMemory ByVal lngSafeArray + 16, lngCountY, 4
    CopyMemory ByVal lngSafeArray + 20, lngLboundY, 4
    CopyMemory ByVal lngSafeArray + 24, lngCountX, 4
    CopyMemory ByVal lngSafeArray + 28, lngLboundX, 4

    ' Geänderten Variant zurückgeben
    MyTransArray = varRange

End Function
```

Die unteren oder oberen Grenzen eines Arrays lassen sich beliebig anpassen, wenn dabei die Anzahl der Elemente gleich bleibt, wie folgendes Beispiel (Listing 5.8) zeigt:

Listing 5.8
Beispiele\02_Grundlagen_API\02_04_Allgemein.xls\mdlLBoundchange

```
Private Type SAFEARRAYBOUND
    cElements   As Long
    lLbound     As Long
End Type

Private Type SAFEARRAY
    cDims          As Integer
    fFeatures      As Integer
    cbElements     As Long
    cLocks         As Long
    pvData         As Long
    Bounds(0 To 0) As SAFEARRAYBOUND
End Type

Private Declare Function VarPtrArray Lib "msvbvm60.dll" _
    Alias "VarPtr" ( _
    ArrPtr() As Any _
    ) As Long

Private Declare Sub CopyMemory Lib "kernel32" _
    Alias "RtlMoveMemory" ( _
    pDst As Any, _
    pSrc As Any, _
    ByVal ByteLen As Long)
```

Listing 5.8 (Forts.)
Beispiele\02_Grundlagen_API\
02_04_Allgemein.xls\
mdlLBoundchange

```
Private Sub LboundChange()
   Dim alngArray(1 To 10)  As Long
   Dim lngPointerToPointer As Long
   Dim lngSafearray        As Long
   Dim udtSafearray        As SAFEARRAY
   Dim strMsg              As String

   ' Zeiger auf Zeiger auf die Safearray-Struktur
   lngPointerToPointer = VarPtrArray(alngArray)

   ' Den eigentlichen Zeiger in die Longvariable kopieren
   CopyMemory lngSafearray, ByVal lngPointerToPointer, 4

   ' Den Safearrayheader in die Struktur kopieren
   CopyMemory udtSafearray, ByVal lngSafearray, Len(udtSafearray)

   ' Ausgabe der Arrayinfos
   MsgBox Safearraystring(udtSafearray), , _
      "LBound(alngArray)=" & LBound(alngArray)

   ' Untere Grenze auf fünf ändern
   udtSafearray.Bounds(0).lLbound = 5

   ' Zurückkopieren in die Safearray-Struktur
   CopyMemory ByVal lngSafearray, udtSafearray, Len(udtSafearray)

   ' Ausgabe der unteren Grenze
   MsgBox "LBound(alngArray)=" & LBound(alngArray), , _
      "Nach der Manipulation"

End Sub

Private Function Safearraystring(udtSafearray As SAFEARRAY) _
   As String
   Dim strMsg As String

   ' Die Daten aus der Struktur auslesen
   With udtSafearray
      strMsg = "Dimensionen : " & .cDims & vbCrLf
      strMsg = strMsg & "Elementgröße : " & _
         .cbElements & vbCrLf

      ' Für jede Dimension eine Struktur SAFEARRAYBOUND
      ' Zählung beginnt bei Null
      With .Bounds(0)
         strMsg = strMsg & "Untere Grenze : " & _
            .lLbound & vbCrLf
         strMsg = strMsg & "Elemente : " & _
            .cElements & vbCrLf
      End With

      ' Als Funktionsergebnis zurückgeben
      Safearraystring = strMsg & "Datenzeiger : " & .pvData
   End With
End Function
```

5.12 Fenster

Ein grundlegendes Konzept von Microsofts Betriebssystemen ist, wie der Name Windows schon andeutet, das der Fenster. Nahezu alles, was als Form auf dem Bildschirm ausgegeben wird, ist ein Fenster. Diese Fenster haben die unterschiedlichsten Aufgaben:

- Sie dienen als Container für andere Fenster.
- Sie werden als Zeichenfläche benutzt.
- Mit ihnen werden Meldungen ausgegeben und Eingaben entgegengenommen.
- Sie dienen als Empfänger für Fensternachrichten.

Jeder Mausklick, jede Mausbewegung und alle Tastaturanschläge, die durchgeführt werden, werden vom Betriebssystem registriert und als Nachricht verpackt an das jeweilige Fenster geschickt. Beispielsweise geht der Tastendruck an das Fenster, das den Eingabefokus hat und der Mausklick an das Fenster, welches sich an der Klickposition befindet. Dabei läuft im Hintergrund eines jeden Fensters kontinuierlich ein Programm, das nur auf solche Meldungen wartet. Die empfangenen Meldungen werden auswertet und es wird entsprechend reagiert. Dieses im Hintergrund laufende Programm ist die so genannte `MessageLoop`. Als Reaktion auf Nachrichten werden zum Beispiel Ereignisse ausgelöst oder einfach nur die Zeichenfläche oder Teile davon neu gezeichnet.

Jedes Fenster hat seine eigene Zugriffsnummer, das so genannte Fensterhandle (`hWnd`). Das ist wie alle `Handles` ein Zeiger und somit ein Longwert. Über die verschiedensten API-Funktionen, in Verbindung mit dem Handle, haben Sie die Möglichkeit, nahezu jedes Fenster auf mannigfaltige Weise zu manipulieren. Das fängt an bei den Systemmenüs wie MAXIMIEREN, MINIMIEREN, SCHLIESSEN usw., geht weiter über die Größe, Form, Transparenz und ist noch lange nicht bei der Sichtbarkeit eines Fensters am Ende. Nahezu jede Eigenschaft lässt sich verändern, ob sie bei der vorliegenden Fensterklasse überhaupt sinnvoll ist und die Änderungen überhaupt sichtbare Wirkungen zeigen, sei dahingestellt.

Um diese wichtige Zugriffsnummer, das *Fensterhandle* zu finden, bedienen Sie sich meistens der Funktion `FindWindow`.

```
Private Declare Function FindWindow Lib "user32.dll"
   Alias "FindWindowA" ( _
   ByVal lpClassName As String, _
   ByVal lpWindowName As String, _
   ) As Long
```

Diese Funktion durchsucht alle Top-Level-Fenster, um das Fenster zu finden, welches den Suchkriterien entspricht. Die einzelnen Fenster sind in einer Baumstruktur angeordnet, mit dem Desktop als der Wurzel. Ein Top-Level-Fenster ist ein direktes Kindfenster des Desktops. Nur Fenster auf dieser Ebene können mit dieser Funktion gefunden werden. Kindfenster der Top-Level-Fenster oder deren Kinder müssen mit anderen Funktionen gesucht werden. Als Ergebnis liefert die Funktion `FindWindow` die Zugriffsnummer zurück. Ist das Ergebnis null, wurde kein entsprechendes Fenster gefunden.

Zwei Eigenschaften hat ein Fenster, über die es identifiziert werden kann. Das ist zum einen der Klassenname und zum anderen der Fenstertext. Die Fensterklasse ist eine Eigenschaft, die den Typ eines Fensters angibt, aus dem das Fenster abgeleitet ist. Der Fenstertext ist auch ein String und gibt den Text des Fensters in der Titelleiste an. Bei Fenstern ohne Titelleiste ist dies zum Teil auch die Beschriftung.

Über den Parameter `lpClassName` können Sie ein Fenster nach dem Klassennamen suchen. Wenn Sie den Klassennamen nicht kennen, übergeben Sie stattdessen einen Nullstring, das ist aber nicht das Gleiche wie ein Leerstring (““). Benutzen Sie dafür die Konstante `vbNullString`.

Über den Parameter `lpWindowName` suchen Sie ein Fenster nach dem Text in der Titelleiste. Wenn Sie den Titel nicht kennen, oder das Fenster keinen Fenstertitel hat, übergeben Sie einen Nullstring, verwenden Sie dafür die Konstante `vbNullString`.

Achtung

Damit einer der beiden Parameter bei `FindWindow` auch wirklich ignoriert wird, muss `vbNullString` übergeben werden, und nicht, wie man vielleicht annehmen könnte, ein leerer String. Übergeben Sie einen Leerstring (zwei Anführungszeichen), wird ein Fenster mit einem Leerstring als Klassenname gesucht, und das wird es sicherlich nicht geben. Wenn Sie null übergeben, macht die automatische Typenumwandlung einen String mit dem Zeichen »0« daraus. Solch ein Fenster werden Sie auch sehr selten finden.

Die API-Funktion `FindWindow` kann nach den Klassennamen, dem Fenstertext oder nach beiden suchen. Ich persönlich benutze zum Suchen meistens den Fenstertext. D.h., bei mir wird zum Ermitteln eines Fensterhandles fast ausschließlich der Parameter `lpWindowName` zum Suchen benutzt.

Der Klassenname in Verbindung mit dem Fenstertext wäre bei einer Suche zwar eindeutiger, ich habe es in Verbindung mit Excel aber dennoch gelassen. Dass ich bei Excel davon Abstand nehme, liegt daran, dass sich die Klassennamen der UserForms in den verschiedenen Officeversionen schon einmal geändert haben. Unter Excel 97 haben die UserForms den klangvollen Klassennamen `ThunderXFrame`, ab Excel 2002 hat man es mit der Klasse `ThunderDFrame` zu tun. Das ist natürlich fatal, wenn eine Mappe unter verschiedenen Excel-Versionen funktionieren soll. Und wenn das schon bei den UserForms so ist, können auch andere Fenster jederzeit einen anderen Klassennamen erhalten. Niemand wird Ihnen irgendeine Garantie geben, dass sich die Klassennamen unter den verschiedenen Versionen nicht ändern und schon gar nicht Microsoft.

Übrigens ist die Kombination Klassenname und Fenstertext auch nicht eindeutig, eine andere Instanz von Excel kann durchaus UserForms mit den gleichen Kombinationen besitzen. Wenn Sie trotzdem ganz sichergehen wollen, das richtige Fenster zu finden, müssen Sie den Zugriff auf die Eigenschaft `Caption` haben, denn es ist wirklich nicht sinnvoll, das Fenster zu suchen, um es dann umbeschriften zu können, damit Sie es daraufhin sicher finden können. Wenn

Sie also Zugriff auf die Eigenschaft `Caption` haben, speichern Sie diesen Wert, geben dem Fenster eine wirklich eindeutige `Caption`, suchen ein Fenster mit diesem Text und schreiben anschließend die ursprüngliche `Caption` wieder zurück.

Ein weiteres wichtiges Handle eines Fensters ist der `DeviceContext` (*hDC*). Das ist eine Zugriffsnummer auf die grafische Oberfläche eines Fensters. Wenn also auf einem Fenster etwas gezeichnet werden soll, brauchen Sie dessen `DC`. Es ist aber meistens nicht nur damit getan, den DC zu kennen, Sie müssen sich diesen auch ausleihen und nach erledigter Arbeit unverzüglich wieder zurückgeben. Wenn Sie dies nicht tun, gehen Sie sofort ins Gefängnis und ziehen nicht über Los. Und das für den Rest der Programmausführung.

Weiterhin ist die *Region* des Fensters ein wichtiger Begriff. Eine Region ist erst einmal eine Fläche, die nahezu jede beliebige Form annehmen kann. Mit verschiedenen API-Funktionen können Sie sich eigene Regionen erzeugen und diese miteinander kombinieren. Sie können zum Beispiel aus einer Region eine andere ausstanzen und vieles mehr. Das Besondere ist aber, dass Sie einem Fenster diese Region zuweisen können. Dann ist nur noch der Teil sichtbar, der von der Region überdeckt wird. Der andere Teil ist aber nicht nur unsichtbar, er existiert auch scheinbar gar nicht mehr. Den Klick auf einen transparenten Teil, beispielsweise durch ein Loch, enthält das darunter liegende Fenster. Durchlöcherte, runde, vieleckige usw. User Formen lassen sich sehr schön damit herstellen.

Menühandles eignen sich hervorragend zur Manipulation. Sie können programmgesteuert Menüpunkte anklicken, ausgrauen, Haken setzen, Icons einfügen und vieles andere mehr. Das funktioniert sogar mit fremden Anwendungen wie Notepad, sofern diese überhaupt noch Menüs besitzen. Sogar gesperrte Menüpunkte lassen sich so entsperren. Echte Menüs kommen aber (leider) in der letzten Zeit immer weniger vor.

5.13 Koordinaten, Einheiten

Die Maßeinheit für API-Funktionen ist ein `Pixel`, im Gegensatz zu `Twips` und `Points` bei anderen Anwendungen. Koordinaten beginnen links oben beim Punkt 0,0 und werden nach rechts unten größer. Selbstverständlich sind auch negative Koordinaten erlaubt, dann liegt der Punkt eben außerhalb des sichtbaren Bereichs. Viele Funktionen liefern Koordinaten in Bezug auf den Screen und nicht auf den Container, in dem sie stecken. In diesem Fall ist dann Umrechnen gefragt.

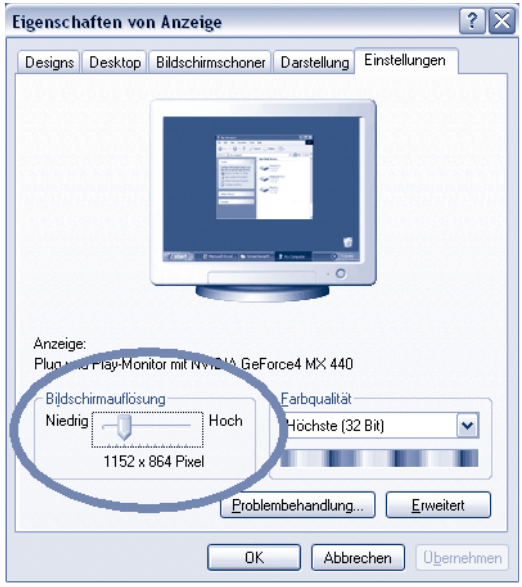
Die Größenangaben einer UserForm, die Sie beispielsweise mit `Me.Width` ermitteln, sind aber keine Angaben in Pixel. Die Maßeinheit, mit der Sie es dabei zu tun haben, ist ein Punkt, der 1/72 Zoll oder 0,35 mm entspricht.

Twips sind noch etwas kleiner. Ein Twip entspricht 1/20 Punkt, also 1/1440 Zoll.

Die Maßeinheiten Zoll und somit auch Millimeter sind in diesem Kontext aber keine physikalischen Einheiten. Im Zusammenhang mit Größen von grafischen Elementen ist das ein virtuelles Maß und entspricht nicht einem physikalischen Zoll oder Millimeter.

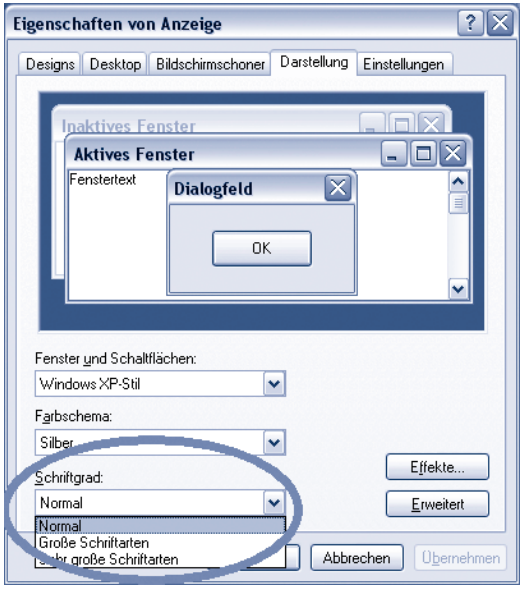
Die tatsächlich dargestellte Größe auf dem Monitor ist abhängig von zwei Einstellungen. Da ist einmal die Bildschirmauflösung (Abbildung 5.1) in `Pixel`, die eingestellt wurde, also zum Beispiel 800 x 600 oder 1152 x 864 Pixel.

Abbildung 5.1
Bildschirmauflösung



Der Schriftgrad (Abbildung 5.2), der eigentlich die virtuelle Auflösung eines virtuellen Zolls in `dpi` (`Dots` per `Inch`) angibt, legt zusammen mit der Bildschirmauflösung die tatsächliche Größe eines virtuellen Zolls fest.

Abbildung 5.2
Virtuelle Auflösung (dpi)



Nimmt man als Beispiel einen normalen 17-Zoll-Monitor, hat dieser eine physikalische Breite von etwa 13 Zoll. Das Verhältnis Diagonale zu Breite habe ich in den folgenden Betrachtungen mit `1,33:1` angenommen.

Wenn eine Bildschirmauflösung von 1024 Pixel in der x-Richtung gewählt wird, ist ein virtuelles Zoll bei einer Auflösung von 96 dpi rund 1,22 Zoll groß. Gerechnet wird dabei so:

1024/96 ergibt 10,66 virtuelle Zoll. Also ist bei einer tatsächlichen Breite von 13 Zoll jedes virtuelle Zoll 1,22-mal so groß, also 1,22 echte Zoll.

Wenn eine Bildschirmauflösung von 800 Pixel in der x-Richtung gewählt wird, ist ein virtuelles Zoll bei einer Auflösung von 96 dpi rund 1,56 Zoll groß.

Noch ein Beispiel:

Sie haben einen 14-Zoll-Monitor auf ihrem Laptop mit einer x-Bildschirmauflösung von 1024 Pixel. Es sind normale Schriftarten mit 96 dpi eingestellt. Wie breit wird eine UserForm mit Width = 240 Punkten dargestellt?

Lösung:

1024/96 = 10,66 = Virtuelle Breite in Zoll in der x-Richtung

14/1,33 = 10,52 = Tatsächliche Breite in Zoll in der x-Richtung

Daraus folgt = 1 Zoll ~ 1 virtueller Zoll

240/72 = 3,33 = Die Form ist 3,33 Zoll, also ca. 8,5 cm breit

5.14 Weiterführende Quellen

Wer der englischen Sprache mächtig ist, der sollte versuchen, sich den *Visual Basic Programmer's Guide to the Win32 API* von Dan Appleman zu beschaffen. Das ist meiner Ansicht nach eines der besten Bücher zu diesem Thema. Das Puzzlebuch von Dan Appleman ist zwar in Deutsch erhältlich, wendet sich aber eher an fortgeschrittenere API-Benutzer.

Bruce McKinney ist auch einer der Cracks auf diesem Gebiet. Suchen Sie einfach einmal nach diesem Namen. Sein Buch *Hardcore Visual Basic* ist, soweit ich weiß, von ihm selbst im Internet als PDF zur Verfügung gestellt worden.

Im Internet unter *http://www.mentalis.org/agnet/apiguide.shtml* finden Sie den API-Guide. Das ist ein Programm, in dem knapp 1000 API-Funktionen beschrieben werden. Zu allen Funktionen ist auch ein Beispiel zu finden. Das ist eine hervorragende Anlaufstelle, wenn Sie Beispiele zu einer Funktion suchen. Eine weitere ergiebige Onlinequelle ist Google und besonders das Newsgrouparchiv *groups.google.com*.

6

Dialoge

6.1 Was Sie in diesem Kapitel erwartet

Die Interaktion mit dem Benutzer erfolgt normalerweise über Dialoge. Excel besitzt eine ganze Menge davon, mit denen man auch alle möglichen Einstellungen vornehmen kann.

Leider reichen all diese eingebauten Dialoge nicht immer aus. So gibt es beispielsweise keinen vernünftigen Dialog, mit dem man so etwas Profanes wie ein Verzeichnis auswählen kann. Windows selbst bietet aber solche Dialoge an, die man mit ein paar API-Funktionen auch für sich selbst benutzen kann. In diesem Kapitel werden ein paar von diesen Dialogen vorgestellt, wie die zur Verzeichnis-, Farb- oder Schriftartauswahl.

Des Weiteren wird gezeigt, wie Sie eine verbesserte Messagebox einsetzen und solch eine Box auch nach Ablauf einer gewissen Zeit wieder automatisch schließen können. Außerdem erfahren Sie, wie Sie die Dialoge zum Ändern der Systemeinstellungen aufrufen.

Wenig bekannt ist auch, dass es in Excel-VBA eine `InputBox`-Methode und eine `InputBox`-Funktion gibt, die unterschiedliche Funktionalitäten bereitstellen. Auch diese Unterschiede werden in diesem Kapitel beleuchtet.

Schließlich wird auch der Windows-interne Dialog zur Dateiauswahl vorgestellt, der einige Vorteile gegenüber den eingebauten Dialogen `GetSaveAsFilename` und `GetOpenFilename` bietet. So kann dort der Anfangspfad vorgegeben werden und es muss dazu nicht das aktuelle Verzeichnis mit `ChDir` angepasst werden.

6.2 Eingebaute Dialoge

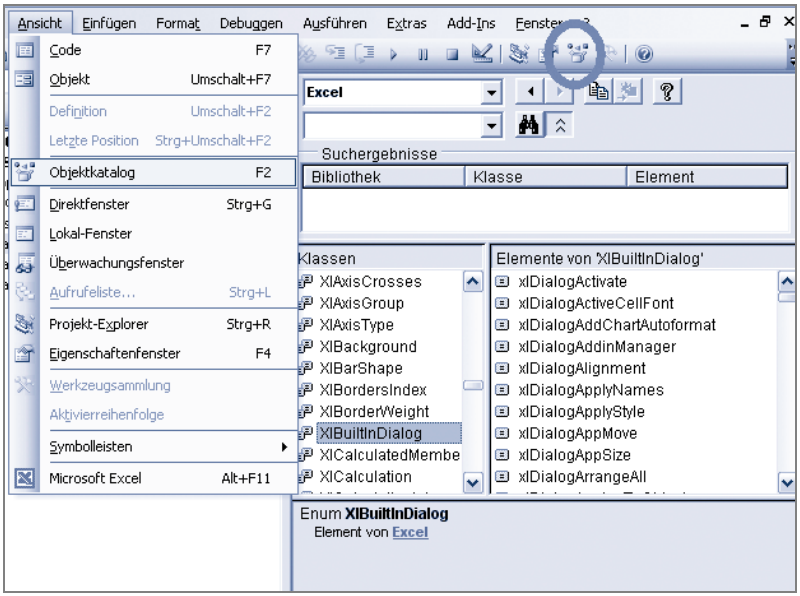
Excel besitzt etwa 200 fest eingebaute Dialoge. Diese können Sie programmgesteuert aufrufen und verwenden. Mit der folgenden Codezeile wird solch ein Dialog aufgerufen:

```
Application.Dialogs(Dialogkonstante).Show
```

Nachdem Sie die Klammer hinter `Dialogs` eingegeben haben, stellt Ihnen Intellisense eine Auswahl der möglichen Typen zur Verfügung, die Sie wählen können. Welche Typen generell möglich sind, entnehmen Sie am besten dem Objektkatalog.

Dazu wählen Sie in der Entwicklungsumgebung von Excel (VBE) den Menüpunkt ANSICHT | OBJEKTKATALOG, drücken die Taste F2 oder klicken auf das entsprechende Symbol (Abbildung 6.1).

Abbildung 6.1
Eingebaute Dialoge im Objektkatalog



Mit nachfolgenden Code (Listing 6.1) können Sie nacheinander die Dialoge aufrufen, ob diese nun in der momentanen Situation gerade sinnvoll sind oder nicht.

Listing 6.1
Beispiele\06_Dialoge\06_01_Dialoge.xls\mdlBuiltInDialog

```
Private Sub TestBuildInDialog()
   Dim i          As Long
   Dim varDummy   As Variant

   On Error Resume Next
   For i = 1 To 900
      Err.Clear
      Application.Dialogs(i).Show
      If Err.Number = 0 Then
         If MsgBox("Wollen Sie den nächsten" & _
            " Dialog aufrufen ?", vbOKCancel, _
            "Dialog Nr.:" & i) <> vbOK _
            Then Exit Sub
      End If
   Next
End Sub
```

6.3 Dialoge zum Ändern der Systemeinstellungen

Die Dialoge zum Ändern der Systemeinstellungen werden mit Hilfe von Dateien mit der Endung *.cpl* aufgerufen. Mit VBA können Sie diese Dialoge ohne Probleme aufrufen. Folgendes Listing (Listing 6.2) zeigt, wie das bewerkstelligt wird.

Listing 6.2
Beispiele\06_Dialoge\06_01_Dialoge.xls\mdlSysDlg

```
Public Sub ShowMyDialog()
   Dim strDialog(1 To 16) As String

   strDialog(1) = "access.cpl"     ' Eingabehilfen
   strDialog(2) = "appwiz.cpl"     ' Software
   strDialog(3) = "desk.cpl"       ' Anzeige
   strDialog(4) = "hdwwiz.cpl"     ' Hardwareassistent
   strDialog(5) = "inetcpl.cpl"    ' Internet
   strDialog(6) = "intl.cpl"       ' Region- und Sprachoptionen
   strDialog(7) = "joy.cpl"        ' Gamecontroller
   strDialog(8) = "main.cpl"       ' Mauseigenschaften
   strDialog(9) = "mmsys.cpl"      ' Sound- und Audiogeräte
   strDialog(10) = "ncpa.cpl"      ' Netzwerkverbindungen
   strDialog(11) = "nusrmgr.cpl"   ' Benutzerkonten
   strDialog(12) = "odbccp32.cpl"  ' ODBC
   strDialog(13) = "powercfg.cpl"  ' Energieoptionen
   strDialog(14) = "sysdm.cpl"     ' Systemeigenschaften
   strDialog(15) = "telephon.cpl"  ' Telefon, Modem
   strDialog(16) = "timedate.cpl"  ' Zeiteinstellungen

   ' Eingabehilfen, 4. Reiter Maus
   Shell "rundll32.exe shell32.dll," & _
        "Control_RunDLL " & strDialog(1) & ",,4"
End Sub
```

6.4 Ausgabe von Meldungen

6.4.1 MsgBox, MessageBoxA

Meldungen können mit einer Messagebox ausgeben werden, indem Sie die in VBA eingebaute Funktion `MsgBox` benutzen. Diese Funktion zeigt eine Meldung in einem Dialogfeld an und wartet darauf, dass auf einen Button geklickt oder dieser anderweitig betätigt wird. Zurückgegeben wird dann ein Wert, der anzeigt, welche Schaltfläche der Benutzer betätigt hat.

Diese `MsgBox`-Funktion ist eigentlich eine gekapselte Version der Windows-internen API-Funktion `MessageboxA`.

Die `MsgBox`-Funktion hat folgende Syntax:

```
MsgBox(prompt[, buttons] [, title] [, helpfile, context])
```

Die API-Funktion `MessageBoxA` wird folgendermaßen deklariert:

```
Private Declare Function MessageBox Lib "user32" Alias "MessageBoxA"
( ByVal hwnd As Long, ByVal lpText As String, ByVal lpCaption As
String, ByVal wType As Long ) As Long
```

In der folgenden Tabelle (Tabelle 6.1) werden die Parameter der VBA-Funktion `MsgBox` der API-Funktion `MessageBoxA` gegenübergestellt.

Tabelle 6.1
Parameter der VBA-`MsgBox` und API-`MessageBoxA` im Vergleich

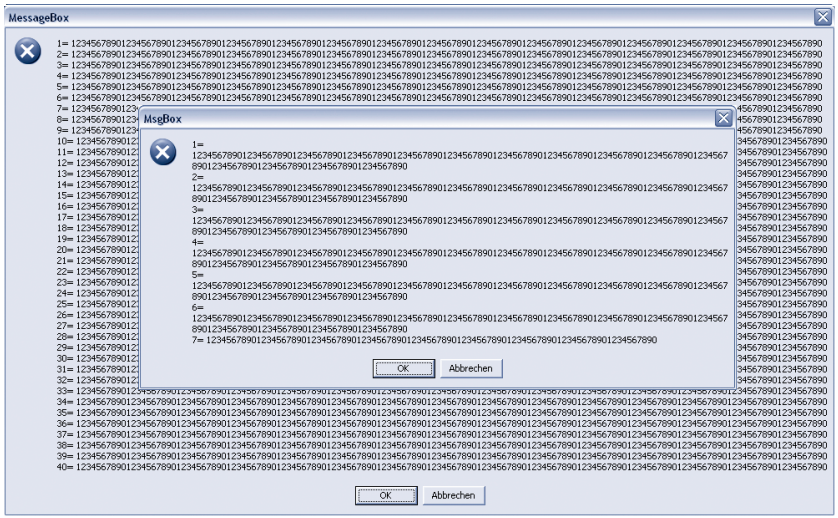
MsgBox	MessageBox	Bedeutung
	`hwnd`	Besitzerfenster
`prompt`	`lpText`	Meldung im Dialogfeld
`buttons`	`wType`	Legt Anzahl und Typ der Schaltflächen, die Symbolart, die Standardschaltfläche und die Bindung des Dialogfeldes fest
`title`	`lpCaption`	Titelleiste des Dialogfeldes
`helpfile`		Hilfedatei
`context`		Hilfekontextkennung

Wie Sie sehen, besteht der Unterschied im Parameter Besitzerfenster und in den Parametern zur Hilfedatei. Sie können also bei der VBA-Funktion kein Fensterhandle und bei der API-Funktion keine Hilfedatei angeben.

Die Werte der Konstanten, welche die Anzahl und Typ der Schaltflächen, die Symbolart, die Standardschaltfläche und die Bindung des Dialogfeldes festlegen, sind identisch. Wenn man sich das Aussehen dieser beiden Messageboxen ansieht, wird man auf den ersten Blick auch keinen Unterschied feststellen.

Bei genauerer Untersuchung (Abbildung 6.2) stellt man jedoch fest, dass die Anzahl der Zeichen in einer Zeile und die Gesamtzahl der Zeichen bei der VBA-Funktion erheblich gegenüber der API-Funktion eingeschränkt sind. Laut Onlinehilfe liegt das Maximum der `MsgBox`-Funktion bei etwa 1024 Zeichen.

Abbildung 6.2
Unterschiede zwischen der API- und VBA-Funktion



Die Messageboxen wurden mit folgendem Code (Listing 6.3) erzeugt:

Listing 6.3
Beispiele\06_Dialoge\
06_01_Dialoge.xls\mdlMsgBox

```
Private Declare Function MessageBox _
   Lib "user32" Alias "MessageBoxA" ( _
   ByVal hwnd As Long, _
   ByVal lpText As String, _
   ByVal lpCaption As String, _
   ByVal wType As Long _
   ) As Long

Private Const MB_ABORTRETRYIGNORE = &H2&
Private Const MB_APPLMODAL = &H0&
Private Const MB_COMPOSITE = &H2
Private Const MB_DEFBUTTON1 = &H0&
Private Const MB_DEFBUTTON2 = &H100&
Private Const MB_DEFBUTTON3 = &H200&
Private Const MB_DEFMASK = &HF00&
Private Const MB_ICONASTERISK = &H40&
Private Const MB_ICONEXCLAMATION = &H30&
Private Const MB_ICONHAND = &H10&
Private Const MB_ICONINFORMATION = MB_ICONASTERISK
Private Const MB_ICONMASK = &HF0&
Private Const MB_ICONQUESTION = &H20&
Private Const MB_ICONSTOP = MB_ICONHAND
Private Const MB_OK = &H0&
Private Const MB_OKCANCEL = &H1&
Private Const MB_PRECOMPOSED = &H1
Private Const MB_RETRYCANCEL = &H5&
Private Const MB_SETFOREGROUND = &H10000
Private Const MB_SYSTEMMODAL = &H1000&
Private Const MB_YESNO = &H4&
Private Const MB_YESNOCANCEL = &H3&

Private Sub test()
   Dim strDummy   As String
   Dim i          As Long
   Dim k          As Long

   For i = 1 To 40  ' 40 Zeilen
      strDummy = strDummy & i & "= "
      For k = 1 To 150 '150 Zeichen in einer Reihe
        strDummy = strDummy & k Mod 10
      Next
      strDummy = strDummy & vbCrLf
   Next

   MessageBox 0&, strDummy, "MessageBox", vbOKCancel Or MB_ICONHAND
   MsgBox strDummy, MB_OKCANCEL Or vbCritical, "MsgBox"

End Sub
```

Wie Sie erkennen können, lassen sich die Konstanten, welche die Icons, Buttons usw. angeben, beliebig zwischen der VBA- und der API-Welt tauschen.

6.4.2 MsgBox Timeout

Leider lassen sich die Messageboxen nicht zeitgesteuert schließen. Die Standard-Messagebox wird *modal* angezeigt. D.h., die Programmausführung stoppt so lange, wie die Messagebox angezeigt wird. Ein Verschieben und Anpassen ist somit ohne weiteres nicht möglich.

Auch die `OnTime`-Methode zum zeitgesteuerten Aufruf einer Prozedur, in der man die Messagebox schließen könnte, funktioniert nicht. Der Benutzer muss also immer auf den Dialog reagieren, um die Programmausführung fortzusetzen, die Messagebox wartet sonst bis ans Ende aller Tage.

Abhilfe bietet der *Windows Script Host* (WSH) (Listing 6.4).

Listing 6.4
Beispiele\06_Dialoge\
06_01_Dialoge.xls\
mdlWSHMsgboxClose

```
Private Sub MsgboxCloseTimeout()
   Dim objWSHShell   As Object
   Dim lngTimeout    As Long
   Dim strMessage    As String

   ' WSH Objekt erzeugen
   Set objWSHShell = CreateObject("wscript.shell")

   lngTimeout = 3 ' Sekunden ?

   ' Meldung anzeigen und Ergebnis auswerten
   Select Case objWSHShell.popup( _
      "Nach " & lngTimeout & " Sekunden schließen", _
      lngTimeout, "wscript.shell.popup", vbYesNoCancel)

      Case vbYes
         strMessage = "Ja"

      Case vbNo
         strMessage = "Nein"

      Case Else
         strMessage = "Abbrechen"

   End Select

   MsgBox "Es wurde " & strMessage & " gewählt"
End Sub
```

Die Verwendung des `WSH` heißt aber auch, sich davon abhängig zu machen. In verschiedenen Firmennetzwerken wird auch von administrativer Seite der WSH nicht geduldet, so dass gerade im geschäftlichen Bereich beim Austausch von Arbeitsmappen Probleme auftreten können. Wenn Sie Ihre Mappen unabhängig von irgendwelchen fremden Installationen benutzen wollen, sollten Sie generell davon Abstand nehmen, etwas anderes als Excel und die API zu benutzen.

Als Alternative könnten Sie sich beispielsweise eine Userform mit ähnlichem Aussehen basteln. Mit der `OnTime`-Methode haben Sie dann die Möglichkeit, die UserForm nach einer bestimmten Zeit programmgesteuert zu schließen.

Eine weitere Möglichkeit besteht darin, mit ein paar API-Funktionen so etwas nachzubauen. Gegenüber dem WSH hat das den großen Vorteil, dass es auch funktioniert, selbst wenn der WSH nicht verfügbar ist. Wenn dagegen die API-Funktionen nicht mehr zur Verfügung stehen, funktioniert wahrscheinlich das ganze System nicht mehr.

Als positiver Nebeneffekt bietet sich dabei die Möglichkeit, die Messagebox ganz an seine eigenen Bedürfnissen anzupassen. Beispielsweise können Sie mit etwas Aufwand die Beschriftungen der Buttons ändern und auch die Position der Box festlegen.

Theoretisch sind den Formatierungsmöglichkeiten wenig Grenzen gesetzt, die Frage, die sich aber immer wieder stellt, ist die, ob sich solch ein Aufwand überhaupt lohnt. Andererseits muss man das Rad ja nicht immer wieder neu erfinden und hat ja die Möglichkeit, seinen Code weiterzuverwenden, sei es mittels Copy-Paste oder ausgelagert in ein AddIn.

Initialisiert man einen API-Timer vor dem Aufruf der Messagebox, wird die Timerfunktion trotz modaler Anzeige aufgerufen. Dort können Sie dann das Messageboxfenster suchen und mit Hilfe von diesem Handle alle möglichen Dinge anstellen, die ein Fenster dieser Klasse zulässt.

Der hier vorgestellte Code (Listing 6.5) verwendet einen solchen API-Timer, der unter anderem einen Funktionszeiger benötigt. Leider steht nur unter Versionen > Excel 97 der Operator `AddressOf` zur Verfügung. Der kann aber unter Excel 97 nachgebaut werden. Es ist zwar etwas mühselig, aber es funktioniert tadellos.

Bemühen Sie dazu einfach mal eine Suchmaschine Ihrer Wahl mit der Suche nach dem Stichwort *TipGetLpfnOfFunctionId*. Mittels bedingter Kompilierung können Sie den Code dann so anpassen, dass er unter allen Excel-Versionen problemlos läuft.

Listing 6.5
Beispiele\06_Dialoge\
06_01_Dialoge.xls\
mdlAPIMsgBoxClose

```
Private Declare Function SetTimer _
   Lib "user32" ( _
   ByVal hwnd As Long, _
   ByVal nIDEvent As Long, _
   ByVal uElapse As Long, _
   ByVal lpTimerFunc As Long _
   ) As Long

Private Declare Function KillTimer _
   Lib "user32" ( _
   ByVal hwnd As Long, _
   ByVal nIDEvent As Long _
   ) As Long

Private Declare Function FindWindow _
   Lib "user32" Alias "FindWindowA" ( _
   ByVal lpClassName As String, _
   ByVal lpWindowName As String _
   ) As Long
```

Listing 6.5 (Forts.)
Beispiele\06_Dialoge\
06_01_Dialoge.xls\
mdlAPIMsgBoxClose

```
Private Declare Function SetWindowPos _
   Lib "user32" ( _
   ByVal hwnd As Long, _
   ByVal hWndInsertAfter As Long, _
   ByVal x As Long, _
   ByVal y As Long, _
   ByVal cx As Long, _
   ByVal cy As Long, _
   ByVal wFlags As Long _
   ) As Long

Private Declare Function SetWindowText _
   Lib "user32" Alias "SetWindowTextA" ( _
   ByVal hwnd As Long, _
   ByVal lpString As String _
   ) As Long

Private Declare Function GetWindow _
   Lib "user32" ( _
   ByVal hwnd As Long, _
   ByVal wCmd As Long _
   ) As Long

Private Declare Function GetWindowText _
   Lib "user32" Alias "GetWindowTextA" ( _
   ByVal hwnd As Long, _
   ByVal lpString As String, _
   ByVal cch As Long _
   ) As Long

Private Declare Function PostMessage _
   Lib "user32" Alias "PostMessageA" ( _
   ByVal hwnd As Long, _
   ByVal wMsg As Long, _
   ByVal wParam As Long, _
   ByVal lParam As Long _
   ) As Long

Private Const SC_CLOSE = &HF060&
Private Const GW_CHILD = 5
Private Const GW_HWNDFIRST = 0
Private Const GW_HWNDNEXT = 2
Private Const WM_LBUTTONDOWN = &H201
Private Const WM_LBUTTONUP = &H202
Private Const WM_SYSCOMMAND = &H112

Private mlngTimer          As Long
Private mlngTimer1         As Long
Private mstrCaption        As String
Private mstrButtonFrom     As String
Private mstrButtonTo       As String
Private mlngX              As Long
Private mlngY              As Long
Private mblnMove           As Boolean
Private mblnRename         As Boolean
Private mstrButtonKlick    As String
```

Listing 6.5 (Forts.)
Beispiele\06_Dialoge\
06_01_Dialoge.xls\
mdlAPIMsgBoxClose

```
Sub testMsgBox()
Dim strMessage As String

Select Case MsgBoxAPI( _
      "Hallo", vbYesNoCancel, "Titelleiste", "VBAXL10.CHM", 0, _
      30, 30, "ja", "Dau", "nein", 5000)
   Case vbYes
      strMessage = "Ja"
   Case vbNo
      strMessage = "Nein"
   Case Else
      strMessage = "Abbrechen"
End Select
MsgBox "Es wurde " & strMessage & " gewählt"
End Sub

Public Function MsgBoxAPI( _
   strMessage As String, _
   Optional lngButtons As Long, _
   Optional strTitle As String, _
   Optional strHelp As String, _
   Optional lngContext As Long, _
   Optional lngXPos As Long, _
   Optional lngYPos As Long, _
   Optional strButtonCaptionFrom As String, _
   Optional strButtonCaptionTo As String, _
   Optional strButtonKlick As String, _
   Optional lngTimeout As Long)

   mblnMove = Not ((lngXPos = 0) And (lngYPos = 0))
   mblnRename = Not ((strButtonCaptionFrom = "") And _
      (strButtonCaptionTo = ""))

   ' Titelleiste
   mstrCaption = strTitle

   ' Position der MsgBox festlegen
   mlngX = lngXPos
   mlngY = lngYPos

   ' Kaufmännische UND Zeichen vor einem Buchstaben stellen
   ' diesen unterstrichen dar, als Kennzeichen für einen Shortcut
   mstrButtonFrom = strButtonCaptionFrom
   mstrButtonTo = strButtonCaptionTo
   mstrButtonKlick = strButtonKlick

   ' Hilfebutton einblenden
   If strHelp <> "" Then lngButtons = lngButtons Or vbMsgBoxHelpButton

   #If VBA6 = 0 Then
      ' Unter Versionen = Excel 97 klappt das so nicht.
      ' Kann man aber nachbauen
      MsgBox "Operator AddressOf in" & _
         " dieser Version nicht verfügbar"
      Exit Function
   #End If
```

Listing 6.5 (Forts.)
Beispiele\06_Dialoge\
06_01_Dialoge.xls\
mdlAPIMsgBoxClose

```
    If mblnMove Or mblnRename Then
        ' Timer zum Verschieben und Umbenennen der Buttons
        mlngTimer = Timerset(5)
    End If

    If lngTimeout > 0 Then
        ' Timer zum Beenden
        mlngTimer1 = Timerset(lngTimeout)
    End If

    ' Messagebox anzeigen
    MsgBoxAPI = MsgBox(strMessage, lngButtons, mstrCaption, _
      strHelp, lngContext)
End Function

Private Sub ApiTimer(ByVal lOwner As Long, _
    ByVal lWindowMessage As Long, _
    ByVal lngTimerID As Long, _
    ByVal lTickCount As Long)

    ' Diese Funktion wird vom Timer aufgerufen. Jeder
    ' Fehler hier oder in einer der Sub-Prozeduren hat
    ' sehr unangenehme Folgen. Deshalb Fehlerbehandlung
    On Error Resume Next

    ' Der Win Timer liefert hier in der Callback-Funktion
    ' unter RückTimerKennung eine Kennung, die auch beim
    ' erzeugen als Handle geliefert wurde. Damit lassen
    ' sich die verschiedenen Timer unterscheiden.

    If lngTimerID = mlngTimer Then
        ' Timerereignis Umbenennen, Verschieben

        KillMyTimer mlngTimer ' Timer löschen
        ' Umbenennen, Verschieben
        MsgBoxReplaceRename
    End If

    If lngTimerID = mlngTimer1 Then
        ' Timerereignis Timeout

        KillMyTimer mlngTimer1 ' Timer löschen
        ' MsgBox schließen
        MsgboxCloseClick
    End If

End Sub

Private Sub KillMyTimer(lngTimer As Long)
    ' Der Timer wird zerstört
    If lngTimer <> 0 Then KillTimer 0, lngTimer
End Sub
```

Listing 6.5 (Forts.)
Beispiele\06_Dialoge\
06_01_Dialoge.xls\
mdlAPIMsgBoxClose

```
Private Function Timerset(lngTime As Long) As Long
    ' lngTime ist die Zeit in Millisekunden.
    ' Timer setzen
   Timerset = SetTimer(0, 0, lngTime, AddressOf ApiTimer)
End Function

Private Sub MsgboxCloseClick()
   Dim lngHwnd          As Long
   Dim lngHwnd1         As Long
   Dim lngRet           As Long
   Dim strCaption       As String
   Dim strFrom          As String

   ' Ein Handle auf die MsgBox wird geliefert
   lngHwnd = FindWindow("#32770", mstrCaption)

   ' Ein Handle auf ein Kindfenster der MsgBox
   lngHwnd1 = GetWindow(lngHwnd, GW_CHILD)

   ' Original Buttontext ohne Kaufmännisches UND
   strFrom = TextWithoutAnd(mstrButtonKlick)

   Do
      'Buffer für Fenstertext
      strCaption = String(255, 0)

      'Fenstertext holen
      lngRet = GetWindowText(lngHwnd1, strCaption, 250)
      strCaption = TextWithoutAnd(Left$(strCaption, lngRet))

      If LCase(strCaption) = LCase(strFrom) Then

         ' Mausklick auf den Button wird simuliert
         PostMessage lngHwnd1, WM_LBUTTONDOWN, 0, 0
         PostMessage lngHwnd1, WM_LBUTTONUP, 0, 0

         Exit Sub

      End If

      'Nächstes Fenster auf gleicher Ebene holen
      lngHwnd1 = GetWindow(lngHwnd1, GW_HWNDNEXT)

   Loop While lngHwnd1 <> 0

   ' Fenster ohne Betätigung eines Buttons schließen
   ' Aber nur, wenn man auch einen Abbrechen-Button
   ' oder ein entsprechendes Systemmenü Schließen hat
   PostMessage lngHwnd, WM_SYSCOMMAND, SC_CLOSE, 0

End Sub

Private Sub MsgBoxReplaceRename()
   Dim lngHwnd          As Long
   Dim lngHwnd1         As Long
   Dim lngRet           As Long
   Dim strCaption       As String
```

Listing 6.5 (Forts.)
Beispiele\06_Dialoge\
06_01_Dialoge.xls\
mdlAPIMsgBoxClose

```
    Dim strFrom          As String

    'Ein Handle auf die MsgBox wird geliefert
    lngHwnd = FindWindow("#32770", mstrCaption)

    If lngHwnd <> 0 Then

       'MsgBox wird verschoben
       SetWindowPos lngHwnd, 0, mlngX, mlngY, 0, 0, 1

    End If

    'Ein Handle auf ein Kindfenster der MsgBox
    lngHwnd1 = GetWindow(lngHwnd, GW_CHILD)

    ' Original Buttontext ohne Kaufmännisches UND
    strFrom = TextWithoutAnd(mstrButtonFrom)

    Do
       'Buffer für Fenstertext
       strCaption = String(255, 0)

       'Fenstertext holen
       lngRet = GetWindowText(lngHwnd1, strCaption, 250)
       strCaption = TextWithoutAnd(Left$(strCaption, lngRet))

       If LCase(strCaption) = LCase(strFrom) Then

          ' Caption des gesuchten Buttons ändern
          SetWindowText lngHwnd1, mstrButtonTo
          mstrButtonFrom = mstrButtonTo
          Exit Sub

       End If

       'Nächstes Fenster auf gleicher Ebene holen
       lngHwnd1 = GetWindow(lngHwnd1, GW_HWNDNEXT)

    Loop While lngHwnd1 <> 0

End Sub

Private Function TextWithoutAnd(ByVal strText As String)
    Dim lngRet As Long

    ' 1. Vorkommen vom Kaufmännischen UND
    lngRet = InStr(1, strText, "&")

    If lngRet Then
       TextWithoutAnd = Left(strText, lngRet - 1) & _
          Mid(strText, lngRet + 1, 10)
    Else
       TextWithoutAnd = strText
    End If
End Function
```

MsgBoxAPI

Diese Funktion dient als Schnittstelle zur Außenwelt. Sie nimmt die verschiedenen Parameter auf, wertet sie aus, startet bis zu zwei API-Timer und liefert das Ergebnis des Dialoges zurück.

Zum Starten der Timer wird die Prozedur `Timerset` aufgerufen, die eine eindeutige Kennung der verschiedenen Timer als Longwert zurückliefert. In den modulweit gültigen Variablen `mlngTimer` und `mlngTimer1` werden diese anschließend gespeichert. Erst nachdem die Timer gestartet wurden, kann die Messagebox aufgerufen werden, da der modale Charakter dieses Dialoges eine weitere Programmausführung verhindert.

Timerset

In dieser Funktion wird der API-Timer gesetzt und als Funktionsergebnis eine eindeutige Kennung des initialisierten Timers zurückgeliefert. Die API-Funktion `SetTimer` benötigt dazu die Zeit in Millisekunden, nach welcher der Timer ausgelöst wird. Weiterhin wird noch der Funktionszeiger auf die Callback-Funktion (Rückruffunktion) gebraucht, die als Timerereignis aufgerufen werden soll. Um einen Zeiger auf die Callback-Funktion zu bekommen, können Sie bei Versionen größer Excel 97 den `AddressOf`-Operator benutzen, unter Excel 97 muss diese Funktionalität nachgebaut werden.

KillMyTimer

Diese Prozedur ist eine Hüllroutine für die API-Funktion `KillTimer`, die initialisierte Timer löscht.

ApiTimer

Der Funktionskopf der Callback-Funktion (Rückruffunktion) selbst steht fest und wenn Sie keine unangenehmen Überraschungen erleben wollen, darf er nicht verändert werden.

Der API-Timer übergibt beim Aufruf der Callback-Funktion verschiedene Parameter, unter anderem auch die Kennung des Timers. Deshalb ist es möglich, für verschiedene Timer die gleiche Callback-Funktion zu verwenden. Je nach Timerereignis wird dann die Prozedur `MsgBoxReplaceRename` oder `MsgboxCloseClick` aufgerufen.

MsgBoxReplaceRename

Zum einen soll die Messagebox nach dem Anzeigen an eine vorbestimmte Position geschoben und eine Buttonbeschriftung soll geändert werden. Dafür ist die Prozedur `MsgBoxReplaceRename` bestimmt.

In dieser wird zuerst die Messagebox über den Klassennamen und dem Fenstertitel gesucht. Das macht die API `FindWindow`, die bei Erfolg ein Handle auf dieses Fenster liefert. Mittels der Funktion `SetWindowPos` wird anschließend die Messagebox an die gewünschte Position geschoben, wenn einer der Parameter `lngXPos` und `lngYPos` beim Aufruf der Funktion `MsgBoxAPI` gesetzt wurde.

Jetzt fehlt noch das Umbenennen eines Buttons. Dazu wird der Button gesucht, der geändert werden soll. Die Beschriftung eines Buttons kann auch ein »kaufmännisches Und« (&) enthalten. Dieses legt das Tastenkürzel für den Button fest (der nachfolgende Buchstabe wird später unterstrichen dargestellt). Es wird mit der Funktion `TextWithoutAnd` entfernt.

Mit diesen Informationen ausgestattet, holen Sie sich mit der Funktion `GetWindow` ein Kindfenster der Messagebox, wobei der zweite Parameter auf `GW_CHILD` gesetzt wird. Die Fenster auf der Ebene unterhalb der eigentlichen Messagebox, die in diesem Fall als Container dient, sind die Buttons. Mit `GetWindow` und dem Parameter auf `GW_HWNDNEXT` können Sie nacheinander alle Fenster dieser Ebene durchlaufen. Bei jedem zurückgelieferten Fenster wird mit `GetWindowText` die Beschriftung der Buttons ausgelesen. Ist das zu ändernde Fenster schließlich gefunden, wird die Beschriftung mit der API `SetWindowText` geändert.

MsgboxCloseClick

Das zweite Timerereignis soll nach einer bestimmten Zeit die Messagebox schließen und dabei einen vorher ausgewählten Button betätigen. Die Funktion, die dazu aufgerufen wird, nennt sich `MsgboxCloseClick`.

Sie macht im Prinzip das Gleiche wie die Funktion `MsgBoxReplaceRename` mit einem kleinen Unterschied. Anstatt den Fenstertext zu ändern, wird ein Mausklick darauf simuliert. Dazu sendet man eine entsprechende Fensternachricht an das entsprechende Fenster, hier also an den Button. Dazu benutze ich `PostMessage` mit dem auf `WM_LBUTTONDOWN` und anschließend `WM_LBUTTONUP` gesetzten Parameter `wMsg`.

Wurde kein entsprechendes Fenster gefunden, wird die Messagebox durch das Senden der Systemnachricht `SC_CLOSE` an die Messagebox geschlossen. Dazu wird `PostMessage` verwendet. Da dies nur eine Aufforderung ist, wird diese Nachricht ignoriert, wenn das Systemmenü SCHLIESSEN nicht vorhanden oder deaktiviert ist. Das kann beispielsweise passieren, wenn Sie beim Starten der Messagebox die Buttons `vbYesNo` gewählt haben, wenn also kein Abbrechen möglich ist.

6.5 Inputbox

Wenig bekannt ist, dass es eine `InputBox`-Funktion und eine `InputBox`-Methode gibt. Die Funktion ist Bestandteil von `VBA.Interaction` und ist in allen Anwendungen, die VBA benutzen, verfügbar. Die Methode ist ein Teil von Excel und besitzt als einzigen Unterschied zu der VBA-Funktion den Parameter `Type`.

Hier die Syntax der Funktion:

```
InputBox(prompt[, title] [, default] [, xpos] [, ypos] [, helpfile,
context])
```

Und hier die Syntax der Methode:

```
Application.InputBox(Prompt, Title, Default, Left, Top, HelpFile,
HelpContextID, Type)
```

In diesem Beispiel (Listing 6.6) werden die verschiedenen Flags benutzt:

Listing 6.6
Beispiele\06_Dialoge\
06_01_Dialoge.xls\mdlInputBox

```
Sub test()
   Dim varRet As Variant

   With Worksheets("TestInputBox")
      .Range("A1:B1") = 3
      .Range("B2") = 99

      'In der lokalen Schreibweise (deutsch)
      .Range("A2").FormulaLocal = "=Summe(A1:B1)"

      'In der englischen Schreibweise
      .Range("A3").Formula = "=Sum(A1:B1)"

      'In der englischen Schreibweise
      .Range("A4").Formula = "=A1"

      'In der englischen Schreibweise
      .Range("A5").Formula = "=1/0"

      ' Ausgabe : '6'
      ' Ergebnis einer Formel
      Application.SendKeys "~"
      MsgBox Application.InputBox("Formel", , "=A3" _
         , , , , 0), , "Formel = 0"

      ' Ausgabe : '3'
      ' Zahlen erlaubt
      Application.SendKeys "~"
      MsgBox Application.InputBox("Zahl", , "=A1", _
         , , , , 1), , "Zahl = 1"
      MsgBox .Range("A3").Formula

      ' Ausgabe : '6'
      Application.SendKeys "~"
      MsgBox Application.InputBox("Text", , "=A5", _
         , , , , 2), , "Text = 2"

      ' Ausgabe : 'Falsch'
      Application.SendKeys "~"
      MsgBox Application.InputBox("Boolean", , "0", _
         , , , , 4), , "Boolean = 4"

      ' Ausgabe : '3'
      Application.SendKeys "~"
      MsgBox Application.InputBox("Zellbezug", , "A4", _
         , , , , 8), , "Zellbezug = 8"

      ' Ausgabe : '$A$3'
      Application.SendKeys "~"
      MsgBox Application.InputBox("Zellbezug", , "A3", _
         , , , , 8).Address, , "Zellbezug = 8"
      ' Ausgabe : 'Fehler 2015'
      Application.SendKeys "~"
      MsgBox CStr(Application.InputBox("Fehlerwert", , "A5", _
         , , , , 16)), , "Fehlerwert = 16"
```

Listing 6.6 (Forts.)
Beispiele\06_Dialoge\
06_01_Dialoge.xls\mdlInputBox

```
        ' Ausgabe : 'A1= 3, B1= 3, A2= 6, B2= 99'
        Application.SendKeys "~"
        varRet = Application.InputBox("Matrix", , "A1:B2", _
           , , , , 64)
        MsgBox "A1= " & varRet(1, 1) & _
           ", B1= " & varRet(1, 2) & _
           ", A2= " & varRet(2, 1) & _
           ", B2= " & varRet(2, 2), , "Matrix = 32"
    End With
End Sub
```

Mit dem Parameter `Type` bestimmen Sie die Rückgabewerte der `InputBox`-Methode. Wird der optionale Parameter nicht übergeben, wird Text zurückgeliefert. Um den Typ vorzugeben, müssen in dem zu übergebenden Wert verschiedene Bits als Flags gesetzt werden. Für jeden Typ ist ein anderes Bit zuständig, deshalb lassen sich auch die einzelnen Typen ohne Probleme miteinander kombinieren.

Das Setzen von Bits erledigen Sie am besten mit Hilfe des binären `ODER`. Will man beispielsweise Bit Nummer null setzen – das entspricht der Wertigkeit eins –, verknüpft man den Wert mit 2^0 (`Wert Or` 2^0), beim Bit Nummer eins mit 2^1 usw. Zum Löschen benutzt man ein binäres UND. Der Ausdruck `Wert And Not` 2^0 löscht beispielsweise Bit Nummer null, lässt aber die anderen Bits in Ruhe.

Ist kein Bit gesetzt, wird das Ergebnis einer Formel zurückgegeben. Ein Klick in die Zelle `A1` trägt in das Eingabefeld `=A1` ein und würde in dem Beispiel `6` zurückliefern.

Bit Nummer null (Wert 1) sorgt dafür, dass Zahlen zurückgeliefert werden. Ein Verweis (beispielsweise `=A1`) auf eine Zelle mit Textinhalt würde einen Fehler auslösen. Die direkte Eingabe einer Zahl oder der Verweis auf eine Zelle mit einer Zahl als Inhalt funktioniert dagegen.

Ist Bit Nummer eins (Wert 2) gesetzt, wird Text zurückgeliefert. Verweise auf Zellen mit Fehlerwerten führen beispielsweise zu Fehlern.

Beim gesetzten Bit zwei (Wert 4) sind boolesche Werte erlaubt.

Ein gesetztes Bit drei (Wert 8) liefert einen Zellbezug. D.h., es wird der Wert dieser angegebenen Zelle zurückgeliefert, wenn man beim Aufruf nicht auf das Objekt zugreift. Das kann man machen, indem man das Ergebnis einer Objektvariablen zuweist oder auf eine Eigenschaft oder Methode des Objektes zugreift.

Folgendes liefert in dem Beispiel (Listing 6.6) den Wert von A4:

```
MsgBox Application.InputBox("Zellbezug", , "A4", , , , , 8)
```

Dieser Code liefert den Zellbezug `$A$4`

```
MsgBox Application.InputBox("Zellbezug", , "A4", , , , , 8).Address
```

Beim Setzen des Bits Nummer vier (Wert 16) sind Fehlerwerte erlaubt. Ein Verweis auf eine Zelle mit dem Fehlerwert `#DIV/0!` liefert den Text »`Fehler 2015`« zurück.

Ist Bit sechs gesetzt, wird eine Matrix zurückgeliefert. Die Eingabe von »`A1:B2`« würde ein zweidimensionales Array liefern.

6.6 Farbauswahl

Excel kann gleichzeitig nur 56 Farben in einer Arbeitsmappe darstellen, die über einen Farbindex angesprochen werden können. Wird eine der 56 Palettenfarben geändert,

```
ActiveWorkbook.Colors(Index) = RGB(Rot, Grün, Blau)
```

werden beispielsweise alle Schriftfarben mit diesem Farbindex in der neuen Farbe dargestellt. Wenn man sich in einem selbstgeschriebenen Programm darauf verlässt, dass ein Farbindex immer die gleiche Farbe darstellt, kann man Probleme bekommen. Das ist zum Beispiel der Fall, wenn das Programm fremde Arbeitsmappen bearbeitet, dessen Farbpalette geändert wurde. Man kann zwar mit

```
ActiveWorkbook.ResetColors
```

die Palettenfarben zurücksetzen, den Besitzer dieser Arbeitsmappe wird das aber wenig erfreuen. Besser ist es, mit RGB-Farben zu arbeiten, es wird dann die ähnlichste Farbe der Palette benutzt.

Es existiert in Excel zwar ein eingebauter Dialog zur Farbauswahl. Leider ist dieser ungeeignet, wenn es darum geht, RGB-Farben als Wert zu bekommen. Zudem nimmt man mit diesem Dialog jedes mal eine direkte Änderung an einer Zelle vor, wenn man einen Farbwert benötigt. Die Möglichkeit, diesen Wert ohne Zelländerung zu bekommen hat man leider nicht.

Das folgende Beispiel (Listing 6.7) zeigt Ihnen, wie Sie den Windows-internen Dialog zur Farbauswahl aus der `comdlg32.dll` aufrufen. Sie haben zusätzlich die Möglichkeit, 16 Farben vorzudefinieren, die im Dialog als benutzerdefiniert angezeigt werden. Wurde eine Farbe ausgewählt, wird diese als Text in der Form `RGB(R,G,B)` in die Zwischenablage kopiert, damit sie von dort aus direkt in ein Codemodul eingefügt werden kann.

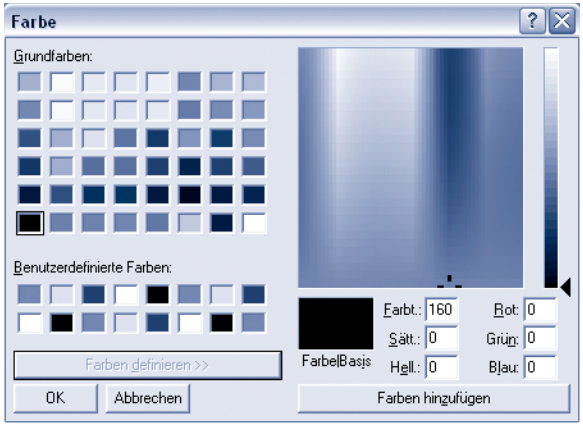


Abbildung 6.3
Windows-interner Dialog zur Farbauswahl

Folgender Code in das Klassenmodul eines Tabellenblattes, das einen Button mit dem Namen `cmdColor` enthält:

Listing 6.7
Beispiele\06_Dialoge\
06_01_Dialoge.xls\mdlColor

```
Private Sub cmdColor_Click()
   Dim lngRGB     As Long
   Dim lngColor   As String
   Dim bytR       As Byte
   Dim bytG       As Byte
   Dim bytB       As Byte

   ' Dialog Aufrufen
   lngRGB = ColorDialog()

   ' Bei einem Fehler abbrechen
   If lngRGB = &HFFFFFFFF Then Exit Sub
   ' Den Longwert in einen Hexstring mit den
   ' letzten 6 Stellen umwandeln
   lngColor = String(6 - Len(Hex(lngRGB)), _
      Asc("0")) & Hex(lngRGB)

   ' Die einzelnen Farbanteile lassen sich
   ' aus einem Hexstring leicht extrahieren
   bytR = CByte("&H" & Right(lngColor, 2))
   bytG = CByte("&H" & Mid(lngColor, 3, 2))
   bytB = CByte("&H" & Left(lngColor, 2))

   ' Das Ergebnis in einer Messagebox ausgeben
   MsgBox "lngColor Lng = " & lngRGB & vbCrLf & _
      "lngColor Hex = " & lngColor & vbCrLf & _
      "Rot = " & Right(lngColor, 2) & vbCrLf & _
      "Grün = " & Mid(lngColor, 3, 2) & vbCrLf & _
      "Blau = " & Left(lngColor, 2) & vbCrLf & _
      "Rot = " & bytR & vbCrLf & _
      "Grün = " & bytG & vbCrLf & _
      "Blau = " & bytB
   ' Zelle einfärben
   Me.Range("a1:b1").Interior.Color = lngRGB
End Sub
```

Den nachfolgenden Code in ein Standardmodul:

```
Private Type CHOOSECOLOR
   lStructSize As Long
   hwndOwner As Long
   hInstance As Long
   rgbResult As Long
   lpCustColors As String
   flags As Long
   lCustData As Long
   lpfnHook As Long
   lpTemplateName As String
End Type

Declare Function CHOOSEMYCOLOR _
   Lib "comdlg32.dll" Alias "ChooseColorA" ( _
   pChoosecolor As CHOOSECOLOR ) As Long

Private Declare Sub CopyMemory _
   Lib "kernel32" Alias "RtlMoveMemory" ( _
   Destination As Any, Source As Any, _
   ByVal Length As Long )
```

Listing 6.7 (Forts.)
Beispiele\06_Dialoge\
06_01_Dialoge.xls\mdlColor

```
Private Declare Function CloseClipboard _
   Lib "user32" () As Long

Private Declare Function OpenClipboard _
   Lib "user32" ( _
   ByVal hWnd As Long _
   ) As Long

Private Declare Function EmptyClipboard _
   Lib "user32" () As Long

Private Declare Function SetClipboardData _
   Lib "user32" ( _
   ByVal wFormat As Long, _
   ByVal hMem As Long _
   ) As Long

Private Declare Function GlobalAlloc _
   Lib "kernel32" ( _
   ByVal wFlags As Long, _
   ByVal dwBytes As Long _
   ) As Long

Private Declare Function GlobalLock _
   Lib "kernel32" ( _
   ByVal hMem As Long _
   ) As Long

Private Declare Function GlobalUnlock _
   Lib "kernel32" ( _
   ByVal hMem As Long _
   ) As Long

Private Const GMEM_ZEROINIT = &H40
Private Const GMEM_MOVEABLE = &H2
Private Const GMEM_SHARE = &H2000
Private Const GMEM_All = ( _
   GMEM_ZEROINIT Or _
   GMEM_SHARE Or _
   GMEM_MOVEABLE)

Private Const CF_TEXT = 1

Public Function ColorDialog() As Long
   Dim udtChoosecolor         As CHOOSECOLOR
   Dim lngRGB                 As Long
   Dim strColor               As String
   Dim abytCustomColors(63)   As Byte
   Dim bytR                   As Byte
   Dim bytG                   As Byte
   Dim bytB                   As Byte

   ' Funktion mit Fehlerwert vorbelegen
   ColorDialog = &HFFFFFFFF

   'Die benutzerdefinierten Farben vorbelegen
   SetDefinedColors abytCustomColor
```

Listing 6.7 (Forts.)
Beispiele\06_Dialoge\
06_01_Dialoge.xls\mdlColor

```
    'Da bei der Übergabe der String in eine lokale Ansi-
    'Kopie umgewandelt wird, muss das Bytearray vorher
    'in einen Unicodestring umgewandelt werden
    udtChoosecolor.lpCustColors = StrConv(abytCustomColors, vbUnicode)
    udtChoosecolor.lStructSize = Len(udtChoosecolor)

    'Den Dialog aufrufen
    If CHOOSEMYCOLOR(udtChoosecolor) <> 0 Then
       ' Farbe wurde gewählt

       ' RGB-Wert ermitteln
       lngRGB = udtChoosecolor.rgbResult

       'In einen Hexstring mit den letzten 6 Stellen umwandeln
       strColor = String(6 - Len(Hex(lngRGB)), _
           Asc("0")) & Hex(lngRGB)

       'Die einzelnen Farbanteile extrahieren
       bytR = CByte("&H" & Right(strColor, 2))
       bytG = CByte("&H" & Mid(strColor, 3, 2))
       bytB = CByte("&H" & Left(strColor, 2))

       ' Als Text in der Form RGB(R,G,B) in das Clipboard bringen
       CopyTextToClip " RGB(" & bytR & _
           " ," & bytG & " ," & bytB & ")"

       ' Den gewählten RGB-Wert zurückgeben
       ColorDialog = lngRGB

    End If
End Function

Private Sub SetDefinedColors(x() As Byte)
    Dim lngRGB As Long
    ' Das übergebene Bytearray wird mit vordefinierten
    ' Farben gefüllt

    ' Benutzerdefinierte Farbe 1
    lngRGB = RGB(255, 0, 0) 'Rot
    CopyMemory x(0), lngRGB, 4

    ' Benutzerdefinierte Farbe 2
    lngRGB = RGB(0, 255, 0) 'Grün
    CopyMemory x(4), lngRGB, 4

    ' Benutzerdefinierte Farbe 3
    lngRGB = RGB(0, 0, 255) 'Blau
    CopyMemory x(8), lngRGB, 4

    ' Benutzerdefinierte Farbe 4
    lngRGB = RGB(255, 255, 255) 'Weiß
    CopyMemory x(12), lngRGB, 4

    ' Benutzerdefinierte Farbe 5
    lngRGB = RGB(0, 0, 0) 'Schwarz
    CopyMemory x(16), lngRGB, 4
```

Listing 6.7 (Forts.)
Beispiele\06_Dialoge\
06_01_Dialoge.xls\mdlColor

```
    ' Benutzerdefinierte Farbe 6
    lngRGB = RGB(255, 0, 0) 'Rot
    CopyMemory x(20), lngRGB, 4

    ' Benutzerdefinierte Farbe 7
    lngRGB = RGB(0, 255, 0) 'Grün
    CopyMemory x(24), lngRGB, 4

    ' Benutzerdefinierte Farbe 8
    lngRGB = RGB(0, 0, 255) 'Blau
    CopyMemory x(28), lngRGB, 4

    ' Benutzerdefinierte Farbe 9
    lngRGB = RGB(255, 255, 255) 'Weiß
    CopyMemory x(32), lngRGB, 4

    ' Benutzerdefinierte Farbe 10
    lngRGB = RGB(0, 0, 0) 'Schwarz
    CopyMemory x(36), lngRGB, 4

    ' Benutzerdefinierte Farbe 11
    lngRGB = RGB(255, 0, 0) 'Rot
    CopyMemory x(40), lngRGB, 4

    ' Benutzerdefinierte Farbe 12
    lngRGB = RGB(0, 255, 0) 'Grün
    CopyMemory x(44), lngRGB, 4

    ' Benutzerdefinierte Farbe 13
    lngRGB = RGB(0, 0, 255) 'Blau
    CopyMemory x(48), lngRGB, 4

    ' Benutzerdefinierte Farbe 14
    lngRGB = RGB(255, 255, 255) 'Weiß
    CopyMemory x(52), lngRGB, 4

    ' Benutzerdefinierte Farbe 15
    lngRGB = RGB(0, 0, 0) 'Schwarz
    CopyMemory x(56), lngRGB, 4

    ' Benutzerdefinierte Farbe 16
    lngRGB = RGB(255, 0, 0) 'Rot
    CopyMemory x(60), lngRGB, 4

End Sub

Public Sub CopyTextToClip(ByVal myText As String)
   Dim lngHMemory    As Long
   Dim lngMemoryText As Long
   Dim lngLength     As Long
   Dim abytText()    As Byte

   ' In ein Ansi-Bytearray umwandeln
   abytText = StrConv(myText & vbNullChar, vbFromUnicode)

   ' Länge des Textes
   lngLength = UBound(abytText) + 1
```

Listing 6.7 (Forts.)
Beispiele\06_Dialoge\
06_01_Dialoge.xls\mdlColor

```
    ' Speicher reservieren
    lngHMemory = GlobalAlloc(GMEM_All, lngLength)

    ' Zeiger holen und sperren. So lange gültig, bis GlobalUnlock
    lngMemoryText = GlobalLock(lngHMemory)

    ' Text in den reservierten Speicher kopieren
    CopyMemory ByVal lngMemoryText, abytText(0), lngLength

    ' Clipboard öffnen
    OpenClipboard 0&

    ' Clipboard leeren
    EmptyClipboard

    ' Clipboard füllen
    SetClipboardData CF_TEXT, ByVal lngMemoryText

    ' Clipboard schließen
    CloseClipboard

    ' Sperrung aufheben
    GlobalUnlock lngHMemory

End Sub
```

cmdColor_Click

In dieser Ereignisprozedur wird die Funktion `ColorDialog` aufgerufen, welche einen Longwert mit der ausgewählten `RGB`-Farbe zurückliefert. Wurde nichts ausgewählt, liefert die Funktion `ColorDialog` den Wert `&HFFFFFFFF` zurück, in diesem Fall wird die weitere Ausführung der Prozedur abgebrochen.

Bei jedem anderen zurückgelieferten Wert wird der Longwert in einen sechsstelligen Hexwert umgewandelt und es werden daraus die einzelnen Farbanteile extrahiert. Das Ergebnis wird anschließend in einer Messagebox ausgeben und die Zellen `A1` und `B1` bekommen diesen Farbwert zugewiesen. Leider kennt Excel nur 56 Farben für Zellen, deshalb wird nur die Farbe dargestellt, die dem RGB-Wert am ähnlichsten ist.

ColorDialog

Diese Funktion öffnet den Farbauswahldialog und gibt die gewählte Farbe als RGB-Wert zurück.

Zuerst wird der Rückgabewert der Funktion auf den Wert `&HFFFFFFFF` gesetzt, der signalisieren soll, dass noch keine Farbe ausgewählt wurde. Die Zahl Null kann dafür nicht benutzt werden, da dieser Wert der Farbe Schwarz entspricht.

Als Nächstes wird ein Bytearray mit 16 Farben vorbelegt, die später im Dialog unter der Rubrik benutzerdefinierte Farben auftauchen sollen. Dazu wird der Prozedur `SetDefinedColors` das Bytearray `abytCustomColors` mit 64 Elementen als Referenz übergeben. Die aufgerufene Prozedur kann somit das Originalarray bearbeiten.

Jeweils 4 Bytes, also ein Longwert dienen zur Darstellung einer Farbe, wobei das höchstwertige Byte jeweils leer bleibt. Die anderen drei enthalten die Farbanteile Rot, Grün und Blau. Dieses Array muss in der Struktur `CHOOSECOLOR` als Element `lpCustColors` übergeben werden.

Da es nicht klappt, ein Array in einer Struktur an eine API-Funktion zu übergeben, muss man einen etwas schmutzigeren Trick anwenden. In der `Type`-Anweisung im Deklarationsbereich des Moduls deklarieren Sie den Typ einfach um und legen fest, dass `lpCustColors` kein Bytearray ist, sondern ein String.

Jetzt kommt aber auch schon das nächste Problem. Bei der Übergabe an eine API-Funktion nimmt VBA an, dass es sich bei dem String um einen Unicodestring handelt und wandelt diesen kurzerhand in ANSI um. Man kann zwar ohne Probleme einem String ein Bytearray zuweisen, aber dann werden daraus keine 64, sondern 32 Unicodezeichen (64 Bytes). Bei der Übergabe als String werden durch die Umwandlung von VBA in ANSI 32 Bytes davon gelöscht.

Also müssen Sie vorher schon jedes einzelne Byte des Arrays in ein Unicodezeichen umwandeln. Das passiert mit der Funktion `StrConv` und dem Parameter `vbUnicode`. Jetzt ist der String 128 Bytes lang, enthält 64 Unicodezeichen und es kann eine Umwandlung nach ANSI ohne Informationsverlust erfolgen.

Nachdem das Element `lStructSize` der Struktur `udtChoosecolor` auf die entsprechende Größe gesetzt wurde, kann der eigentliche Dialog zur Farbauswahl mit der API-Funktion `CHOOSEMYCOLOR` aufgerufen werden.

Die Funktion `CHOOSEMYCOLOR` liefert einen Longwert zurück, der ungleich null ist, wenn eine Farbe ausgewählt wurde. Wird null zurückgeliefert, wird die Funktion verlassen. Wurde eine Farbe gewählt, enthält das Element `rgbResult` der übergebenen Struktur `udtChoosecolor` den ausgewählten Farbwert. Daraus wird ein Hexstring mit den letzten sechs Stellen gemacht, aus dem Sie mittels `Left`, `Mid` und `Right` leicht die einzelnen Farbwerte extrahieren können.

Mit der Funktion `CopyTextToClip` wird ein String in die Zwischenablage kopiert, der die RGB-Funktion mit den Farbwerten in der Form `RGB(100,255,0)` enthält, so dass Sie ihn mit Strg+V direkt in ein Codemodul der VBE einfügen können. Der Rückgabewert dieser Funktion ist ein Long, der den RGB-Wert der ausgewählten Farbe enthält.

SetDefinedColors

Diese Prozedur legt die 16 benutzerdefinierten Farben des Dialogs fest.

Das gesamte übergebene Bytearray enthält 64 Elemente. Jeweils vier Bytes sind für jede einzelne Farbe zuständig und drei davon sollen den RGB-Wert der Farbe aufnehmen.

Ich gehe in diesem Beispiel den Weg über `CopyMemory`, indem ich die vier Bytes des RGB-Longwertes direkt an die Speicheradresse des ersten Bytes jeder Farbe kopiere. Da die Bytes eines Bytearrays im Speicher direkt nebeneinander stehen, spiegelt sich anschließend der Longwert exakt im Bytearray wieder.

Das wird dann für jeden einzelnen Viererblock der 16 Farben durchgeführt. Da das Array als Referenz übergeben wurde, manipuliert man sowieso das Originalarray und braucht diese Prozedur nicht als Funktion auszulegen.

CopyTextToClip

Der Prozedur `CopyTextToClip` wird als Argument ein Text übergeben, welcher ins Clipboard als Text kopiert wird.

Dieser Unicodetext wird mit `StrConv` und dem Argument `vbFromUnicode` in das ANSI Bytearray `abytText` umgewandelt. Am Ende des Bytearrays muss eine Null vorhanden sein, da ja in Wirklichkeit nullterminierte Strings benötigt werden. Deshalb wird diesem String vor der Umwandlung ein `vbNullChar` hinzugefügt.

Daraufhin wird mit `GlobalAlloc` ein Speicherbereich in der Größe des Bytearrays reserviert. Als Ergebnis bekommen Sie ein Handle auf den reservierten Bereich. Dieser Speicherblock wird dann durch `GlobalLock` exklusiv für Sie gesperrt.

Anschließend wird mit der API-Prozedur `CopyMemory` der Inhalt des Bytearrays in diesen Speicher kopiert. Ist das passiert, wird das Clipboard geöffnet und geleert.

Daraufhin kann das Clipboard mit Text gefüllt werden. Dazu wird der Funktion `SetClipboardData` die Konstante `CF_Text` als Argument übergeben. Damit wird dieser Funktion mitgeteilt, dass Text kopiert werden soll, und zwar der Text, der ab dem Parameter `lngMemoryText` im Speicher steht.

Die Sperrung des Speicherblocks wird anschließend durch `GlobalUnlock` aufgehoben und der Speicher freigegeben. Zuletzt wird noch das Clipboard geschlossen.

6.7 Schriftartdialog

Es existiert in Excel ein eingebauter Dialog zur Änderung der Schriftart (Abbildung 6.4).

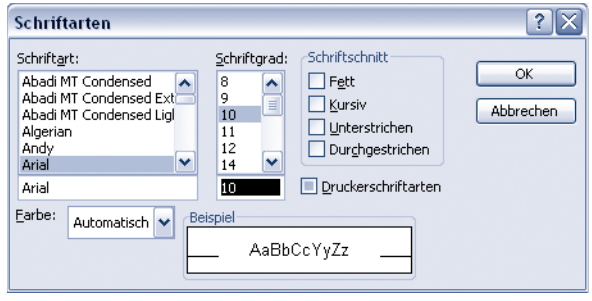


Abbildung 6.4
Dialog Schriftart Excel

Leider ist dieser ungeeignet, wenn es darum geht, an die Informationen selbst zu kommen.

Das folgende Beispiel (Listing 6.8) zeigt Ihnen, wie Sie den Windows-internen Dialog zur Schriftartauswahl (Abbildung 6.5) aus der `comdlg32.dll` aufrufen.

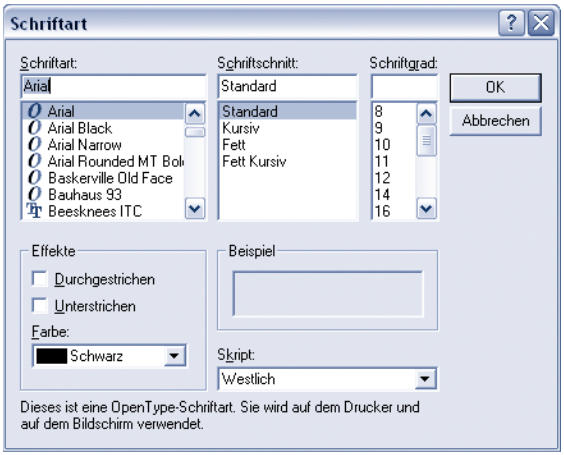


Abbildung 6.5
Dialog Schriftart API

Listing 6.8
Beispiele\06_Dialoge\
06_01_Dialoge.xls\mdlFont

```
Private Declare Function CHOOSEFONT _
   Lib "comdlg32.dll" Alias "ChooseFontA" ( _
   pChoosefont As CHOOSEFONT _
   ) As Long

Private Declare Sub CopyMemory _
   Lib "kernel32" Alias "RtlMoveMemory" ( _
   hpvDest As Any, _
   hpvSource As Any, _
   ByVal cbCopy As Long _
   )

Private Declare Function GlobalLock _
   Lib "kernel32" ( _
   ByVal hMem As Long _
   ) As Long

Private Declare Function GlobalUnlock _
   Lib "kernel32" ( _
   ByVal hMem As Long _
   ) As Long

Private Declare Function GlobalAlloc _
   Lib "kernel32" ( _
   ByVal wFlags As Long, _
   ByVal dwBytes As Long _
   ) As Long

Private Declare Function GlobalFree _
   Lib "kernel32" ( _
   ByVal hMem As Long _
   ) As Long
```

Listing 6.8 (Forts.)
Beispiele\06_Dialoge\
06_01_Dialoge.xls\mdlFont

```
' Font Weight
Private Const FW_NORMAL = 400
Private Const FW_BOLD = 700
Private Const FW_BLACK = 900
Private Const FW_DEMIBOLD = 600
Private Const FW_EXTRABOLD = 800
Private Const FW_EXTRALIGHT = 200
Private Const FW_HEAVY = 900
Private Const FW_LIGHT = 300
Private Const FW_MEDIUM = 500
Private Const FW_REGULAR = 400
Private Const FW_SEMIBOLD = 600
Private Const FW_THIN = 100
Private Const FW_ULTRABOLD = 800
Private Const FW_ULTRALIGHT = 200

Private Const CF_FORCEFONTEXIST = &H10000
Private Const CF_INITTOLOGFONTSTRUCT = &H40&
Private Const CF_LIMITSIZE = &H2000&
Private Const CF_PRINTERFONTS = &H2
Private Const CF_SCREENFONTS = &H1
Private Const CF_EFFECTS = &H100&

Private Const GMEM_MOVEABLE = &H2
Private Const GMEM_ZEROINIT = &H40

Private Type CHOOSEFONT
        lStructSize As Long
        hwndOwner As Long
        hDC As Long
        lpLogFont As Long
        iPointSize As Long
        flags As Long
        rgbColors As Long
        lCustData As Long
        lpfnHook As Long
        lpTemplateName As String
        hInstance As Long
        lpszStyle As String
        nFontType As Integer
        MISSING_ALIGNMENT As Integer
        nSizeMin As Long
        nSizeMax As Long
End Type

Private Type LOGFONT
        lfHeight As Long
        lfWidth As Long
        lfEscapement As Long
        lfOrientation As Long
        lfWeight As Long
        lfItalic As Byte
        lfUnderline As Byte
        lfStrikeOut As Byte
        lfCharSet As Byte
        lfOutPrecision As Byte
        lfClipPrecision As Byte
```

Listing 6.8 (Forts.)
Beispiele\06_Dialoge\
06_01_Dialoge.xls\mdlFont

```
        lfQuality As Byte
        lfPitchAndFamily As Byte
        lfFaceName As String * 31
End Type

Private Sub testFont()
   Dim varFont As Variant

   Set varFont = GetFont()

   MsgBox "Fontname = " & varFont("FontName") _
      & vbCrLf & "Schriftgröße= " & varFont("PointSize") _
      & vbCrLf & "Schriftstärke Normal= " & varFont("Normal") _
      & vbCrLf & "Schriftstärke Fett= " & varFont("Bold") _
      & vbCrLf & "ITALIC= " & varFont("Italic") _
      & vbCrLf & "RGB-Farbe= " & varFont("RgbColors") _
      & vbCrLf & "Unterstrichen= " & varFont("Underline") _
      & vbCrLf & "Durchgestrichen= " & varFont("StrikeOut")

End Sub

Public Function GetFont() As Variant
   Dim udtChoosefont As CHOOSEFONT
   Dim udtFont       As LOGFONT
   Dim lngMemory     As Long
   Dim lngPtrMemory  As Long
   Dim Fontname      As String

   Dim colFont As New Collection

   With udtFont

      ' Angezeigte Schriftart
      .lfFaceName = "Arial" & Chr(0)

      ' Globalen Speicher bereitstellen
      lngMemory = GlobalAlloc(GMEM_MOVEABLE Or _
         GMEM_ZEROINIT, Len(udtFont))

      ' Sperren und Zeiger holen
      lngPtrMemory = GlobalLock(lngMemory)

      ' Die Struktur LOGFONT dahinschieben
      CopyMemory ByVal lngPtrMemory, udtFont, Len(udtFont)

      ' Die Struktur CHOOSEFONT initialisieren
      ' Länge von CHOOSEFONT
      udtChoosefont.lStructSize = Len(udtChoosefont)

      ' Der Zeiger auf die LOGFONT-Struktur
      ' im reservierten Speicher
      udtChoosefont.lpLogFont = lngPtrMemory

      ' Verschiedene Flags für Voreinstellungen
      udtChoosefont.flags = CF_INITTOLOGFONTSTRUCT Or _
         CF_PRINTERFONTS Or CF_SCREENFONTS Or CF_EFFECTS _
         Or CF_FORCEFONTEXIST Or CF_LIMITSIZE
```

Listing 6.8 (Forts.)
Beispiele\06_Dialoge\
06_01_Dialoge.xls\mdlFont

```
        ' Kleinste Schriftgröße in Punkt
        udtChoosefont.nSizeMin = 6

        ' Größte Schriftgröße in Punkt
        udtChoosefont.nSizeMax = 72

        ' udtChoosefont an CHOOSEFONT übergeben
        ' So Bill will, wird der reservierte Speicher
        ' gefüllt
        If CHOOSEFONT(udtChoosefont) <> 0 Then

            ' CHOOSEFONT war erfolgreich
            ' Das Ergebnis in die Struktur udtFont zurückschreiben
            CopyMemory udtFont, ByVal lngPtrMemory, Len(udtFont)

            ' und auswerten
            Fontname = Left(.lfFaceName, InStr(.lfFaceName, _
               Chr(0)) - 1)

            ' Ergebnis in Collection eintragen
            colFont.Add Fontname, "FontName"

            colFont.Add CBool(.lfWeight = FW_NORMAL), "Normal"

            colFont.Add CBool(.lfWeight = FW_BOLD), "Bold"

            colFont.Add udtChoosefont.iPointSize \ 10, "PointSize"

            colFont.Add udtChoosefont.rgbColors, "RgbColors"

            colFont.Add CBool(.lfItalic), "Italic"

            colFont.Add CBool(.lfUnderline), "Underline"

            colFont.Add CBool(.lfStrikeOut), "StrikeOut"

            Set GetFont = colFont

        End If

        ' Reservierten Speicher freigeben
        GlobalUnlock lngMemory
        GlobalFree lngMemory
    End With
End Function
```

Zum Starten des Dialoges wird die API-Funktion `ChooseFontA` aus der Bibliothek `comdlg32.dll` benutzt, der ich alsden Namen `ChooseMyFont` gegeben habe. Ich habe diesen `Alias` benutzt, um besser zwischen Funktion und der Struktur mit Namen `CHOOSEFONT` zu unterscheiden, obwohl es auch bei Namensgleichheit keine Probleme geben dürfte. In dieser *.DLL* finden Sie übrigens neben dieser Funktion auch noch einige andere Dialoge wie zum Beispiel `ChooseColorA` zur Auswahl einer Farbe.

An die Funktion `ChooseMyFont` wird eine zum Teil ausgefüllte Struktur vom Typ `CHOOSEFONT` als Parameter übergeben. Und damit beginnen auch einige Prob-

leme, die man umschiffen muss. Das Element lpLogFont dieser Struktur ist ein Longwert, der die Speicheradresse einer LOGFONT-Struktur enthalten soll. Es gibt ein paar Möglichkeiten an solch einen Wert zu kommen.

Ich reserviere und sperre dazu mit den Funktionen GlobalAlloc und GlobalLock einen Speicherbereich. Die Funktion GlobalLock liefert mir dabei einen Wert, der auf den reservierten Speicherbereich verweist und den ich dann dem Element lpLogFont zuweise. Mit der API CopyMemory kopiere ich an diese Stelle die LOGFONT-Struktur, die schon den Namen einer Schriftart enthält, die im Dialog als Voreinstellung erscheinen soll.

Das Element lStructSize nimmt die Länge der Struktur CHOOSEFONT auf, nSizeMin und nSizeMax die Grenzen der auswählbaren Schriftgrößen in Punkt. Das Element flags ist für verschiedene Voreinstellungen gedacht, wobei die einzelnen Flags (gesetzte und nicht gesetzte Bits) auch kombiniert werden können.

Nach der Übergabe der Struktur CHOOSEFONT an ChooseMyFont enthält das Element lfFaceName die ausgewählte Schriftart. Die anderen gewählten Einstellungen werden in den Speicherbereich geschrieben, der für die Struktur LOGFONT reserviert wurde. Diesen Speicherinhalt müssen Sie nur noch mit CopyMemory in die eigentliche Struktur zurückkopieren, um an die Werte zu kommen.

Diese Werte werden in einer Collection gespeichert und die Collection als Funktionsergebnis zurückgegeben. Die Schlüsselnamen, unter denen Sie anschließend auf die einzelnen Elemente der Collection zugreifen können, sind FontName, PointSize, Normal, Bold, Italic, RgbColors, Underline und StrikeOut.

6.8 Dateiauswahl

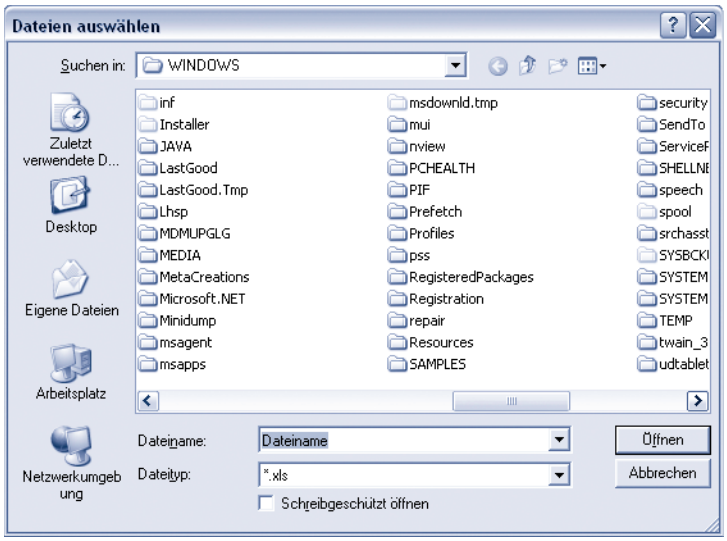
Excel bietet neben vielen anderen Dialogen auch einen zum Öffnen von Arbeitsmappen an. Dieser BuiltInDialog öffnet aber sofort die ausgewählte Mappe und gibt nicht etwa nur den Dateinamen zurück. Benötigen Sie aber nur den Dateinamen, kann Ihnen die GetSaveAsFileName und GetOpenFileName-Methode des Application-Objekts weiterhelfen.

```
MsgBox Application.GetOpenFileName("Text Files (*.txt), *.txt", , _
"Datei öffnen")
```

Diese Dialoge kapseln die Windows-internen Dialoge, bieten aber nicht alle Möglichkeiten, die Ihnen die API-Lösungen bieten. Es ist beispielsweise nicht möglich, dem Excel-Dialog einen Anfangspfad mitzugeben, mit dem der Dialog beginnen soll. Mit ChDir kann man zwar das aktuelle Verzeichnis oder den aktuellen Ordner anpassen, dieser gilt aber nicht nur für diesen einen Dialog.

Um die API-Funktion einzusetzen, muss man einmalig etwas mehr Zeit beim Eintippen aufwenden, und das auch nur, wenn man den Quelltext nicht in elektronischer Form vorliegen hat. Als Gegenleistung bekommen Sie den gewohnten Windowsdialog (Abbildung 6.6) und können ohne Probleme einen Anfangspfad vorgeben.

Abbildung 6.6
Dateiauswahl API



Folgendermaßen wird der Dialog benutzt:

```
Sub test()
   'So aufrufen,
   MsgBox BrowseFile( _
      strFilterName:="Exceldateien *.xls", _
      strFilterExt:="*.xls", _
      strInitPath:="c:\windows", _
      strTitle:="Dateien auswählen", _
      strInitFile:="Dateiname")

   'oder so
   MsgBox BrowseFile("Exceldateien *.xls", "*.xls", _
      "c:\windows", "Dateien auswählen", "Dateiname")
End Sub
```

Ab hier folgt der eigentliche Dialogaufruf:

Listing 6.9
Beispiele\06_Dialoge\
06_01_Dialoge.xls\mdlBrowseFile

```
Private Declare Function GetOpenFileName _
   Lib "comdlg32.dll" Alias "GetOpenFileNameA" ( _
   pOpenfilename As OPENFILENAME _
   ) As Long

Private Type OPENFILENAME
   lStructSize As Long
   hwndOwner As Long
   hInstance As Long
   lpstrFilter As String
   lpstrCustomFilter As String
   nMaxCustFilter As Long
   nFilterIndex As Long
   lpstrFile As String
   nMaxFile As Long
   lpstrFileTitle As String
   nMaxFileTitle As Long
   lpstrInitialDir As String
```

Listing 6.9 (Forts.)
Beispiele\06_Dialoge\
06_01_Dialoge.xls\mdlBrowseFile

```
    lpstrTitle As String
    flags As Long
    nFileOffset As Integer
    nFileExtension As Integer
    lpstrDefExt As String
    lCustData As Long
    lpfnHook As Long
    lpTemplateName As String
End Type

Private Const OFN_FILEMUSTEXIST = &H1000

Public Function BrowseFile( _
   Optional strFilterName As String = "Alle Dateien ( *.* )", _
   Optional strFilterExt As String = "*.*", _
   Optional strInitPath As String, _
   Optional strTitle As String = "Datei öffnen", _
   Optional strInitFile As String)

   Dim OFStrukt As OPENFILENAME

   With OFStrukt
      .lStructSize = Len(OFStrukt)
      .lpstrFilter = strFilterExt & Chr$(0) & _
         strFilterExt & Chr$(0)
      .lpstrFile = strInitFile & Chr$(0) & Space$(256)
      .nMaxFile = 256
      .lpstrFileTitle = Space$(256)
      .nMaxFileTitle = 256
      .lpstrInitialDir = strInitPath
      .lpstrTitle = strTitle
      .flags = OFN_FILEMUSTEXIST

      If GetOpenFileName(OFStrukt) Then
         BrowseFile = Trim(.lpstrFile)
      End If

   End With
End Function
```

Die Funktion ist eigentlich schnell erklärt. Die Struktur `OFStrukt` wird ausgefüllt und das Element `lpstrFile` gibt nach der Rückkehr der API-Funktion `GetOpenFileName` den Dateinamen inklusive Pfad zurück. Das Element `lpstrFilter` gibt den Dateifilter vor, `lpstrFileTitle` den Dialogtitel, `lpstrInitialDir` den Anfangspfad.

Mit dem Element `flags` können Sie das Aussehen und Verhalten mitbestimmen. Möglich sind folgende Konstanten, auch in Kombination:

- `Private Const OFN_ALLOWMULTISELECT = &H200`

  Die Auswahl von mehreren Dateien ist möglich.

- `Private Const OFN_CREATEPROMPT = &H2000`

  Wenn die Datei nicht existiert, erscheint ein Dialog.

- `Private Const OFN_ENABLEHOOK = &H20`

  Der Parameter `lpfnHook` wird ausgewertet.

- `Private Const OFN_ENABLETEMPLATE = &H40`
  `Private Const OFN_ENABLETEMPLATEHANDLE = &H80`

  Die Dialogfeldvorlage wird aktiviert, was immer das auch sein mag.
- `Private Const OFN_EXPLORER = &H80000`

  Nutzt Explorer-Dialoge. Dies ist die Defaulteinstellung.
- `Private Const OFN_EXTENSIONDIFFERENT = &H400`

  Es kann auch ein Dateiname mit einer anderen Erweiterung eingegeben werden, als unter `lpstrDefExt` angegeben.
- `Private Const OFN_FILEMUSTEXIST = &H1000`

  Zusammen mit `OFN_PATHMUSTEXIST` wird dafür gesorgt, dass nur existierende Dateien eingegeben werden können.
- `Private Const OFN_HIDEREADONLY = &H4`

  Das Kontrollkästchen NUR LESEN wird ausgeblendet.
- `Private Const OFN_LONGNAMES = &H200000`

  In älteren Dialogen werden lange Dateinamen unterstützt.
- `Private Const OFN_NOCHANGEDIR = &H8`

  Nach dem Ende des Dialogs wird das ursprüngliche Verzeichnis wiederhergestellt.
- `Private Const OFN_NODEREFERENCELINKS = &H100000`

  Bei einer Verknüpfung wird die Verknüpfungsdatei zurückgeliefert, nicht das Ziel der Verknüpfung.
- `Private Const OFN_NOLONGNAMES = &H40000`

  In älteren Dialogen werden lange Dateinamen nicht unterstützt.
- `Private Const OFN_NONETWORKBUTTON = &H20000`

  Die Schaltfläche NETZWERK wird ausgeblendet.
- `Private Const OFN_NOREADONLYRETURN = &H8000`

  Stellt sicher, dass nur Dateien zurückgeliefert werden, die nicht schreibgeschützt sind und sich nicht in einem schreibgeschützten Verzeichnis befinden.
- `Private Const OFN_NOVALIDATE = &H100`

  Es wird nicht überprüft, ob sich im Pfad unerlaubte Zeichen befinden.
- `Private Const OFN_PATHMUSTEXIST = &H800`

  Zusammen mit `OFN_FILEMUSTEXIST` wird dafür gesorgt, dass nur existierende Dateien eingegeben werden können.
- `Private Const OFN_READONLY = &H1`

  Die Checkbox SCHREIBGESCHÜTZT ÖFFNEN ist beim Anzeigen des Dialogs gesetzt.

Die Funktion `GetOpenFileName` kann ohne große Änderungen durch die API-Funktion `GetSaveAsFileName` ersetzt werden.

Die Deklarationsanweisung dazu lautet wie folgt:

```
Private Declare Function GetSaveFileName Lib "comdlg32.dll" Alias
"GetSaveFileNameA" (pOpenfilename As OPENFILENAME) As Long
```

6.9 Verzeichnisauswahl

Excel bietet keinen Dialog zur Auswahl eines Verzeichnisses. Windows stellt aber einen solchen Dialog zur Verfügung, der über die API-Funktion `SHBrowseForFolder` benutzbar gemacht wird.

Mit diesem Beispiel (Listing 6.10) kann man den Dialog zur Auswahl von Verzeichnissen (Abbildung 6.7) nutzen. Dabei ist es möglich, mittels einer Callback-Funktion den Anfangspfad zur Suche zu setzen.

Weiterhin können Sie beispielsweise festlegen, dass auch ein kompletter Pfad inklusive einer Datei zurückgeliefert werden kann. Sogar eine Editbox kann zusätzlich angezeigt werden und ab Windows 2000 ist es auch möglich, aus dem Dialog heraus ein Verzeichnis anzulegen.

Abbildung 6.7
Dialog Browse For Folder mit den Parametern von VBGetFolder

Zum Testen des Dialoges:

```
Private Sub TestBrowseFolder()
   Dim blnNewFolder As Boolean
   Dim blnEditbox As Boolean
   Dim blnWithFiles As Boolean
   Dim blnOnlyFolder As Boolean
   Dim blnDescription As Boolean
   Dim strStartdir As String
   Dim strCaption As String

   blnNewFolder = True
   blnWithFiles = True
   blnEditbox = True
   blnOnlyFolder = False
   blnDescription = True
   strStartdir = "c:\Dokumente und Einstellungen"
   strCaption = "Dialog Verzeichnisauswahl"
```

```
   MsgBox VBGetFolder( _
      strCaption, _
      strStartdir, _
      blnNewFolder, _
      blnWithFiles, _
      blnEditbox, _
      blnOnlyFolder, _
      blnDescription)
End Sub
```

In ein Modul:

Listing 6.10
Beispiele\06_Dialoge\
06_01_Dialoge.xls\
mdlBrowseFolder

```
Private Declare Function SHBrowseForFolder _
   Lib "shell32" ( _
   lpbi As BROWSEINFO _
   ) As Long

Private Declare Function SHGetPathFromIDList _
    Lib "shell32" ( _
    ByVal pidList As Long, _
    ByVal lpBuffer As String _
    ) As Long

Private Type BROWSEINFO
   hwndOwner      As Long
   pidlRoot       As Long
   pszDisplayName As Long
   lpszTitle      As Long
   ulngFlags      As Long
   lpfnCallback   As Long
   lParam         As Long
   iImage         As Long
End Type

Private Const BIF_BROWSEINCLUDEFILES = &H4000
Private Const BIF_EDITBOX = &H10
Private Const BIF_RETURNONLYFSDIRS = &H1
Private Const BIF_STATUSTEXT = &H4
Private Const BIF_USENEWUI = &H40

Private Const WM_SETTEXT = &HC
Private Const WM_USER = &H400
Private Const BFFM_INITIALIZED = 1
Private Const BFFM_SETSELECTION = (WM_USER + 102)
Private Const BFFM_SELCHANGED = 2
Private Const BFFM_SETSTATUSTEXT = (WM_USER + 100)


Private Declare Function SendMessageString _
   Lib "user32" Alias "SendMessageA" ( _
   ByVal hwnd As Long, _
   ByVal wMsg As Long, _
   ByVal wParam As Long, _
   ByVal lParam As String _
   ) As Long

Private mstrStartDir As String
```

Listing 6.10 (Forts.)
Beispiele\06_Dialoge\
06_01_Dialoge.xls\
mdlBrowseFolder

```
Private Function BrowseCallback( _
   ByVal hwnd As Long, _
   ByVal uMsg As Long, _
   ByVal lp As Long, _
   ByVal pData As Long _
   ) As Long

   Dim strBuffer As String

   On Error Resume Next
   ' Diese Funktion wird vom Dialog aufgerufen
   ' und darf nicht angehalten werden. Auch nicht
   ' durch einen nicht behandelten Fehler.

   If uMsg = BFFM_INITIALIZED Then
   ' Wenn Dialog initialisiert wird

      If Len(mstrStartDir) > 1 Then

         'Jetzt wird das Startverzeichnis gesetzt
         SendMessageString hwnd, BFFM_SETSELECTION, _
          1, mstrStartDir

      End If

   End If

   If uMsg = BFFM_SELCHANGED Then
   ' Selektion wird geändert

      ' Aktueller Pfad wird ausgelesen
      strBuffer = String(512, 0)
      SHGetPathFromIDList lp, strBuffer
      strBuffer = Left(strBuffer, InStr(strBuffer, Chr(0)) - 1)

      ' Jetzt wird die Erklärung gesetzt, aber nicht
      ' unter allen Kombinationen angezeigt
      SendMessageString hwnd, BFFM_SETSTATUSTEXT, 0, strBuffer

   End If
End Function

Public Function VBGetFolder( _
   strCaption As String, _
   strStartdir As String, _
   Optional blnNewFolder As Boolean, _
   Optional blnWithFiles As Boolean, _
   Optional blnEditbox As Boolean, _
   Optional blnOnlyFolder As Boolean, _
   Optional blnDescription As Boolean _
   ) As String

   Dim lngListID     As Long
   Dim strBuffer     As String
   Dim abytTitel()   As Byte
   Dim lngFlags      As Long
   Dim udtBrowseInfo As BROWSEINFO
```

Listing 6.10 (Forts.)
Beispiele\06_Dialoge\06_01_Dialoge.xls\mdlBrowseFolder

```
    ' Modulweite Variable mit dem Anfangspfad füllen
    mstrStartDir = strStartdir & vbNullChar

    ' Den Titel als Bytearray mit einem NullChar am Ende
    abytTitel = StrConv(strCaption & vbNullChar, vbFromUnicode)

    ' Je nach übergebenen Parametern Flags setzen
    If blnNewFolder Then _
       lngFlags = lngFlags Or BIF_USENEWUI
    If blnWithFiles Then _
       lngFlags = lngFlags Or BIF_BROWSEINCLUDEFILES
    If blnEditbox Then _
       lngFlags = lngFlags Or BIF_EDITBOX
    If blnOnlyFolder Then _
       lngFlags = lngFlags Or BIF_RETURNONLYFSDIRS
    If blnDescription Then _
       lngFlags = lngFlags Or BIF_STATUSTEXT

    ' Die Struktur BrowseInfo ausfüllen
    With udtBrowseInfo
       .hwndOwner = 0
       .lpszTitle = VarPtr(abytTitel(0))
       .ulngFlags = lngFlags
       #If VBA6 Then
          ' Callback-Funktion initialisieren zum Setzen des
          ' Anfangspfades und zum Anzeigen des Pfades
          ' als Statustext beim Verzeichniswechsel
          .lpfnCallback = AddressOf_ToLong(AddressOf BrowseCallback)
       #Else
          MsgBox "Operator AddressOf in" & _
             " dieser Version nicht verfügbar"
       #End If
    End With

    ' Dialog aufrufen
    lngListID = SHBrowseForFolder(udtBrowseInfo)

    If lngListID Then
       'Den Pfad aus der ID-List extrahieren
       strBuffer = String(512, 0)
       SHGetPathFromIDList lngListID, strBuffer
       strBuffer = Left(strBuffer, InStr(strBuffer, Chr(0)) - 1)

       ' und als Funktionsergebnis zurückgeben
       VBGetFolder = strBuffer
    End If
End Function

Private Function AddressOf_ToLong(ByVal lFPointer As Long) As Long
    ' Wenn AddressOf benutzt wird, ist diese auf
    ' den ersten Blick unnötige Funktion wichtig
    AddressOf_ToLong = lFPointer
End Function
```

BrowseCallback

Diese Prozedur wird als Rückrufprozedur aufgerufen, wenn der Dialog initialisiert wird oder andere Ereignisse wie die Auswahl eines Verzeichnisses (`BFFM_SELCHANGED`) ausgelöst werden. Fehler oder Haltepunkte in dieser Funktion lassen die Anwendung abstürzen, also sollte auch immer eine Fehlerbehandlung eingesetzt werden.

Als Parameter `hwnd` wird das Fensterhandle und als `uMsg` ein Longwert mitgegeben, der die Art der Nachricht kennzeichnet. Die Parameter `lp` und `pData` sind Longwerte, die abhängig von der Art der Nachricht sind.

Beim Initialisieren des Dialoges wird als Parameter `uMsg` die Nachricht `BFFM_INITIALIZED` übergeben. Dieser Parameter wird ausgewertet und man sendet bei Übereinstimmung mit der API-Funktion `SendMessageString` die Nachricht `BFFM_SETSELECTION` an das Dialogfenster und übergibt mit der Variablen `mstrStartDir` das Startverzeichnis.

Wird im Dialog ein Verzeichnis ausgewählt, kommt an der Callback-Funktion die Message `BFFM_SELCHANGED` an. Das nun aktuelle Verzeichnis holen Sie sich anschließend mit der Funktion `SHGetPathFromIDList`.

Dazu wird der an die Callback-Funktion übergebene Parameter `lp` an die API-Funktion `SHGetPathFromIDList` als Parameter `ID` weitergereicht. Der zweite Parameter dieser Funktion ist der Puffer `strBuffer` der in so einer Größe angelegt wurde, dass er den kompletten Pfad aufnehmen kann.

Wenn Sie den Pfad aus dem Puffer extrahiert haben, senden Sie die Message `BFFM_SETSTATUSTEXT` an das Dialogfenster und übergeben mit der Variable `strBuffer` das aktuelle Verzeichnis als String.

VBGetFolder

Diese Funktion wird aufgerufen, um durch einen Dialog ein Verzeichnis auszuwählen. Als Parameter werden der Titel, der Anfangspfad und optional einige boolesche Werte übergeben, die das Aussehen und die Funktionalität des Dialogs mitbestimmen.

Die API `SHBrowseForFolder` startet den Dialog. Dieser Funktion muss als Parameter eine ausgefüllte Struktur vom Typ `BROWSEINFO` mitgegeben werden. Das Element `ulFlags` enthält dabei die Flags, die durch die übergebenen Parameter `blnNewFolder`, `blnWithFiles`, `blnEditbox`, `blnOnlyFolder` und `blnDescription` gesetzt werden und die Erscheinungsform des Dialogs bestimmen.

- `blnNewFolder`

  Ist diese Variable Wahr, wird das Flag `BIF_USENEWUI` gesetzt und es erscheint ab Windows 2000 ein Button auf dem Dialog zum Anlegen eines Verzeichnisses. Leider wird unter XP dadurch nicht mehr der Statustext angezeigt.

- `blnWithFiles`

  Ist diese Variable Wahr, wird das Flag `BIF_BROWSEINCLUDEFILES` gesetzt und es werden im Dialog Dateien angezeigt, die ausgewählt werden können und die dann auch zusammen mit dem Pfad zurückgeliefert werden.

- `blnEditbox`

  Ist diese Variable Wahr, wird das Flag `BIF_EDITBOX` gesetzt und es wird im Dialog eine Editbox angezeigt.

- `blnOnlyFolder`

  Ist diese Variable Wahr, wird das Flag `BIF_RETURNONLYFSDIRS` gesetzt und der Button OK wird ausgegraut dargestellt, wenn kein Verzeichnis gewählt wurde.

- `blnDescription`

  Ist diese Variable Wahr, wird das Flag `BIF_STATUSTEXT` gesetzt und es wird in diesem Beispiel bei einem Verzeichniswechsel das ausgewählte Verzeichnis in einem zur Verfügung gestellten Statusbereich angezeigt.

Wird eine `ListID` zurückgeliefert, die anzeigt, ob etwas ausgewählt wurde, wird die Funktion `SHGetPathFromIDList` benutzt, um daraus einen String zu extrahieren. Als ein Parameter wird dabei die `ListID` übergeben, der zweite Parameter ist der Puffer `strBuffer` in der entsprechenden Größe, um den Pfad aufnehmen zu können.

AddressOf_ToLong

Diese Funktion wird bei Excelversionen größer Excel 97 benutzt, bei denen der Operator `AddressOf` existiert. Ihr wird der Zeiger auf die Callback-Funktion übergeben und sie liefert einen `Long`-Wert mit der Funktionsadresse als Wert zurück.

7

Dateien und Verzeichnisse

7.1 Was Sie in diesem Kapitel erwartet

Unter VBA können Sie die eingebauten Befehle und Funktionen zur Dateimanipulation benutzen, aber daneben stehen dem Programmierer auch andere Objekte zur Verfügung, die den Umgang mit Dateien zum Teil erheblich vereinfachen. In diesem Kapitel werden die verschiedenen Möglichkeiten und Objekte dazu vorgestellt.

Es wird dabei näher darauf eingegangen, wie Sie Dateien suchen, löschen und die Dateiattribute wie beispielsweise die Zeit der Dateierstellung auslesen und manipulieren können. Des Weiteren werden Möglichkeiten vorgestellt, ganze Verzeichnisse inklusive den darin enthaltenen Dateien und Unterverzeichnissen zu kopieren und zu verschieben. Mithilfe der API ist es auch möglich, diese Verzeichnisse in den Papierkorb zu schieben, ohne vorher die einzelnen Dateien und Unterverzeichnisse gelöscht zu haben.

Mit ein paar Zeilen Code können Sie in einem Rutsch komplette Dateipfade anlegen, ohne dazu erst jedes einzelne Verzeichnis zu erstellen. Ein weiteres Thema ist die Umwandlung eines normalen Pfads zu einer existierenden Datei in den kürzeren MS-DOS-Pfad in der 8+3-Notation.

In einem weiteren Beispiel wird Ihnen gezeigt, wie Sie die Pfade zu den Sonderverzeichnissen wie beispielsweise das Temp-, Programm- oder Windowsverzeichnis auslesen können.

Zum Schluss wird noch Code vorgestellt, mit dem man beispielsweise Textdateien an einer bestimmten Textstelle kürzen kann, ohne sie vorher zu löschen und neu anzulegen.

7.2 Allgemeines

7.2.1 VBA-Befehle und -Funktionen

VBA besitzt einige eingebaute VBA-Funktionen und -Anweisungen, wie zum Beispiel `Kill`, `Name`, `Dir`, `MkDir`, `RmDir`, `ChDir`, `CurDir`, `GetAttr`, `SetAttr` usw. Wenn Sie nur einfache Dateioperationen durchführen wollen, kommen Sie damit auch hervorragend zurecht.

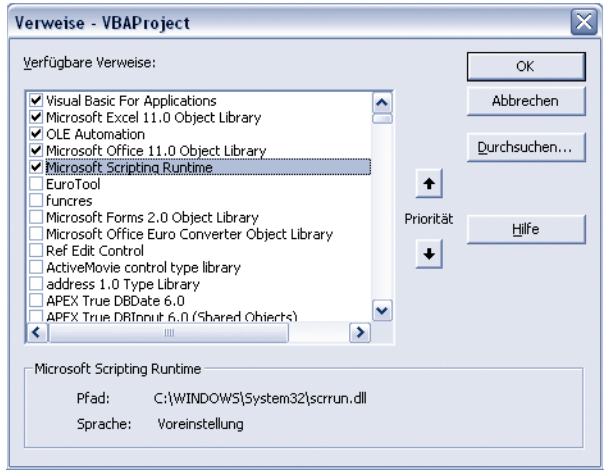
7.2.2 FileSearch-Objekt

In vielen Fällen hilft das `FileSearch`-Objekt weiter. Dieses Objekt ist eine Instanz der Klasse `FileSearch`, die wiederum Bestandteil von Microsoft Office ist. Laut Onlinehilfe repräsentiert dieses Objekt die Funktionalität des Dialogfeldes `Öffnen` im Menü `Datei`. Um dieses zu benutzen, muss auch nichts nachinstalliert werden.

7.2.3 FileSystemObject

Eine weitere Möglichkeit zur Dateimanipulation bietet die Bibliothek Scripting Runtime, die sich in der DLL `ScrRun.Dll` verbirgt. Verwenden Sie das `FileSystemObject`, das ein Element davon ist, muss diese Datei auf ihrem System verfügbar sein. Um die Typen und Konstanten aus der Typenbibliothek zu benutzen, benötigen Sie einen Verweis (Abbildung 7.1) darauf.

Abbildung 7.1
Verweis auf die Scripting Runtime



Sie können aber auch ohne Typenbibliothek mit diesem Objekt arbeiten, dazu erzeugen Sie sich mit `CreateObject` eine Objektinstanz:

```
Set FSO = CreateObject("Scripting.FileSystemObject")
```

Die Typen und Konstanten stehen Ihnen dann allerdings nicht mehr zur Verfügung. Am besten erstellen Sie sich in dem Fall die benötigten Konstanten selbst.

Das ist auf jeden Fall besser, als die reinen Zahlenwerte zu benutzen. Ein weiterer Nachteil dieser späten Bindung (late Binding) ist, dass auch Intellisense ohne die Einbindung der Typenbibliothek für dieses Objekt nicht mehr funktioniert.

7.2.4 API

Die Lösungen mit API-Funktionen sind nicht ganz so einfach zu realisieren. Dafür sind sie aber ungemein schnell und auf nahezu allen Windows-Systemen verfügbar. Auch VBA, das `FileSearch`- und das `FileSystem`-Objekt bedienen sich im Hintergrund dieser Funktionen und kapseln diese nur.

Ein weiterer Vorteil beim Einsatz der API ist, dass Sie damit Dinge anstellen können, die von den kapselnden Objekten nicht bereitgestellt werden. Als Beispiel ist das Verschieben in den Papierkorb zu nennen, das mit ein paar API-Funktionen im Handumdrehen erledigt ist.

7.2.5 DOS-Befehle

Sie haben auch die Möglichkeit, `DOS`-Befehle zu benutzen. Mit der `Shell`-Anweisung können Sie diese folgendermaßen ausführen:

```
Shell Environ$("COMSPEC") & " /C " & Befehle
```

Die Umgebungsvariable `COMSPEC`, die mit `Environ$("COMSPEC")` ausgelesen wird, gibt dabei den Kommandozeileninterpreter zurück.

Die Ausgaben können Sie aber mit VBA nicht so einfach auslesen. Es ist zwar möglich, Programme an der Konsole zu starten und auch aus dieser Konsole zu lesen. Ich habe so etwas auch schon realisiert, aber es ist nicht gerade trivial. Ob sich der Aufwand für ein paar Dateibefehle lohnt, sei dahingestellt. Im einfachsten Fall können Sie die Ausgaben in eine Textdatei umleiten und diese dann programmgesteuert auslesen. Die folgende Zeile würde die Ausgabe in die Datei `c:\ping.txt` umleiten.

```
Shell Environ$("COMSPEC") & " /C " & "ping 127.0.0.1 > c:\ping.txt"
```

7.3 Dateien suchen

Häufig werden Sie mit der Aufgabe konfrontiert, in einem Verzeichnis nach Dateien zu suchen, die einem bestimmten Muster entsprechen. Das ist nicht immer eine leichte Aufgabe, besonders wenn auch Unterverzeichnisse in die Suche einbezogen werden müssen. Unter Umständen kann die Suche sehr lange dauern, die Suchgeschwindigkeit hängt dabei nicht nur davon ab, wie umfangreich die Verzeichnisstruktur ist, sondern auch die Suchmethode hat einen großen Einfluss darauf.

Achtung

Beim Messen der verschiedenen Laufzeiten spielt es eine große Rolle, ob der Verzeichnisbaum schon einmal durchlaufen wurde. Ist das der Fall, geht die Suche erheblich schneller vonstatten. Wahrscheinlich steht die Dateiliste nach der ersten Suche irgendwo im Cache und braucht nicht erst neu ausgelesen zu werden. Wenn Sie dem nicht Rechnung tragen, kann das die Messergebnisse maßgeblich beeinflussen.

7.3.1 Dir

Diese Funktion wird häufig eingesetzt, um zu überprüfen, ob eine Datei, die einem bestimmten Suchkriterium entspricht, in einem Verzeichnis vorhanden ist. Beim nochmaligen Aufruf ohne ein Argument wird dann jeweils der nächste Dateiname zurückgeliefert, der dem aktuellen Suchkriterium entspricht.

Achtung

Übergeben Sie an diese Funktion den String »Falsch«, wird nicht etwa ein leerer String zurückgeliefert, wie es bei der Übergabe eines nicht existierenden Pfades geschieht, sondern der String »Falsch«. Das kann zu Problemen führen, wenn damit überprüft werden soll, ob beispielsweise `Application.GetOpenFilename` eine Datei zurückgeliefert hat. Wenn Abbrechen gewählt wurde, enthält eine Stringvariable, der das Ergebnis des Dialoges zugewiesen wurde, nämlich genau diesen String.

Wenn die Unterverzeichnisse mit in die Suche einbezogen werden sollen, ergibt sich jedoch ein Problem. `Dir` lässt sich nämlich nicht rekursiv einsetzen, da Sie keine neue Suche starten können, ohne die alten Einstellungen zu überschreiben. Ein verändertes Suchmuster, beispielsweise durch die Angabe eines neuen Pfades, beginnt eine neue Suche und Sie haben keine Möglichkeit, die vorherige Suche an der Position fortzusetzen, an der Sie mit den neuen Parametern begonnen haben.

Dies kann jedoch vermieden werden, indem die Unterverzeichnisse zwischengespeichert werden, bis die komplette Suche auf einer Verzeichnisebene abgeschlossen ist. Danach können Sie dann mit der gleichen Prozedur nacheinander die Unterverzeichnisse abarbeiten.

Nachfolgendes Beispiel (Listing 7.1) demonstriert dies:

Listing 7.1
Beispiele\07_Dateien\07_01_Dateien.xls\mdlDir

```
Public Sub myFileListDir()
   Dim strPath    As String
   Dim strFilter  As String
   Dim varItem    As Variant
   Dim objWS      As Worksheet
   Dim colList    As New Collection
   Dim i          As Long
   Dim dtmBegin   As Date
```

Listing 7.1 (Forts.)
Beispiele\07_Dateien\
07_01_Dateien.xls\mdlDir

```
    dtmBegin = Time

    ' Zieltabelle festlegen
    Set objWS = Worksheets("Dateiliste")

    ' strFilter
    strFilter = objWS.Range("B2")

    ' Suchpfad
    strPath = objWS.Range("B3")

    ' Suche starten
    myFilesearch strPath, colList, strFilter

    With objWS ' Zieltabelle

        .Range("A5:H65535").ClearContents ' Zielbereich säubern

        i = 4 ' Bei Zeile 5 anfangen, einzutragen

        For Each varItem In colList
        ' Alle Elemente der Collection durchlaufen
           i = i + 1
           .Cells(i, 1) = varItem ' Eintragen
        Next

    End With
    MsgBox "Dir = " & Format(Time - dtmBegin, "nn:ss")
End Sub

Private Sub myFilesearch( _
    ByVal strStart As String, _
    ByRef colList As Collection, _
    Optional ByVal strFilter As String)

    Dim astrFolder()  As String
    Dim i             As Long
    Dim strCurFolder  As String
    Dim strFile       As String

    On Error Resume Next

    ' Erst einmal 100 Unterverzeichnisse annehmen
    ReDim astrFolder(1 To 100)

    If Left(strFilter, 1) <> "*" Then strFilter = "*" & strFilter

    If Right$(strStart, 1) <> "\" Then

        ' Nachschauen, ob übergebener Pfad auch einen
        ' Backslash enthält. Wenn nicht, dann anhängen
        strStart = strStart & "\"

    End If

    strCurFolder = strStart

    ' Alle Dateien liefern
    strStart = strStart & "*"
```

Listing 7.1 (Forts.)
Beispiele\07_Dateien\
07_01_Dateien.xls\mdlDir

```
    ' Suche mit Dir initialisieren
    strFile = Dir(strStart, vbSystem Or _
       vbHidden Or vbDirectory Or vbNormal)

    Do While strFile <> ""
    ' So lange durchlaufen, wie
    ' durch Dir etwas geliefert wird

        If GetAttr(strCurFolder & strFile) And vbDirectory Then
        ' wenn Datei ein Verzeichnis ist
            If Right$(strFile, 1) <> "." Then
                ' und zwar ein untergeordnetes,
                ' (Punkte sind Übergeordnete Verzeichnisse)

                i = i + 1

                If i > UBound(astrFolder) Then

                    ' Wenn Array zu klein ist, anpassen
                    ReDim Preserve astrFolder(1 To i + 1)

                End If

                ' dann ein Array mit Verzeichnissen füllen.
                astrFolder(i) = strFile

            End If
       Else
       ' Handelt es sich um eine Datei,

          If LCase(strFile) Like LCase(strFilter) Then
          ' und entspricht sie noch den strFilterbedingungen,

             ' dann den Pfad an die Collection colList hängen.
             colList.Add strCurFolder & strFile, _
                strCurFolder & strFile

          End If
       End If
       strFile = Dir$()
    Loop

    ' Keine Unterverzeichnisse vorhanden, dann beenden
    If i = 0 Then Exit Sub

    ' Array anpassen
    ReDim Preserve astrFolder(1 To i)

    ' Jetzt erst werden die Unterverzeichnisse abgearbeitet,
    ' weil Dir$ mit Rekursionen nicht klarkommt.
    For i = 1 To UBound(astrFolder)

       ' Jetzt ruft sich diese Prozedur noch einmal auf.
       myFilesearch strCurFolder & astrFolder(i), colList, strFilter

    Next

End Sub
```

myFileListDir

Diese Prozedur holt sich aus dem Tabellenblatt `Dateiliste` den Suchpfad und den Filter. Zusammen mit einer neuen Collection werden diese Parameter an die Prozedur `myFilesearch` übergeben. Nach der Rückkehr enthält die Collection die gefundenen Dateien inklusive Pfad dorthin.

Diese Pfade, die in der Collection gespeichert sind, werden anschließend in einer Schleife in die Spalte A eingetragen.

myFilesearch

In dieser Prozedur wird die Suche mit `Dir` so initialisiert, dass nacheinander alle Dateien eines übergebenen Verzeichnisses geliefert werden. Zu den Dateien gehören auch unter- und übergeordnete Verzeichnisse. Die Dateien mit dem Namen . (Punkt) und .. (Doppelpunkt) stehen für das gleiche- und das übergeordnete Verzeichnis.

Unterverzeichnisse werden in einem Array zwischengespeichert, Dateien, die dem Suchkriterium entsprechen, werden zusammen mit dem Pfad zu der als Referenz übergebenen Collection hinzugefügt.

Nachdem die Suche in dieser Verzeichnisebene beendet ist, wird für jedes Unterverzeichnis die gleiche Prozedur noch einmal aufgerufen, aber mit dem um das Unterverzeichnis erweiterten Suchpfad. Die aufgerufene Prozedur wird auch erst beendet, wenn alle ihre Unterverzeichnisse abgearbeitet sind.

7.3.2 FileSearch-Objekt

Mithilfe des `FileSearch`-Objektes können Sie die Suche wesentlich komfortabler, kompakter und schneller bewerkstelligen (Listing 7.2).

Listing 7.2
Beispiele\07_Dateien\
07_01_Dateien.xls\mdlFileSearch

```
Public Sub myFileListFS()
   Dim strPath    As String
   Dim strFilter  As String
   Dim objWS      As Worksheet
   Dim i          As Long
   Dim objFS      As Filesearch
   Dim dtmBegin   As Date

   dtmBegin = Time

   ' Objektvariable des Objekts anlegen
   Set objFS = Application.Filesearch

   ' Zieltabelle festlegen
   Set objWS = Worksheets("Dateiliste")

   ' strFilter
   strFilter = objWS.Range("B2")

   ' Suchpfad
   strPath = objWS.Range("B3")
```

Listing 7.2 (Forts.)
Beispiele\07_Dateien\
07_01_Dateien.xls\mdlFileSearch

```
    ' Suche starten
    With objFS
        .NewSearch
        .LookIn = strPath
        .SearchSubFolders = True
        .Filename = "*" & strFilter
        .MatchTextExactly = False
        .FileType = msoFileTypeAllFiles
        .Execute
    End With

    With objWS ' Zieltabelle

       .Range("A5:H65535").ClearContents ' Zielbereich säubern

       For i = 1 To objFS.FoundFiles.Count
       ' Alle Elemente durchlaufen

          .Cells(i + 4, 1) = objFS.FoundFiles(i) ' Eintragen

       Next

    End With

    MsgBox "Filesearch = " & Format(Time - dtmBegin, "nn:ss")

End Sub
```

myFileListFS

Damit das `FileSearch`-Objekt weiß, nach was für einer Datei und in welchem Verzeichnis gesucht werden soll, müssen verschiedene Eigenschaften des Objektes gesetzt werden. Außerdem können Sie optional über verschiedene andere Eigenschaften noch Einfluss auf die Suche selbst nehmen.

Die Methode `NewSearch` setzt zu Beginn die Einstellungen des Objektes zurück. Mit der `LookIn`-Eigenschaft legen Sie den Suchpfad, mit `Filename` die Suchmaske fest. Bei der Suchmaske können Sie auch Platzhalterzeichen * (Sternchen) oder ? (Fragezeichen) benutzen. Das Fragezeichen ersetzt ein Zeichen, das Sternchen beliebig viele.

Wenn die Eigenschaft `SearchSubFolders` auf `True` gesetzt wird, werden auch die Unterverzeichnisse durchsucht.

Mit der `FileType`-Eigenschaft legen Sie die Art der zu suchenden Dateien fest. Im Objektkatalog (Abbildung 7.2) können Sie nachschauen, welche Konstanten es in der Enum `MsoFileType` gibt.

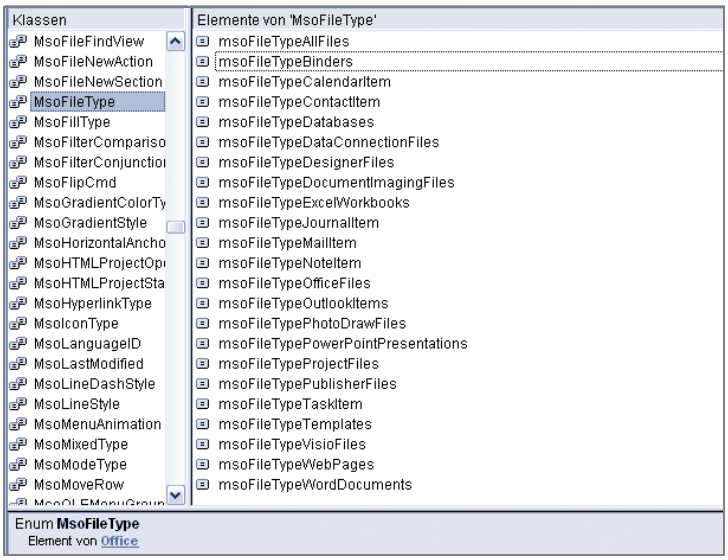


Abbildung 7.2
MsoFileType-Konstanten im Objektkatalog

Beispielsweise enthält die `msoFileTypeOfficeFiles`-Konstante alle Dateien mit den Erweiterungen **.doc*, **.dot*, **.htm*, **.html*, **.mdb*, **.mpd*, **.obd*, **.obt*, **.ppt*, **.pps*, **.pot*, **.xlt* und **.xls*.

Die Eigenschaft `MatchTextExactly` legt mit einem booleschen Wert fest, ob nur ein Teil oder wenn auf `True` gesetzt, der ganze Dateiname mit dem Parameter `Filename` übereinstimmen soll.

Wird `LastModified` gesetzt, geben Sie an, dass nur Dateien angezeigt werden, die in bestimmten Zeiträumen geändert wurden. Möglich sind folgende vordefinierte Konstanten, wobei die Namen für sich sprechen:

- `msoLastModifiedAnyTime`
- `msoLastModifiedLastMonth`
- `msoLastModifiedLastWeek`
- `msoLastModifiedThisMonth`
- `msoLastModifiedThisWeek`
- `msoLastModifiedToday`
- `msoLastModifiedYesterday`

Mit der `Execute`-Methode beginnen Sie die Suche, dabei können Sie noch Parameter mit übergeben. Der erste Parameter `SortBy` legt das Sortierkriterium fest. Möglich sind folgende Konstanten, der Standardwert ist `msoSortByFileName`:

- `msoSortByFileName`
- `msoSortByFileType`
- `msoSortByLastModified`
- `msoSortByNone`
- `msoSortBySize`

Der zweite Parameter `SortOrder` legt die Reihenfolge fest. Möglich sind folgende Konstanten, Standardwert ist dabei `msoSortOrderAscending`, also die aufsteigende Reihenfolge.

- `msoSortOrderAscending`
- `msoSortOrderDescending`

Die Eigenschaft `FoundFiles` liefert anschließend ein Objekt, das die gefundenen Dateien enthält. Dieses besitzt als Auflistung die Eigenschaft `Count`, mit der Sie die Anzahl der in der Auflistung enthaltenen Elemente feststellen können. Über die Standardeigenschaft Item zusammen mit dem Index können Sie auf die einzelnen Elemente zugreifen.

Diese Elemente werden anschließend in einem Tabellenblatt eingetragen.

7.3.3 Scripting.FileSystemObject

Zum Durchsuchen von Verzeichnissen können Sie auch das `FileSystemObject` (`FSO`) der Scripting Runtime (Listing 7.3) benutzen.

Listing 7.3
Beispiele\07_Dateien\
07_01_Dateien.xls\mdlFSO

```
Public Sub myFileListFSO()
   Dim strPath    As String
   Dim strFilter  As String
   Dim varItem    As Variant
   Dim objWS      As Worksheet
   Dim colList    As New Collection
   Dim i          As Long
   Dim dtmBegin   As Date

   dtmBegin = Time

   ' Zieltabelle festlegen
   Set objWS = Worksheets("Dateiliste")

   ' strFilter
   strFilter = objWS.Range("B2")

   ' Suchpfad
   strPath = objWS.Range("B3")

   ' Suche starten
   myFilesearchFSO strPath, colList, strFilter

   With objWS ' Zieltabelle

      .Range("A5:H65535").ClearContents ' Zielbereich säubern

      i = 4 ' Bei Zeile 5 anfangen, einzutragen

      For Each varItem In colList
      ' Alle Elemente der Collection durchlaufen
         i = i + 1
         .Cells(i, 1) = varItem ' Eintragen
      Next

   End With
```

Listing 7.3 (Forts.)
Beispiele\07_Dateien\
07_01_Dateien.xls\mdlFSO

```
   MsgBox "FSO = " & Format(Time - dtmBegin, "nn:ss")

End Sub

Private Sub myFilesearchFSO( _
   ByVal strStart As String, _
   ByRef colList As Collection, _
   Optional ByVal strFilter As String)

   Dim objFSO     As New Scripting.FileSystemObject
   Dim objCur     As Scripting.Folder
   Dim objSub     As Scripting.Folder
   Dim objFiles   As Files
   Dim objFile    As File

   If Left(strFilter, 1) <> "*" Then strFilter = "*" & strFilter

   If Right$(strStart, 1) <> "\" Then

       ' Nachschauen, ob übergebener Pfad auch einen
       ' Backslash enthält. Wenn nicht, dann anhängen
      strStart = strStart & "\"

   End If

   ' Verzeichnisliste aktueller Ordner
   Set objCur = objFSO.GetFolder(strStart)

   ' Dateiliste aktueller Ordner
   Set objFiles = objCur.Files

   For Each objFile In objFiles
      If LCase(objFile.Name) Like LCase(strFilter) Then
         colList.Add strStart & "\" & objFile.Name, _
            strStart & "\" & objFile.Name
      End If
   Next

   ' Unterverzeichnisse abarbeiten
   For Each objSub In objCur.SubFolders
      myFilesearchFSO strStart & objSub.Name & "\", _
         colList, strFilter
   Next
End Sub
```

myFileListFSO

Diese Prozedur holt sich aus dem Tabellenblatt `Dateiliste` den Suchpfad und den Filter. Zusammen mit einer neuen Collection werden diese Parameter an die Prozedur `myFilesearchFSO` übergeben. Nach der Rückkehr enthält die Collection die gefundenen Dateien inklusive Pfad dorthin.

Diese Pfade werden anschließend in einer Schleife in die Spalte A eingetragen.

myFilesearchFSO

Um Verbindung zu einem Verzeichnis herstellen zu können, benötigen Sie die `GetFolder`-Methode des `FSO`. Dieser Methode übergeben Sie als Parameter ein Verzeichnis als String und bekommen ein `Folder`-Objekt zurück, das Sie der Objektvariablen `objCur` zuweisen.

Die `Files`-Eigenschaft des nun verbundenen `Folder`-Objektes, liefert eine Auflistung zurück, die der Objektvariablen `objFiles` zugewiesen wird. Die einzelnen Elemente dieser Auflistungen sind vom Typ `File` und repräsentieren jeweils eine Datei. Die Dateinamen werden dann nacheinander über die `Name`-Eigenschaft ausgelesen und mit dem Suchkriterium verglichen. Passende Dateinamen werden zusammen mit dem Pfad zu der als Referenz übergebenen Collection hinzugefügt.

Die `Path`-Eigenschaft würde den kompletten Pfad zu der Datei, die `ShortName`- und `ShortPath`-Eigenschaften die Namen in alter (8.3) DOS-Schreibweise liefern.

Um auch die Unterverzeichnisse mit dem `FSO` auszulesen, wird nach dem Durchlaufen der `Files`-Auflistung die `SubFolders`-Auflistung ausgelesen. Für jedes Unterverzeichnis wird die Prozedur `myFilesearchFSO` noch einmal aufgerufen, aber mit dem um das Unterverzeichnis erweiterten Suchpfad. Die aufgerufene Prozedur wird auch erst beendet, wenn alle ihre Unterverzeichnisse abgearbeitet sind.

7.3.4 API

Wenn Sie eine vorhandene Datei in einem Verzeichnisbaum suchen wollen, können Sie eine API-Funktion aus der `imagehlp.dll` benutzen. Die Funktion `SearchTreeForFile` erlaubt zwar keine Platzhalterzeichen und findet nur die erste Datei mit diesem Namen, dafür ist sie aber auch extrem schnell (Listing 7.4).

Listing 7.4
Beispiele\07_Dateien\07_01_Dateien.xls\mdlOneFileSearch

```
Private Declare Function SearchTreeForFile _
   Lib "imagehlp" ( _
   ByVal RootPath As String, _
   ByVal InputPathName As String, _
   ByVal OutputPathBuffer As String _
   ) As Long

Private Const MAX_PATH = 260

Sub test()
MsgBox FindMyFile("boot.ini")
End Sub

Public Function FindMyFile( _
   strFile As String, _
   Optional strStart As String = "c:\" _
   ) As String

   Dim strBuffer  As String
   Dim lngRet     As Long
```

Listing 7.4 (Forts.)
Beispiele\07_Dateien\
07_01_Dateien.xls\
mdlOneFileSearch

```
    strBuffer = String(MAX_PATH, 0)

    If SearchTreeForFile(strStart, strFile, strBuffer) Then
       FindMyFile = Left(strBuffer, InStr(1, strBuffer, Chr(0)) - 1)
    End If
End Function
```

Falls dagegen mehrere Dateien mit gleichen Namen oder Suchmustern existieren und wenn Sie alle diese Dateien zurückbekommen wollen, müssen Sie den ganzen Verzeichnisbaum durchlaufen. Dafür sind die API-Funktionen `FindFirstFile`, `FindNextFile` und `FindClose` gedacht.

In diesem Beispiel wird eine selbst geschriebene Klasse benutzt, die eine Dateiliste mit den Informationen Pfad, Name, 8.3-Name, Erstellungszeitpunkt, Änderungszeitpunkt, letzter Zugriff und der Größe liefert.

Das Benutzen der Klasse (Listing 7.5) stellt sich ebenso einfach dar wie das Benutzen des `FileSystemOcject`. Sie legen einfach eine Instanz der Klasse an (`Dim APIClass As New clsVerzeichnisbaum`) und benutzen die Methode `MakeFileList`, der Sie den Anfangspfad übergeben und die ein Variantarray mit den Informationen zurückliefert.

Danach wird jedes Element des zurückgelieferten Arrays in das Tabellenblatt eingetragen. Jedes Element ist wiederum ein Array mit sieben Elementen, welche die Eigenschaften Name, 8.3-Name, Pfad, Erstellungszeitpunkt, Letzter Zugriff, Änderungszeitpunkt und Dateigröße repräsentieren.

Listing 7.5
Beispiele\07_Dateien\
07_01_Dateien.xls\mdlAPI

```
Public Sub myFileListAPI()
   Dim varItem    As Variant
   Dim objWS      As Worksheet
   Dim varList    As Variant
   Dim i          As Long
   Dim dtmBegin   As Date
   Dim APIClass   As New clsVerzeichnisbaum

   dtmBegin = Time

   ' Zieltabelle festlegen
   Set objWS = Worksheets("Dateiliste")

   ' strFilter
   APIClass.Filter = objWS.Range("B2")

   ' Suche starten
   varList = APIClass.MakeFileList(objWS.Range("B3"))

   With objWS ' Zieltabelle

      .Range("A5:H65535").ClearContents ' Zielbereich säubern

      For i = 1 To UBound(varList)
         ' Alle Elemente durchlaufen und eintragen
```

Listing 7.5 (Forts.)
Beispiele\07_Dateien\
07_01_Dateien.xls\mdlAPI

```
            ' Pfad und Namen
            .Cells(i + 4, 1) = varList(i)(3) & varList(i)(1)

            ' Namen
            .Cells(i + 1, 1) = varList(i)(1)

            ' 8+3-Namen
            .Cells(i + 4, 2) = varList(i)(2)

            ' Pfad
            .Cells(i + 4, 1) = varList(i)(3)

            ' Erstellungszeitpunkt
            .Cells(i + 4, 3) = varList(i)(4)

            ' Letzter Zugriff
            .Cells(i + 4, 4) = varList(i)(5)

            ' Geändert am
            .Cells(i + 4, 5) = varList(i)(6)

            ' Dateigröße
            .Cells(i + 4, 6) = varList(i)(7)

        Next

    End With

    MsgBox "API = " & Format(Time - dtmBegin, "nn:ss")

End Sub
```

Die eigentliche Arbeit erledigt die Klasse `clsVerzeichnisbaum` (Listing 7.6).

Listing 7.6
Beispiele\07_Dateien\
07_01_Dateien.xls\mdlAPI

```
Private Declare Function FindClose _
    Lib "kernel32" ( _
    ByVal hFindFile As Long _
    ) As Long

Private Declare Function FindFirstFile _
    Lib "kernel32" Alias "FindFirstFileA" ( _
    ByVal lpFileName As String, _
    lpFindFileData As WIN32_FIND_DATA _
    ) As Long

Private Declare Function FindNextFile _
    Lib "kernel32" Alias "FindNextFileA" ( _
    ByVal hFindFile As Long, _
    lpFindFileData As WIN32_FIND_DATA _
    ) As Long

Private Declare Function FileTimeToLocalFileTime _
    Lib "kernel32" ( _
    lpFileTime As FILETIME, _
    lpLocalFileTime As FILETIME _
    ) As Long
```

Listing 7.6 (Forts.)
Beispiele\07_Dateien\
07_01_Dateien.xls\mdlAPI

```
Private Declare Function FileTimeToSystemTime _
   Lib "kernel32" ( _
   lpFileTime As FILETIME, _
   lpSystemTime As SYSTEMTIME _
   ) As Long

Private Const FILE_ATTRIBUTE_DIRECTORY = &H10
Private Const MAX_PATH = 260

Private Type SYSTEMTIME
        wYear As Integer
        wMonth As Integer
        wDayOfWeek As Integer
        wDay As Integer
        wHour As Integer
        wMinute As Integer
        wSecond As Integer
        wMilliseconds As Integer
End Type

Private Type FILETIME
        dwLowDateTime As Long
        dwHighDateTime As Long
End Type

Private Type WIN32_FIND_DATA
        dwFileAttributes As Long
        ftCreationTime As FILETIME
        ftLastAccessTime As FILETIME
        ftLastWriteTime As FILETIME
        nFileSizeHigh As Long
        nFileSizeLow As Long
        dwReserved0 As Long
        dwReserved1 As Long
        cFileName As String * MAX_PATH
        cAlternate As String * 14
End Type

Private mavarFileList() As Variant
Private mlngIndex       As Long
Private mcurSize        As Currency
Private mstrFilter      As String

Public Function MakeFileList(strPath As String)
    ' Öffentliche Methode zum Zugriff und
    ' Erstellen einer Dateiliste
   On Error Resume Next

   ' Einfach einmal 1000 Elemente vorgeben
   ReDim mavarFileList(1 To 1000)
   mlngIndex = 0

   ' Das Arbeitspferd aufrufen
   mcurSize = LoopPath(strPath, mstrFilter)
```

Listing 7.6 (Forts.)
Beispiele\07_Dateien\07_01_Dateien.xls\mdlAPI

```
    If mlngIndex = 0 Then
        ' Keine entsprechende Datei gefunden
       ReDim mavarFileList(0)
    Else
        ' Größe des Arrays anpassen
       ReDim Preserve mavarFileList(1 To mlngIndex)
    End If

    ' Array zurückgeben
    MakeFileList = mavarFileList

End Function

Public Property Get FullSize() As Currency
    ' Gesamtgröße
    FullSize = mcurSize
End Property

Public Property Get FilesCount() As Long
    ' Anzahl der Dateien
    FilesCount = mlngIndex
End Property

Public Property Get Filter() As String
    ' Dateifilter
    Filter = mstrFilter
End Property

Public Property Let Filter(ByVal vNewValue As String)
    ' Dateifilter
    mstrFilter = vNewValue
End Property

Private Function LoopPath( _
    ByVal strPath As String, _
    Optional strFilter As String _
    ) As Currency

    Dim lngSearchHandle As Long
    Dim lngRet As Long
    Dim strSearch As String
    Dim udtFind As WIN32_FIND_DATA
    Dim strFileName As String
    Dim strDosName As String
    Dim avarProperty(1 To 7)
    Dim curFolderSize As Currency

    ' Führende und nachfolgende Leerzeichen entfernen
    strPath = Trim(strPath)

    ' Wenn nötig, Backslash anhängen
    If Right$(strPath, 1) <> "\" Then strPath = strPath & "\"

    ' Alle Dateien suchen
    strSearch = strPath & "*"
```

Listing 7.6 (Forts.)
Beispiele\07_Dateien\
07_01_Dateien.xls\mdlAPI

```
With udtFind

   .cAlternate = String(14, Chr(0))  ' Puffer 8+3
   .cFileName = String(260, Chr(0))  ' Puffer Dateiname

   ' Erstes Filehandle auf dieser Ebene ermitteln
   lngSearchHandle = FindFirstFile(strSearch, udtFind)
   lngRet = lngSearchHandle

   Do While lngRet <> 0
      ' Datei oder Verzeichnis gefunden

      ' Namen am NullChar kürzen
      strFileName = StrSpaceNullTrim(.cFileName)
      strDosName = StrSpaceNullTrim(.cAlternate)

      If strFileName <> ".." And strFileName <> "." Then
         ' Directory oder File gefunden. Gleiches- (.),
         ' oder übergeordnetes Verzeichnis (..) ignorieren

         If (.dwFileAttributes And FILE_ATTRIBUTE_DIRECTORY) _
            = FILE_ATTRIBUTE_DIRECTORY Then

            ' Rekursiver Aufruf, wenn Unterverzeichnis
            curFolderSize = curFolderSize + _
               LoopPath((strPath & strFileName), strFilter)

         Else
            ' Datei gefunden

            If LCase(Right$(strFileName, Len(strFilter))) =
               LCase(strFilter) Then
               ' Infos in Array avarProperty kopieren,
               ' wenn die Filterbedingungen stimmen

               avarProperty(1) = strFileName
               If Len(strDosName) = 0 Then
                  strDosName = strFileName
               avarProperty(2) = strDosName
               avarProperty(3) = strPath
               avarProperty(4) = ChangeTime( _
                  .ftCreationTime)
               avarProperty(5) = ChangeTime( _
                  .ftLastAccessTime)
               avarProperty(6) = ChangeTime( _
                  .ftLastWriteTime)
               avarProperty(7) = .nFileSizeLow

               ' Dateigrößen zusammenzählen
               curFolderSize = curFolderSize + _
                  CCur(.nFileSizeLow)

               '
               mlngIndex = mlngIndex + 1
```

Listing 7.6 (Forts.)
Beispiele\07_Dateien\
07_01_Dateien.xls\mdlAPI

```
                        ' Wenn mehr Dateien vorhanden, als mavarFileList
                        ' aufnehmen kann, Array redimensionieren und Werte
                        ' beibehalten
                        If mlngIndex > UBound(mavarFileList) Then _
                           ReDim Preserve mavarFileList(1 To _
                              mlngIndex + 1000)

                        mavarFileList(mlngIndex) = avarProperty

                     End If
                  End If
               End If

               .cAlternate = String(14, Chr(0)) ' Puffer 8+3
               .cFileName = String(260, Chr(0)) ' Puffer Dateiname

               ' Nächste Datei
               lngRet = FindNextFile(lngSearchHandle, udtFind)

            Loop

         End With

         LoopPath = curFolderSize
         FindClose lngSearchHandle
      End Function

      Private Function StrSpaceNullTrim(sTxt As String) As String
         ' Beim ersten Auftreten von VbNullChar kürzen
         StrSpaceNullTrim = Trim(Left(sTxt, InStr(1, sTxt, Chr(0)) - 1))
      End Function

      Private Function ChangeTime(udtFiletime As FILETIME) As Date
         Dim udtSysTime    As SYSTEMTIME
         Dim udtFiletime1  As FILETIME

         ' Umwandlung udtFiletime in Systemzeit
         FileTimeToLocalFileTime udtFiletime, udtFiletime1
         FileTimeToSystemTime udtFiletime1, udtSysTime

         With udtSysTime
            If .wYear >= 1900 Then
               ChangeTime = _
                  CDbl( _
                  DateSerial(.wYear, .wMonth, .wDay) _
                  + _
                  TimeSerial(.wHour, .wMinute, .wSecond) _
                  )
            Else
               ChangeTime = 0
            End If
         End With

      End Function
```

MakeFileList

In der nach außen hin öffentlichen Funktion `MakeFileList` wird ein Variantarray angelegt. Dieses enthält bei der Initialisierung erst einmal 1000 Elemente. Die Größe des Arrays ist dynamisch und wird mit `ReDim Preserve` angepasst, wenn im weiteren Verlauf mehr Elemente gebraucht werden. Dabei werden die Dimensionen des Arrays in größeren Schritten angepasst, um ein allzu häufiges Redimensionieren zu vermeiden. Wenn Sie häufig mit größeren Verzeichnissen arbeiten, kann es durchaus von Vorteil sein, die Anfangsgröße und Schrittweite höher zu wählen. Als Funktionsergebnis wird das gefüllte Array zurückgegeben.

LoopPath

An die Funktion `LoopPath` werden der Anfangspfad und der Dateifilter übergeben. Wenn nötig, wird in dieser Funktion ein Backslash an den übergebenen Pfad gehängt. Die Variable `strSearch` nimmt das Suchmuster in Form des Pfades und einem abschließenden Sternchen als Platzhalter für beliebige Buchstabenfolgen auf. Das Sternchen deshalb, weil alle Dateien, darunter auch Verzeichnisse, gefunden werden sollen.

Um die Suche zu beginnen, wird noch eine Variable des Typs `WIN32_FIND_DATA` mit dem Namen `udtFind` angelegt, die nachher die Dateieigenschaften aufnimmt. In dieser Struktur muss auch jeweils ein Puffer für den File- und den 8.3-Namen angelegt werden. Dazu wird dem Element `cAlternate` ein Stringpuffer in der Größe von 14 Zeichen und dem Element `cFileName` ein Puffer von 260 Zeichen zugewiesen. Das muss jedes Mal gemacht werden, bevor eine der API-Funktionen diese Struktur ausfüllt.

Der API-Funktion `FindFirstFile` wird als Parameter die Variable `strSearch`, die das Suchmuster angibt, und die Struktur `udtFind` übergeben. Der Rückgabewert dieser Funktion ist das Suchhandle (ein Longwert) auf dieser Ebene, die Struktur `udtFind` wird mit den Eigenschaften der ersten Datei gefüllt.

Daraufhin wird geprüft, ob es sich bei der zurückgelieferten Datei (auch Verzeichnisse sind in Wirklichkeit Dateien) um Verzeichnisse handelt. Innerhalb der Struktur `udtFind` liefert das Element `dwFileAttributes` die Attribute der Datei. Ist das Flag `FILE_ATTRIBUTE_DIRECTORY` gesetzt, was Sie mit einer binären UND-Verknüpfung überprüfen können, handelt es sich dabei um ein Verzeichnis. Wenn eine normale Datei gefunden wurde, werden alle Eigenschaften in ein Array geschrieben und dieses Array als Element eines anderen Variantarrays abgelegt.

Die Struktur liefert Dateizeiten in einem anderen Format, nämlich im Format `FileTime`. Das muss erst mit der Funktion `FileTimeToLocalFileTime` in das Gebietsschema und mit `FileTimeToSystemTime` in eine Struktur `SYSTEMTIME` umgewandelt werden. Aber auch das ist noch nicht der benötigte Datentyp `Date`, deshalb die anschließende Umwandlung mit `DateSerial` und `TimeSerial`.

Bei Dateigrößen, die den positiven Bereich eines Long überschreiten, das sind etwa 2 Gbyte, muss man sich überlegen, ob man die Größe anders ermittelt. Momentan wird nur `nFileSizeLow` ausgewertet. Folgendes müsste in dem Fall funktionieren:

```
'In den Deklarationsbereich der Klasse

Private Type myCur
     x As Currency
End Type

Private Type copyCur
     x As Long
     y As Long
End Type

'Und als Ersatz für die Zeile
'avarProperty(7) =.nFileSizeLow

Dim udtLength As myCur
Dim udtCC As copyCur
Dim strRes As String

udtCC.x =.nFileSizeLow
udtCC.y =.nFileSizeHigh
LSet udtLength = udtCC
strRes = Format((udtLength.x * CCur(10000)), "#.##0")
avarProperty(7) = strRes
```

Wenn die gefundene Datei ein Verzeichnis ist, wird geprüft, ob es sich dabei um ein untergeordnetes handelt. Ein Punkt und ein Doppelpunkt werden ausgeschlossen. Haben Sie ein untergeordnetes Verzeichnis gefunden, rufen Sie die Funktion `LoopPath` noch einmal mit dem Unterverzeichnis als Startverzeichnis auf. Finden Sie dort wieder ein untergeordnetes Verzeichnis, wird die Funktion noch einmal aufgerufen. Das Spiel wiederholt sich auch in der nächsten Ebene, und zwar so lange, bis alle Unterverzeichnisse und Unter/Unter/... Verzeichnisse gelesen sind. Das nennt man eine Rekursion.

Um an die nächste Datei auf dieser Ebene zu kommen, benutzen Sie die API-Funktion `FindNextFile`. Dieser Funktion wird als Parameter das von `FindFirstFile` gelieferte Suchhandle und die Struktur `udtFind` übergeben, wobei vorher die Puffer `cAlternate` und `cFileName` neu angelegt sein müssen.

Beendet wird die Funktion, wenn alle Dateien dieser Ebene ausgelesen wurden, was Sie daran erkennen, dass `FindFirstFile` oder `FindNextFile` eine Null zurückliefert. Ist der ganze Verzeichnisbaum gelesen, stehen alle Dateieigenschaften in einem Variantarray, das auch von der Klasse zurückgegeben wird.

Die Klasse besitzt auch noch die schreibgeschützten Eigenschaften `FilesCount` (Anzahl Dateien) und `FullSize` (gesamter belegter Speicher).

7.3.5 Fazit

Die Geschwindigkeitsunterschiede der verschiedenen Methoden beim Durchsuchen einer Ebene sind zwar prozentual groß, spielen aber beim Durchsuchen einer Verzeichnisebene absolut gesehen keine große Rolle. Deshalb ist das Benutzen der `Dir`-Funktion bei Dateien einer Verzeichnisebene kein großer Nachteil.

Selbst beim Durchsuchen ganzer Verzeichnisbäume ist die Dir-Funktion noch ein wenig schneller als das FileSystemObject (FSO) der Scripting Runtime. Dass das FSO beim Durchsuchen von Verzeichnisbäumen so langsam ist, kommt daher, dass wie bei Dir immer nur eine Verzeichnisebene durchlaufen werden kann und Rekursionen eingesetzt werden müssen.

Wenn auf Geschwindigkeit geachtet werden muss und es sich um größere, verschachtelte Verzeichnisse handelt, kann das FileSearch-Objekt benutzt werden. Nach meinen Tests ist das etwa fünfmal schneller als Dir und das FSO.

Was die Geschwindigkeit angeht, ist die Lösung mit der API aber nicht zu schlagen. Beim Durchsuchen eines größeren Verzeichnisses hat die Dir-Funktion ca. 50, das FSO ca. 60, das FileSearch-Objekt ca. 14 und die Lösung mit der API weniger als eine Sekunde benötigt.

7.4 Dateiattribute lesen und schreiben

7.4.1 GetAttr/SetAttr

Zum Auslesen der Attribute einer Datei oder eines Verzeichnisses gibt es unter VBA die Funktion GetAttr, die einen Wert zurückgibt, bei dem einzelne Bits als Flags (Tabelle 7.1) für verschiedene Eigenschaften dienen. Hier die Syntax:

```
GetAttr(Pfadname)
```

Da jedes Bit unabhängig von den anderen gesetzt oder nicht gesetzt sein kann, ist prinzipiell eine beliebige Kombination der Flags ohne Beeinflussung der anderen möglich. Ob ein Bit oder Flag gesetzt ist, können Sie mit einem binären And überprüfen.

```
If myAttr And vbReadOnly Then MsgBox "Flag gesetzt"
```

Tabelle 7.1
Attributes Flags

Name	Wert	Bedeutung
vbNormal	0	Normale Datei
vbReadOnly	1	Schreibgeschützte Datei
vbHidden	2	Versteckt
vbSystem	4	Systemdatei. Beim Macintosh nicht verfügbar
vbDirectory	16	Verzeichnis oder Ordner
vbArchive	32	Datei wurde seit dem letzten Sichern geändert. Nicht bei Macintosh
vbAlias	64	Dateiname ist ein Alias. Nur beim Macintosh verfügbar

Wenn Sie die Attribute setzen wollen, können Sie das mit SetAttr erledigen.

```
SetAttr pathname, attributes
```

Um ein einzelnes Flag zu setzen, können Sie mit einem binären Or arbeiten.

```
SetAttr pathname, attributes Or vbArchive ' Setzt das Flag vbArchive
```

Mithilfe einer binären And-Verknüpfung und der negierten Konstanten kann es wieder gelöscht werden.

```
SetAttr pathname, attributes And (Not vbArchive) ' Löscht das Flag
```

7.4.2 FileSystemObject

Die GetFolder-Methode des FSO stellt eine Verbindung zu einem Verzeichnis her. Zurückgegeben wird ein Folder-Objekt. Die Files-Eigenschaft des nun verbundenen Folder-Objektes, liefert eine Auflistung zurück, die Elemente vom Typ File enthält. Diese Elemente repräsentieren jeweils eine Datei.

Die Attributes-Eigenschaft eines File-Objekts gibt die Dateiattribute wieder oder setzt diese (nur die Werte 0, 1, 2, 4, 32 können gesetzt werden). Folgende Werte (Tabelle 7.2) sind einzeln oder kombiniert möglich:

Tabelle 7.2
Attributes Flags

Wert	Bedeutung	Wert	Bedeutung
0	Normale Datei	16	Verzeichnis
1	ReadOnly-Datei	32	Geänderte Datei oder Archiv
2	Versteckte Datei	64	Verknüpfung
4	Systemdatei	128	Komprimierte Datei
8	Laufwerk		

Auswerten können Sie das mit

```
If myFile.Attributes And 2 Then MsgBox "Versteckte Datei"
```

Um ein einzelnes Flag zu setzen, können Sie mit einem binären Or arbeiten.

```
myFile.Attributes= myFile.Attributes Or 2 ' Setzt das Flag
"Versteckte Datei"
```

Mithilfe einer binären And-Verknüpfung und dem negierten Wert kann das Flag wieder gelöscht werden:

```
myFile.Attributes= myFile.Attributes And (Not 2) ' Löscht das Flag
"Versteckte Datei"
```

Die DateCreated-Eigenschaft des Objektes liefert den Erstellungszeitpunkt, die DateLastAccessed-Eigenschaft den Zeitpunkt des letzten Dateiaufrufs und DateLastModified gibt den Zeitpunkt der letzten Änderung der Datei zurück.

7.4.3 API

Um die Attribute einer Datei auszulesen, können Sie das Element `dwFileAttributes` aus der Struktur WIN32_FIND_DATA benutzen. Diese Struktur wird ausgefüllt, wenn Sie eine der Funktionen `FindFirstFile` und `FindNextFile` benutzen. In Listing 7.6 finden Sie ein Beispiel, wie Sie mithilfe des Elementes `dwFileAttributes` erkennen, ob es sich um ein Verzeichnis handelt.

Hier die möglichen Werte, auch in Kombination:

```
Public Const FILE_ATTRIBUTE_ARCHIVE = &H20
Public Const FILE_ATTRIBUTE_DIRECTORY = &H10
Public Const FILE_ATTRIBUTE_HIDDEN = &H2
Public Const FILE_ATTRIBUTE_NORMAL = &H80
Public Const FILE_ATTRIBUTE_READONLY = &H1
Public Const FILE_ATTRIBUTE_SYSTEM = &H4
Public Const FILE_ATTRIBUTE_TEMPORARY = &H100
```

Auch hier sollten Sie mir dem binären `And` die gesetzten Flags auslesen können.

Wenn die Datei feststeht, deren Attribute ausgelesen werden sollen, können Sie die API-Funktion `GetFileAttributes` benutzen. Zum Setzen der Attribute wird die Funktion `SetFileAttributes` verwendet. Nachfolgend (Listing 7.7) wird der Einsatz dieser zwei Funktionen demonstriert:

Listing 7.7
Beispiele\07_Dateien\
07_01_Dateien.xls\mdlAttrAPI

```
Private Declare Function SetFileAttributes _
   Lib "kernel32" Alias "SetFileAttributesA" ( _
   ByVal lpFileName As String, _
   ByVal dwFileAttributes As Long _
   ) As Long

Private Declare Function GetFileAttributes _
   Lib "kernel32" Alias "GetFileAttributesA" ( _
   ByVal lpFileName As String _
   ) As Long

Private Declare Function GetTempPath _
   Lib "kernel32" Alias "GetTempPathA" ( _
   ByVal nBufferLength As Long, _
   ByVal lpBuffer As String _
   ) As Long

Const FILE_ATTRIBUTE_ARCHIVE = &H20
Const FILE_ATTRIBUTE_DIRECTORY = &H10
Const FILE_ATTRIBUTE_HIDDEN = &H2
Const FILE_ATTRIBUTE_NORMAL = &H80
Const FILE_ATTRIBUTE_READONLY = &H1
Const FILE_ATTRIBUTE_SYSTEM = &H4
Const FILE_ATTRIBUTE_TEMPORARY = &H100

Public Sub TestAttribute()
   Dim strFile As String
   Dim strPath As String
   Dim lngRet  As Long

   ' Temp-Pfad ermitteln
```

Listing 7.7 (Forts.)
Beispiele\07_Dateien\
07_01_Dateien.xls\mdlAttrAPI

```
    strPath = String(255, 0)
    lngRet = GetTempPath(255, strPath)

    ' Pfad und Name zusammenfügen
    strFile = Left(strPath, lngRet) & "~AttrTest.txt"

    ' Eventuell vorhandene Datei löschen
    If Dir(strFile) <> "" Then Kill strFile

    ' Datei anlegen
    Open strFile For Binary As #1: Close

    ' Dateiattribute vor Änderung
    MsgBox GetAttrString(strFile), , "Vor Änderung"

    ' Attribute anpassen
    AddFileAttr strFile, FILE_ATTRIBUTE_READONLY
    AddFileAttr strFile, FILE_ATTRIBUTE_TEMPORARY
    RemoveFileAttr strFile, FILE_ATTRIBUTE_ARCHIVE

    MsgBox GetAttrString(strFile), , "Nach Änderung"

    ' Schreibschutz löschen, damit Kill wieder funzt
    RemoveFileAttr strFile, FILE_ATTRIBUTE_READONLY

    ' Explorer mit ausgewähltem Pfad öffnen
    Shell "explorer.exe " & strPath, vbMaximizedFocus

End Sub

Public Sub AddFileAttr(strFileName As String, NewAttr As Long)
    Dim lngAttr As Long

    ' Attribute auslesen
    lngAttr = GetFileAttributes(strFileName)

    ' Flag setzen
    lngAttr = lngAttr Or NewAttr

    ' Attribute ändern
    SetFileAttributes strFileName, lngAttr

End Sub

Public Sub RemoveFileAttr(strFileName As String, OldAttr As Long)
    Dim lngAttr As Long

    ' Attribute auslesen
    lngAttr = GetFileAttributes(strFileName)

    ' Flag löschen
    lngAttr = lngAttr And Not (OldAttr)

    ' Attribute ändern
    SetFileAttributes strFileName, lngAttr

End Sub
```

Listing 7.7 (Forts.)
Beispiele\07_Dateien\
07_01_Dateien.xls\mdlAttrAPI

```
Public Function GetAttrString(strFileName As String) As String
   Dim lngAttr As Long
   Dim strAtt  As String

   ' Attribute auslesen
   lngAttr = GetFileAttributes(strFileName)

   ' Abfragen, ob einzelne Flags gesetzt sind und
   ' einen String mit dem Ergebnis erzeugen
   strAtt = "Archiv =" & _
      CBool(lngAttr And FILE_ATTRIBUTE_ARCHIVE) & vbCrLf
   strAtt = strAtt & "Directory =" & _
      CBool(lngAttr And FILE_ATTRIBUTE_DIRECTORY) & vbCrLf
   strAtt = strAtt & "HIDDEN =" & _
      CBool(lngAttr And FILE_ATTRIBUTE_HIDDEN) & vbCrLf
   strAtt = strAtt & "READONLY =" & _
      CBool(lngAttr And FILE_ATTRIBUTE_READONLY) & vbCrLf
   strAtt = strAtt & "SYSTEM =" & _
      CBool(lngAttr And FILE_ATTRIBUTE_SYSTEM) & vbCrLf
   strAtt = strAtt & "TEMPORARY =" & _
      CBool(lngAttr And FILE_ATTRIBUTE_TEMPORARY) & vbCrLf
   strAtt = strAtt & "Normal =" & _
      CBool(lngAttr = 0)

   ' Erzeugten String zurückgeben
   GetAttrString = strAtt

End Function
```

TestAttribute

In dieser Prozedur wird mit der API-Funktion `GetTempPath` das aktuelle *Temp*-Verzeichnis ermittelt. Dazu wird ein Stringpuffer angelegt und zusammen mit der Länge des Puffers als Parameter an die Funktion übergeben. Das Funktionsergebnis enthält die Textlänge des in den Puffer geschriebenen Pfades. Der Puffer, der mit einer Größe von 255 Zeichen angelegt wurde, wird dann auf die zurückgelieferte Länge gestutzt.

Anschließend wird an den Pfad zum *Temp*-Verzeichnis noch ein beliebiger Dateiname angehängt, in diesem Beispiel wird der Name `~AttrTest.txt` verwendet. Die Tilde wird nur benutzt, damit sich die später angelegte Datei im *Temp*-Verzeichnis leichter wiederfinden lässt. Dadurch steht Sie nämlich bei aufsteigender Sortierung im Explorer ziemlich am Anfang der Dateiliste.

Existiert die Datei schon, wird sie mit der `Kill`-Anweisung gelöscht. Ist das Schreibschutzattribut dieser Datei gesetzt, funktioniert diese Anweisung aber nicht.

Um eine Beispieldatei im *Temp*-Verzeichnis anzulegen, benutzen Sie die `Open`-Anweisung im Modus `Binary` (`Append`, `Output` oder `Random` ist auch möglich). Damit wird eine Datei angelegt, wenn Sie nicht bereits existiert.

Mit der Funktion `GetAttrString` wird ein String erzeugt, der die Dateiattribute enthält. Dieser wird in einer Messagebox ausgegeben. Anschließend werden Dateiattribute mit `RemoveFileAttr` gelöscht und mit `AddFileAttr` gesetzt. Die neuen Dateiattribute werden wiederum ausgelesen und in einer Messagebox ausgegeben.

Zum Schluss wird noch der Explorer mit dem *Temp*-Verzeichnis als Anfangspfad geöffnet.

AddFileAttr

Mit `GetFileAttributes` werden die Attribute ausgelesen und in der Variablen `lngAttr` gespeichert. In dieser wird mit dem binären `Or` das neue Attribut gesetzt. Mit `SetFileAttributes` werden die geänderten Attribute zurückgeschrieben.

RemoveFileAttr

Mit `GetFileAttributes` werden die Attribute ausgelesen und in der Variablen `lngAttr` gespeichert. In dieser wird mit dem binären `And Not` das übergebene Attribut gelöscht. Mit `SetFileAttributes` werden die geänderten Attribute zurückgeschrieben.

GetAttrString

In dieser Funktion werden mit `GetFileAttributes` die Attribute ausgelesen und die einzelnen Bits daraufhin geprüft, ob sie gesetzt sind. Jedes Attribut wird durch eine eigene durch `vbCrLf` getrennte Zeile im String repräsentiert und anschließend als Funktionsergebnis zurückgegeben.

7.5 Dateizeiten lesen und schreiben

Wenn Sie sich im Explorer die Dateien eines Verzeichnisses anschauen und als Ansicht DETAILS eingestellt haben, stellen Sie fest, dass Dateien noch mehr Attribute besitzen als nur den Namen und die Dateiattribute wie zum Beispiel den Schreibschutz. Besonders die Zeiten der Erstellung, der letzten Änderung und des letzten Zugriffs sind wichtige Hilfsmittel, vor allem, wenn es darum geht, die neuesten Versionen zu suchen oder ältere zu entsorgen.

Diese Zeiten anzupassen kann beispielsweise notwendig sein, um alle zusammengehörenden Dateien auf einen einheitlichen Erstellungszeitpunkt zu setzen, um irgendwann mit den Suchoptionen des Explorers diese Dateien wiederzufinden. Motive, die eine Änderung der Zeiten rechtfertigen, gibt es viele, wie immer kann so etwas aber auch missbraucht werden, um beispielsweise zu verschleiern, dass man selbst eine Datei unwiederbringlich verhunzt hat. Ich bin aber der Meinung, dass das Positive überwiegt und wer wirklich Missbrauch treiben will, findet immer Mittel und Wege.

7.5.1 FileDateTime

Die `FileDateTime`-Funktion von VB liefert aber nur die Zeit der letzten Änderung, außerdem kann die Zeit nicht gesetzt werden.

```
MsgBox FileDateTime("c:\Config.sys")
```

7.5.2 FileSystemObject

Das `FileSystemObject` liefert drei Zeiten. Ein Ändern der Dateizeiten ist aber nicht möglich (Listing 7.8).

Listing 7.8
Beispiele\07_Dateien\
07_01_Dateien.xls\
mdlFileTime_VBA_FSO

```
Private Sub TestFileTimeFSO()
   Dim strFile As String
   Dim strMsg  As String
   Dim objFSO  As Object
   Dim objFile As Object

   ' Dateipfad festlegen
   strFile = "c:\Config.sys"

   ' Objektinstanz erzeugen. Late Binding (ohne Verweis)
   Set objFSO = CreateObject("Scripting.FileSystemObject")

   ' Fileobjekt erzeugen
   Set objFile = objFSO.GetFile(strFile)

   ' Zeiten extrahieren
   With objFile
      strMsg = strMsg & "Erstellt: " & _
         .DateCreated & vbCrLf
      strMsg = strMsg & "Letzter Zugriff: " & _
         .DateLastAccessed & vbCrLf
      strMsg = strMsg & "Letzte Änderung: " & _
         .DateLastModified
   End With

   ' Zeiten ausgeben
   MsgBox strMsg, vbInformation, strFile

End Sub
```

7.5.3 API

Die API bietet Funktionen, die es erlauben, Dateizeiten zu lesen und zu schreiben.

Folgendermaßen (Listing 7.9) werden die Funktionen benutzt:

Listing 7.9
Beispiele\07_Dateien\
07_01_Dateien.xls\
mdlFileTime_API

```
Private Declare Function GetTempPath _
   Lib "kernel32" Alias "GetTempPathA" ( _
   ByVal nBufferLength As Long, _
   ByVal lpBuffer As String _
   ) As Long
```

Listing 7.9 (Forts.)
Beispiele\07_Dateien\07_01_Dateien.xls\mdlFileTime_API

```
Public Sub testFiletime()
   Dim strFile    As String
   Dim strPath    As String
   Dim lngRet     As Long
   Dim dteCreate  As Date

   ' Temp-Pfad ermitteln
   strPath = String(255, 0)
   lngRet = GetTempPath(255, strPath)
   strPath = Left(strPath, lngRet)

   ' Pfad und Name zusammenfügen
   strFile = strPath & "~~FileTime.tmp"

   ' Datei anlegen
   Open strFile For Binary As #1: Close

   ' Erstellungszeitpunkt auslesen
   dteCreate = GetCreateTime(strFile)

   ' Erstellungszeitpunkt ausgeben
   MsgBox dteCreate, , "Erstellungszeitpunkt"

   ' Einen Monat abziehen
   dteCreate = DateSerial( _
      Year(dteCreate), Month(dteCreate) - 1, Day(dteCreate)) + _
      TimeSerial( _
      Hour(dteCreate), Minute(dteCreate), Second(dteCreate))

   ' Erstellungszeitpunkt ändern
   SetMyFileTime strFile, dteCreate

   MsgBox GetCreateTime(strFile), , "Erstellungszeitpunkt"

   ' Explorer mit ausgewähltem Pfad öffnen
   Shell "explorer.exe " & strPath, vbMaximizedFocus

End Sub
```

Nachfolgend (Listing 7.10) der Code, der das Ändern und Auslesen der Dateizeiten ermöglicht.

Listing 7.10
Beispiele\07_Dateien\07_01_Dateien.xls\mdlFileTime_API

```
Private Declare Function CreateFile _
   Lib "kernel32" Alias "CreateFileA" ( _
   ByVal lpFileName As String, _
   ByVal dwDesiredAccess As Long, _
   ByVal dwShareMode As Long, _
   ByVal lpSecurityAttributes As Long, _
   ByVal dwCreationDisposition As Long, _
   ByVal dwFlagsAndAttributes As Long, _
   ByVal hTemplateFile As Long _
   ) As Long

Private Const GENERIC_READ          As Long = &H80000000
Private Const GENERIC_WRITE         As Long = &H40000000
Private Const FILE_SHARE_READ       As Long = &H1&
Private Const FILE_SHARE_WRITE      As Long = &H2&
```

Listing 7.10 (Forts.)
Beispiele\07_Dateien\
07_01_Dateien.xls\
mdlFileTime_API

```
Private Const OPEN_EXISTING         As Long = &H3&
Private Const INVALID_HANDLE_VALUE  As Long = -&H1&

Private Declare Function CloseHandle _
   Lib "kernel32" ( _
   ByVal hObject As Long) As Long

Private Declare Function SetFileTime _
   Lib "kernel32" ( _
   ByVal hFile As Long, _
   lpCreationTime As FILETIME, _
   lpLastAccessTime As FILETIME, _
   lpLastWriteTime As FILETIME _
   ) As Long

Private Declare Function GetFileTime _
   Lib "kernel32" ( _
   ByVal hFile As Long, _
   lpCreationTime As FILETIME, _
   lpLastAccessTime As FILETIME, _
   lpLastWriteTime As FILETIME _
   ) As Long

Private Declare Function LocalFileTimeToFileTime _
   Lib "kernel32" ( _
   lpLocalFileTime As FILETIME, _
   lpFileTime As FILETIME _
   ) As Long

Private Declare Function SystemTimeToFileTime _
   Lib "kernel32" ( _
   lpSystemTime As SYSTEMTIME, _
   lpFileTime As FILETIME _
   ) As Long

Private Declare Function FileTimeToLocalFileTime _
   Lib "kernel32" ( _
   lpFileTime As FILETIME, _
   lpLocalFileTime As FILETIME _
   ) As Long

Private Declare Function FileTimeToSystemTime _
   Lib "kernel32" ( _
   lpFileTime As FILETIME, _
   lpSystemTime As SYSTEMTIME _
   ) As Long

Private Const FILE_ATTRIBUTE_DIRECTORY = &H10
Private Const MAX_PATH = 260

Private Type SYSTEMTIME
        wYear As Integer
        wMonth As Integer
        wDayOfWeek As Integer
        wDay As Integer
        wHour As Integer
        wMinute As Integer
```

Listing 7.10 (Forts.)
Beispiele\07_Dateien\07_01_Dateien.xls\mdlFileTime_API

```
        wSecond As Integer
        wMilliseconds As Integer
End Type

Private Type FILETIME
        dwLowDateTime As Long
        dwHighDateTime As Long
End Type

Private Function GetCreateTime(strFilename As String) As Date
   Dim dtDatum                As Date
   Dim hwndFile               As Long
   Dim udtCreationFileTime    As FILETIME
   Dim udtLastAccessFileTime  As FILETIME
   Dim udtLastWriteFileTime   As FILETIME

   If Dir(strFilename) = "" Then Exit Function

   'Filehandle holen
   hwndFile = CreateFile(strFilename, GENERIC_WRITE, _
       FILE_SHARE_WRITE, ByVal 0&, _
       OPEN_EXISTING, 0&, 0&)

   ' Dateizeiten holen
   GetFileTime hwndFile, _
      udtCreationFileTime, _
      udtLastAccessFileTime, _
      udtLastWriteFileTime

   ' Filetime in Lokalzeit umwandeln
   GetCreateTime = FiletimeToNormalTime(udtCreationFileTime)

   'Filehandle schließen
   CloseHandle hwndFile

End Function

Private Function GetLastAccessTime(strFilename As String) As Date
   Dim dtDatum                As Date
   Dim hwndFile               As Long
   Dim udtCreationFileTime    As FILETIME
   Dim udtLastAccessFileTime  As FILETIME
   Dim udtLastWriteFileTime   As FILETIME

   If Dir(strFilename) = "" Then Exit Function

   'Filehandle holen
   hwndFile = CreateFile(strFilename, GENERIC_WRITE, _
       FILE_SHARE_WRITE, ByVal 0&, _
       OPEN_EXISTING, 0&, 0&)

   ' Dateizeiten holen
   GetFileTime hwndFile, _
      udtCreationFileTime, _
      udtLastAccessFileTime, _
      udtLastWriteFileTime
```

Listing 7.10 (Forts.)
Beispiele\07_Dateien\
07_01_Dateien.xls\
mdlFileTime_API

```
    ' Filetime in Lokalzeit umwandeln
    GetLastAccessTime = FiletimeToNormalTime(udtLastAccessFileTime)

    'Filehandle schließen
    CloseHandle hwndFile

End Function

Private Function GetLastWriteTime(strFilename As String) As Date
    Dim dtDatum                As Date
    Dim hwndFile               As Long
    Dim udtCreationFileTime    As FILETIME
    Dim udtLastAccessFileTime  As FILETIME
    Dim udtLastWriteFileTime   As FILETIME

    If Dir(strFilename) = "" Then Exit Function

    'Filehandle holen
    hwndFile = CreateFile(strFilename, GENERIC_WRITE, _
        FILE_SHARE_WRITE, ByVal 0&, _
        OPEN_EXISTING, 0&, 0&)

    ' Dateizeiten holen
    GetFileTime hwndFile, _
       udtCreationFileTime, _
       udtLastAccessFileTime, _
       udtLastWriteFileTime

    ' Filetime in Lokalzeit umwandeln
    GetLastWriteTime = FiletimeToNormalTime(udtLastWriteFileTime)

    'Filehandle schließen
    CloseHandle hwndFile

End Function

Private Sub SetMyFileTime( _
    strFilename As String, _
    Optional dteCreationTime As Date, _
    Optional dteLastAccessTime As Date, _
    Optional dteLastWriteTime As Date)

    Dim dtDatum                As Date
    Dim hwndFile               As Long
    Dim udtCreationFileTime    As FILETIME
    Dim udtLastAccessFileTime  As FILETIME
    Dim udtLastWriteFileTime   As FILETIME
    Dim lngRet As Long
    If Dir(strFilename) = "" Then Exit Sub

    'Filehandle holen
    hwndFile = CreateFile(strFilename, GENERIC_WRITE, _
        FILE_SHARE_WRITE, ByVal 0&, _
        OPEN_EXISTING, 0&, 0&)
```

Listing 7.10 (Forts.)
Beispiele\07_Dateien\07_01_Dateien.xls\mdlFileTime_API

```
    ' Dateizeiten holen
    GetFileTime hwndFile, _
       udtCreationFileTime, _
       udtLastAccessFileTime, _
       udtLastWriteFileTime

    If dteCreationTime Then
       'Erstellungszeitpunkt ändern, wenn übergeben
       udtCreationFileTime = NormalToFileTime(dteCreationTime)
    End If

    If dteLastAccessTime Then
       'Letzten Zugriff ändern, wenn übergeben
       udtLastAccessFileTime = NormalToFileTime(dteLastAccessTime)
    End If

    If dteLastWriteTime Then
       'Letzte Änderung ändern, wenn übergeben
       udtLastWriteFileTime = NormalToFileTime(dteLastWriteTime)
    End If

    'Filezeiten zurückschreiben
    lngRet = SetFileTime(hwndFile, _
       udtCreationFileTime, _
       udtLastAccessFileTime, _
       udtLastWriteFileTime)

    'Filehandle schließen
    CloseHandle hwndFile

End Sub

Private Function NormalToFileTime( _
   ByVal dteDateTime As Date _
   ) As FILETIME
   Dim udtSysTime As SYSTEMTIME

   dteDateTime = NormalToGMT(dteDateTime)

   With udtSysTime
        .wYear = Year(dteDateTime)
        .wMonth = Month(dteDateTime)
        .wDay = Day(dteDateTime)
        .wDayOfWeek = WeekDay(dteDateTime) - 1
        .wHour = Hour(dteDateTime)
        .wMinute = Minute(dteDateTime)
        .wSecond = Second(dteDateTime)
   End With

   'Umwandlung Systemzeit zu Filezeit.
   SystemTimeToFileTime udtSysTime, NormalToFileTime

End Function
```

Listing 7.10 (Forts.)
Beispiele\07_Dateien\
07_01_Dateien.xls\
mdlFileTime_API

```
Private Function FiletimeToNormalTime( _
   udtFileTime As FILETIME _
   ) As Date

   Dim udtSysTime As SYSTEMTIME

   FileTimeToSystemTime udtFileTime, udtSysTime

   With udtSysTime
      FiletimeToNormalTime = _
         CDbl( _
         DateSerial(.wYear, .wMonth, .wDay) _
         + _
         TimeSerial(.wHour, .wMinute, .wSecond) _
         )
   End With

   FiletimeToNormalTime = GmtToNormal(FiletimeToNormalTime)

End Function

Private Function NormalToGMT(DatumZeit As Date)
   Dim dteBegin   As Date
   Dim dteEnd     As Date
   Dim dteYear    As Long

   dteYear = Year(DatumZeit)
   dteBegin = DateSerial(dteYear, 4, 0) - (WeekDay( _
       DateSerial(dteYear, 4, 0), 2) Mod 7) + TimeSerial(2, 0, 0)
   dteEnd = DateSerial(dteYear, 11, 0) - (WeekDay( _
       DateSerial(dteYear, 11, 0), 2) Mod 7) + TimeSerial(2, 0, 0)

   If DatumZeit > dteBegin And DatumZeit < dteEnd Then
       NormalToGMT = DatumZeit - TimeSerial(2, 0, 0)
   Else
       NormalToGMT = DatumZeit - TimeSerial(1, 0, 0)
   End If
End Function

Private Function GmtToNormal(DatumZeit As Date)
   Dim dteBegin   As Date
   Dim dteEnd     As Date
   Dim dteYear    As Long

   dteYear = Year(DatumZeit)
   dteBegin = DateSerial(dteYear, 4, 0) - (WeekDay( _
       DateSerial(dteYear, 4, 0), 2) Mod 7) + TimeSerial(1, 0, 0)
   dteEnd = DateSerial(dteYear, 11, 0) - (WeekDay( _
       DateSerial(dteYear, 11, 0), 2) Mod 7) + TimeSerial(1, 0, 0)

   If DatumZeit > dteBegin And DatumZeit < dteEnd Then
       GmtToNormal = DatumZeit + TimeSerial(2, 0, 0)
   Else
       GmtToNormal = DatumZeit + TimeSerial(1, 0, 0)
   End If
End Function
```

testFiletime

Mit der API-Funktion `GetTempPath` wird der für den angemeldeten Benutzer gültige *Temp*-Pfad ermittelt. Dazu wird ein Stringpuffer angelegt und zusammen mit der Angabe der Puffergröße übergeben. Der Rückgabewert gibt die Anzahl der Zeichen zurück, die in den Puffer geschrieben wurden.

Anschließend wird an den Pfad zum *Temp*-Verzeichnis noch ein beliebiger Dateiname angehängt, in diesem Beispiel wird der Name `~~FileTime.tmp` verwendet. Die Tilden werden nur benutzt, damit sich die später angelegte Datei im *Temp*-Verzeichnis leichter wiederfinden lässt. Dadurch steht Sie nämlich bei aufsteigender Sortierung im Explorer ziemlich am Anfang der Dateiliste.

Um eine Beispieldatei im *Temp*-Verzeichnis anzulegen, benutzen Sie die `Open`-Anweisung im Modus `Binary` (`Append`, `Output` oder `Random` ist auch möglich). Damit wird eine Datei angelegt, wenn Sie nicht bereits existiert.

Anschließend wird mit der Funktion `GetCreateTime` der Erstellungszeitpunkt ausgelesen und in einer Messagebox ausgegeben. Von dem Erstellungsdatum wird noch ein Monat abgezogen und mit `SetMyFileTime` der Erstellungszeitpunkt geändert. Dann wird noch einmal das Erstellungsdatum ausgelesen und ausgegeben. Zum Schluss wird mit der `Shell`-Anweisung noch der Explorer mit dem *Temp*-Pfad geöffnet.

GetCreateTime, GetLastAccessTime, GetLastWriteTime

Mit der API-Funktion `CreateFile` holen Sie sich ein Handle auf die Datei. Anschließend können Sie mit der API-Funktion `GetFileTime` die drei Zeiten der Datei auslesen. Die drei Zeiten liegen aber in einer `Filetimestruktur` vor, und zwar in der Greenwich Mean Time, `GMT`. Die Greenwich-Zeit (Weltzeit) ist die lokale Zeit am Nullmeridian im Londoner Vorort Greenwich.

Die gewünschte Zeit wird in die lokale umgewandelt und zurückgegeben. Zum Umwandeln in die lokale Zeit benutzen Sie die Funktion `FileTimeToNormalTime`.

FileTimeToNormalTime

Die Funktion `FileTimeToNormalTime` dient dazu, aus einer `FILETIME`-Struktur eine Variable vom Typ `Date` in der auf das jeweilige System passenden Ortszeit zu erzeugen. Die Dateiinformationen selbst werden nämlich im GMT-Format gespeichert.

Dazu wird mit der Funktion `GmtToNormal` das übergebene Datum in das lokale Format transferiert und damit eine Struktur vom Typ `SYSTEMTIME` ausgefüllt. Mit der API `SystemTimeToFileTime` wird diese Struktur in eine `FILETIME`-Struktur umgewandelt und anschließend als Funktionsergebnis zurückgegeben.

NormalTimeToFileTime

Die Funktion `NormalTimeToFileTime` dient dazu, aus einer Variablen vom Typ `Date`, in der ein Zeitpunkt der lokal geltenden Zeitzone steckt, eine `FILETIME`-Struktur zu machen, die als `GMT` vorliegt.

Dazu wird mit der Funktion NormalToGMT das übergebene Datum in das GMT-Format transferiert und damit eine Struktur vom Typ SYSTEMTIME ausgefüllt. Mit der API-Funktion SystemTimeToFileTime wird diese Struktur in eine FILETIME-Struktur umgewandelt und anschließend als Funktionsergebnis zurückgegeben.

SetMyFileTime

Die Prozedur SetMyFileTime dient dazu, die Zeiten einer Datei zu ändern. Dazu werden drei Strukturen vom Typ FILETIME benötigt, welche die neuen Zeitinformationen aufnehmen, es handelt sich dabei um die Strukturen dteCreationFileTime, dteLastAccessFileTime und dteLastWriteFileTime.

Anschließend benötigen Sie ein Filehandle auf die Zieldatei mit der Berechtigung zum Schreiben. Das erledigt die Funktion CreateFile, die im Erfolgsfall das Handle als Funktionsergebnis liefert. Wird Null zurückgeliefert, deutet das auf einen Fehler hin, weil entweder die Datei nicht existiert oder man nicht über die benötigten Rechte verfügt. In diesem Fall wird die Prozedur verlassen.

Erst werden die aktuellen Dateizeiten mit der API-Funktion GetFileTime auslesen. Danach füllen Sie die zu ändernden Filetime-Strukturen mit den neuen Zeiten, wobei Sie die interne Funktion NormalTimeToFileTime zur Umwandlung benutzen.

Mit der API-Funktion SetFileTime können Sie anschließend die Zeiten der Datei ändern. Dazu wird als erster Parameter das Filehandle übergeben, die anderen Parameter sind die drei FILETIME-Strukturen.

Am Ende der Prozedur muss das Filehandle noch geschlossen werden.

GmtToNormal und NormalToGMT

Zum Umwandeln der Zeiten von und zu den lokalen Zeitzonen gibt es die API-Funktionen FileTimeToLocalFileTime und LocalFileTimeToFileTime. Diese zu benutzen ist an sich keine größere Aktion, Sie füllen eine einfache Struktur vom Typ Systemzeit aus und wandeln sie mit SystemTimeToFileTime in die etwas undurchsichtigere Struktur FileTime um. Diese lässt sich dann mit den angesprochenen Funktionen manipulieren.

Leider funkt da die Sommerzeit dazwischen. Die Umwandlungsfunktionen benutzen bei der Berücksichtigung unglücklicherweise den momentan eingestellten Offset zur lokalen Zeit. Wenn Sie also im Sommer diese Funktionen benutzen, wird je nach Funktion noch eine Stunde hinzugezählt oder abgezogen, unabhängig davon, ob die Zeit, die umgewandelt werden soll, auch tatsächlich im Sommer liegt. Wird damit beispielsweise eine Zeit im Winter umgewandelt und bei der Ausführung der Funktion ist gerade Sommerzeit, werden zwei Stunden zur GMT zugeschlagen, obwohl nur eine Stunde richtig wäre.

Um das zu umgehen bin ich einen anderen Weg gegangen. Dazu errechne ich die Eckpunkte, ab der die Sommerzeit beginnt und endet. Sie beginnt am letzten Sonntag im März und endet am letzten Sonntag im Oktober jeweils um 01:00 Uhr UTC (Weltzeit). Das gilt seit 1996 in Deutschland und soweit ich weiß auch in jedem Mitgliedstaat der Europäischen Union, für andere Gebiete müsste die Berechnung der Eckpunkte den tatsächlichen Gegebenheiten angepasst werden.

Liegt die übergebene Zeit im Bereich der Sommerzeit, werden zwei Stunden gegenüber GMT berücksichtigt, im anderen Fall eine Stunde.

Die Umwandlung in eine lesbare Struktur vom Typ `SYSTEMTIME` erfolgt mit der API `FileTimeToSystemTime` und diese Struktur kann anschließend mithilfe von `DateSerial` und `TimeSerial` in das Datenformat `Date` umgewandelt werden.

7.6 Komplette Pfade anlegen

Um komplette Pfade mit allen Verzeichnissen anzulegen, müssen Sie von der Wurzel aus anfangen und mit `MkDir` nacheinander alle Verzeichnisse anlegen. Eine Möglichkeit dazu zeigt folgendes Beispiel (Listing 7.11):

Listing 7.11
Beispiele\07_Dateien\
07_01_Dateien.xls\mdlMKDelDir

```
Private Sub TestCreatePathVBA()
   VBA_CreatePath "c:\CopyDelNew\VBA\VBA\VBA"
End Sub

Public Sub VBA_CreatePath(ByVal strPath As String)
   Dim varPath As Variant
   Dim i       As Long

   On Error Resume Next

   If Right(strPath, 1) = "\" Then _
      strPath = Left(strPath, Len(strPath) - 1)

   varPath = Split(strPath, "\")

   strPath = varPath(0)

   For i = 1 To UBound(varPath)
      strPath = strPath & "\" & varPath(i)
      MkDir strPath
   Next

End Sub
```

Für solch eine simple Aufgabe ist das doch eine recht aufwändige Lösung. Mit der API-Funktion `MakeSureDirectoryPathExists` (Listing 7.12) lässt sich die gleiche Aufgabe aber viel eleganter erledigen.

Listing 7.12
Beispiele\07_Dateien\
07_01_Dateien.xls\mdlMKDelDir

```
Private Declare Function MakePath _
   Lib "imagehlp.dll" Alias "MakeSureDirectoryPathExists" ( _
   ByVal lpPath As String _
   ) As Long

Private Sub TestCreatePathAPI()
   API_CreatePath "c:\CopyDelNew\API\API\API\"
End Sub

Public Function API_CreatePath(strPath As String) As Boolean
   If Right(strPath, 1) <> "\" Then strPath = strPath & "\"
   If MakePath(strPath) <> 0 Then API_CreatePath = True
End Function
```

Sie müssen nur darauf achten, dass an das Ende des anzulegenden Pfades ein Backslash gehört, sonst wird das letzte Unterverzeichnis nicht angelegt.

7.7 Dateioperationen mit der API

Wollen Sie mit VBA ganze Verzeichnisse löschen, müssen Sie mit der `Kill`-Anweisung erst alle Dateien und danach mit `RmDir` alle Unterverzeichnisse löschen. Ein weiterer Nachteil von `Kill`, `FSO` und Co. ist der, dass gelöschte Dateien oder Verzeichnisse nicht erst im Papierkorb landen, von wo sie im Bedarfsfall wieder hervorgeholt werden könnten.

Glücklicherweise werden fast alle Funktionalitäten des Betriebssystems über die API bereitgestellt und lassen sich zum größten Teil auch unter VBA benutzen. Dazu gehört auch die Funktion `SHFileOperation`, mit der Sie Dateien und Verzeichnisse löschen, umbenennen, kopieren und verschieben können. Als Ergebnis liefert diese Funktion einen Longwert zurück, der null ist, wenn die Funktion erfolgreich war. Hier die Deklarationsanweisung und ein paar Erläuterungen dazu:

```
Private Declare Function SHFileOperation Lib "shell32.dll" Alias
"SHFileOperationA" ( pFileOp As Any ) As Long
```

Der Übergabeparameter `lpFileOp` ist ein Zeiger auf eine `SHFILEOPSTRUCT`-Struktur.

```
Private Type SHFILEOPSTRUCT
    hwnd As Long
    wFunc As Long
    pFrom As String
    pTo As String
    fFlags As Integer
    fAnyOperationsAborted As Long
    hNameMappings As Long
    lpszProgressTitle As String
End Type
```

- `Hwnd`

  Das Handle des Besitzerfensters. Für Dialoge, die beim Ausführen auftreten können.

- `wFunc`

  Damit wird der Funktion mitgeteilt, welche Operation ausgeführt werden soll.

  ```
  Private Const FO_DELETE = &H3
  ```

  Die Datei oder das Verzeichnis wird gelöscht.

  ```
  Private Const FO_MOVE = &H1
  ```

  Die Datei oder das Verzeichnis wird verschoben.

  ```
  Private Const FO_RENAME = &H4
  ```

  Die Datei oder das Verzeichnis wird umbenannt.

  ```
  Private Const FO_COPY = &H2&
  ```

  Die Datei oder das Verzeichnis wird kopiert.

- `pFrom`

  Hier wird ein Pointer auf einen String übergeben, der die Quelldatei oder das Quellverzeichnis angibt. An dem String muss ein doppeltes `vbNullChar` angehängt sein, da auch Listen von Dateien/Verzeichnissen erlaubt sind, deren Elemente durch ein einzelnes Nullzeichen voneinander getrennt sind. Das Ende der kompletten Liste wird an einem doppelten `vbNullChar` erkannt.

- `pTo`

  Hier wird ein Pointer auf einen String übergeben, der die Zieldatei oder das Zielverzeichnis angibt. An dem String muss wie bei `pFrom` ein doppeltes `vbNullChar` angehängt sein.

- `fFlags`

  Ein `Integer`-Flagfeld, das einige Funktionalitäten mitbestimmt. Folgende Flags sind möglich und können auch kombiniert werden:

  ```
  Private Const FOF_ALLOWUNDO = &H40
  ```

  Wenn möglich, werden Undo-Informationen gesichert.

  ```
  Private Const FOF_FILESONLY = &H80
  ```

  Dateioperationen werden nur mit Wildcard-Dateinamen (*.*) durchgeführt.

  ```
  Private Const FOF_MULTIDESTFILES = &H1
  ```

  Für jede Quelldatei existieren mehrere Ziele im Element `pTo`.

  ```
  Private Const FOF_NOCONFIRMATION = &H10
  ```

  Alle Dialoge, die auftreten können, werden automatisch mit Ja bestätigt.

  ```
  Private Const FOF_NOCONFIRMMKDIR = &H200
  ```

  Es wird keine Bestätigung zum Erstellen von neuen Verzeichnissen verlangt.

  ```
  Private Const FOF_NOCOPYSECURITYATTRIBS = &H800
  ```

  Unter NT 4.71 werden die Sicherheitsattribute nicht mit kopiert.

  ```
  Private Const FOF_NOERRORUI = &H400
  ```

  Ist ein Fehler aufgetreten, wird kein Dialog zum Bestätigen angezeigt.

  ```
  Private Const FOF_RENAMEONCOLLISION = &H8
  ```

  Wenn beim Verschieben, Umbenennen oder Kopieren die Zieldatei bereits existiert, wird ein Dialog zum Umbenennen angezeigt.

  ```
  Private Const FOF_SILENT = &H4
  ```

  Ein Fortschrittsdialog wird nicht angezeigt.

  ```
  Private Const FOF_SIMPLEPROGRESS = &H100
  ```

  Es wird ein Fortschrittsdialog angezeigt, allerdings ohne Dateinamen.

  ```
  Private Const FOF_WANTMAPPINGHANDLE = &H20
  ```

  Zeigt an, dass das Element `hNameMappings` aktiv ist.

Im Allgemeinen benötigen Sie die nächsten Elemente nicht. Wenn Sie diese trotzdem benutzen möchten, gibt es ein Problem, das mit der Ausrichtung an Doppelwortgrenzen zusammenhängt. VB(A) richtet Teilvariablen in einem Typ an Doppelwortgrenzen aus. D.h., wenn das Element eines Typs vom Datentyp `Integer` mit 2 Bytes ist, beginnt das nächste Element nicht etwa nach 2 Bytes, sondern 4 Bytes weiter. Es werden also von VB(A) zwei Füllbytes eingefügt, wovon die API-Funktion nichts ahnt. Werten Sie unter VB die nächsten Elemente aus, bringen diese zwei Bytes alles durcheinander. Sie müssen also alles, was im Speicher hinter dem problematischen Element kommt, um zwei Bytes nach hinten schieben, damit wieder alles passt. Hier muss dann mit `CopyMemory` gearbeitet werden.

- `fAnyOperationsAborted`

  Ist dieser Parameter ungleich null, wurde die Aktion durch den Benutzer abgebrochen. Beachten Sie auch die Anmerkungen zur Doppelwortgrenze im Element `fFlags`.

- `hNameMappings`

  Dieses Element liefert einen Zeiger auf ein Array vom Typ `SHNAMEMAPPING`, das für jede verschobene, kopierte oder umbenannte Datei einen Eintrag enthält. Dieses Element wird nur ausgefüllt, wenn der Parameter `FOF_WANTMAPPINGHANDLE` im Flagfeld `fFlags` gesetzt ist. Stellen Sie sicher, dass das Handle mit `SHFreeNameMappings` geschlossen wird.

```
Private Type SHNAMEMAPPING
   pszOldPath As String
   pszNewPath As String
   cchOldPath As Long
   cchNewPath As Long
End Type
```

- `pszOldPath`

  Zeiger auf ein Bytearray (String) mit dem alten Pfadnamen.

- `pszNewPath`

  Zeiger auf ein Bytearray (String) mit dem neuen Pfadnamen.

- `cchOldPath`

  Anzahl der Zeichen im alten Pfadnamen.

- `cchNewPath`

  Anzahl der Zeichen im neuen Pfadnamen.

- `lpszProgressTitle`

  Ein String mit dem Titel der Fortschrittsanzeige, wenn im Flagfeld `fFlags` das Flag `FOF_SIMPLEPROGRESS` gesetzt ist.

Nachfolgend nun ein paar selbst geschriebene Funktionen, die mit der API-Funktion `SHFileOperation` arbeiten.

7.7.1 Verschieben und Kopieren

Mit der Funktion XXL_Copy (Listing 7.13) ist es möglich, Verzeichnisse samt deren Unterverzeichnisse in ein anderes zu kopieren oder zu verschieben. Dazu muss lediglich das Quellverzeichnis und das Zielverzeichnis als Parameter übergeben werden. Optional kann mit dem dritten Parameter angegeben werden, ob verschoben oder kopiert werden soll. Das funktioniert auch mit einzelnen Dateien.

Listing 7.13
Beispiele\07_Dateien\
07_01_Dateien.xls\mdlMKDelDir

```
Public Declare Function MakePath _
   Lib "imagehlp.dll" Alias "MakeSureDirectoryPathExists" ( _
   ByVal lpPath As String _
   ) As Long

Private Declare Function SHFileOperation _
   Lib "shell32.dll" Alias "SHFileOperationA" ( _
   lpFileOp As Any _
   ) As Long

Private Type SHFILEOPSTRUCT
    hwnd As Long
    wFunc As Long
    pFrom As String
    pTo As String
    fFlags As Integer
    fAnyOperationsAborted As Long
    hNameMappings As Long
    lpszProgressTitle As String
End Type

Private Const FO_DELETE = &H3
Private Const FO_MOVE = &H1
Private Const FO_RENAME = &H4
Private Const FO_COPY = &H2&
Private Const FOF_RENAMEONCOLLISION = &H8
Private Const FOF_NOCONFIRMMKDIR = &H200
Private Const FOF_NOCONFIRMATION = &H10
Private Const FOF_MULTIDESTFILES = &H1
Private Const FOF_ALLOWUNDO = &H40

'############################################################

Sub TestXXL_Kopie()
   Dim FF      As Long
   Dim strPath As String

   ' Erst einmal einen Pfad anlegen
   strPath = "c:\CopyDelNew\Copy\Copy"
   API_CreatePath strPath

   ' Zwei Dateien anlegen
   FF = FreeFile
   Open strPath & "\DATEI1" For Binary As FF: Close FF
   Open strPath & "\DATEI2" For Binary As FF: Close FF
```

Listing 7.13 (Forts.)
Beispiele\07_Dateien\
07_01_Dateien.xls\mdlMKDelDir

```
    ' Diese zwei Dateien in ein neu angelegtes
    ' Unterverzeichnis kopieren
    XXL_Copy strPath & "\*", strPath & "\Kopie\"

    ' Alle Dateien ab Quellverzeichnis in ein neu
    ' angelegtes Unterverzeichnis verschieben
    XXL_Copy strPath & "\*", strPath & "\Kopie1\", True

End Su

Public Function XXL_Copy( _
   ByVal strSourceFolder As String, _
   ByVal strDestinationFolder As String, _
   Optional blnMove As Boolean) As Boolean

   Dim udtXCopy As SHFILEOPSTRUCT

   ' Zeimal Nullchar anhängen, damit das Ende erkannt wird
   strSourceFolder = strSourceFolder & _
   vbNullChar & vbNullChar
   strDestinationFolder = strDestinationFolder & _
   vbNullChar & vbNullChar

   With udtXCopy

     If blnMove Then
       ' Verschieben
      .wFunc = FO_MOVE
     Else
       ' Kopieren
      .wFunc = FO_COPY
     End If

     ' Quelle und Ziel
     .pFrom = strSourceFolder
     .pTo = strDestinationFolder

     .fFlags = FOF_MULTIDESTFILES _
     Or FOF_NOCONFIRMATION _
     Or FOF_NOCONFIRMMKDIR _
     Or FOF_ALLOWUNDO

  End With

  ' Ausführen und Ergebnis zurückliefern
  If SHFileOperation(udtXCopy) = 0 _
  Then XXL_Copy = True

End Function
```

7.7.2 Löschen

Um auf einen Rutsch Verzeichnisse und deren Unterverzeichnisse samt Inhalt zu löschen und/oder in den Papierkorb zu schieben, nachfolgend ein Beispiellisting (Listing 7.14). Die Vorgehensweise und der Deklarationsabschnitt ist fast so, wie im vorherigen Abschnitt (7.7.1) zum Kopieren und Verschieben.

Listing 7.14
Beispiele\07_Dateien\
07_01_Dateien.xls\mdlMKDelDir

```
Sub TestXXL_Del()
  XXL_Del "c:\CopyDelNew\"
End Sub

Public Function XXL_Del(strSource As String) As Boolean
   Dim strSourceFolder        As String
   Dim strDestinationFolder   As String
   Dim strPath                As String
   Dim udtFileOP              As SHFILEOPSTRUCT

   ' Überprüfen, ob das zu löschende überhaupt existiert
   If Dir$(strSource, vbDirectory) = "" Then Exit Function

   XXL_Del = True

   ' Zeimal Nullchar anhängen, damit das Ende
   ' erkannt wird
   strSourceFolder = strSource _
   & vbNullChar & vbNullChar
   strDestinationFolder = vbNullString & _
   vbNullChar & vbNullChar

   With udtFileOP

      ' Löschen
      .wFunc = FO_DELETE

      ' Quelle und Ziel
      .pFrom = strSource
      .pTo = strDestinationFolder

      .fFlags = FOF_MULTIDESTFILES _
      Or FOF_NOCONFIRMATION _
      Or FOF_NOCONFIRMMKDIR _
      Or FOF_ALLOWUNDO ' In den Papierkorb

   End With

   ' Ausführen und Ergebnis zurückliefern
   If SHFileOperation(udtFileOP) <> 0 _
      Then XXL_Del = False

End Function
```

7.8 Lange und kurze Dateinamen

In früheren Zeiten, als DOS noch das ultimative Betriebssystem war, wurde die Länge von Dateinamen auf elf und die von Verzeichnissen auf acht Zeichen beschränkt. Diese Beschränkungen sind mittlerweile aufgehoben, aber aus Kompatibilitätsgründen wird bei moderneren Dateisystemen zu jedem langen Dateinamen zusätzlich noch ein kurzer Dateiname erzeugt und gespeichert. Dieser Name besteht aus den ersten sechs Buchstaben des langen Namens, anschließend folgt die Tilde ~ und danach wird eine Ziffer angehängt, die bei Übereinstimmung der ersten sechs Buchstaben schrittweise erhöht wird.

Die Größe der heutigen Festplatten verführt dazu, Verzeichnisse tief zu verschachteln. Zudem sind Verzeichnisnamen auch nicht mehr auf acht Zeichen beschränkt. So kann es also leicht vorkommen, dass die Pfade länger als 255 Zeichen werden. In solch einem Fall gibt es mit Excel beispielsweise Probleme beim Öffnen von Arbeitsmappen mit der Methode `Workbooks.Open`.

Abhilfe schaffen da kurze Dateipfade. Die Windows-API stellt die Funktion `GetShortPathName` bereit, die einen Pfad in die 8+3-Notation bringt. Ab Windows 98 gibt es auch die Funktion `GetLongPathName`, die aus einem kurzen Pfad einen langen Pfad macht.

Achtung

Es ist mit `GetShortPathName` nicht möglich, aus einem fiktiven Pfad zu einer Datei einen kurzen in der 8+3-Notation zu machen. Die Datei und der Pfad dorthin müssen existieren.

Beiden Funktionen wird als erstes Argument der Pfad übergeben, der umgewandelt werden soll. Als zweites Argument wird ein ausreichend dimensionierter Puffer benötigt, der den umgewandelten Pfad aufnimmt, und als drittes Argument wird die Länge des Puffers übergeben.

Das klappt auch ganz hervorragend. Leider sträubt sich die Funktion `GetShortPathName` bei Pfaden mit einer Größe von über 259 Zeichen. Aber gerade dafür werden ja die kurzen Pfade am meisten benötigt. Bei einem fehlgeschlagenen Versuch, den Pfad umzuwandeln, muss also etwas anders vorgegangen werden.

In diesem Fall benutzen Sie die API-Funktion `FindFirstFile`, die eine Struktur vom Typ `WIN32_FIND_DATA` mit den Dateiinformationen ausfüllt. Die Vorgehensweise dabei ist die, dass mit der Split-Funktion der Dateipfad in seine Bestandteile zerlegt wird, das Trennzeichen ist der Backslash (\). Von der Wurzel her wird anschließend der Pfad neu zusammengesetzt, aber die 8+3-Informationen in der `WIN32_FIND_DATA`-Struktur benutzt, die im Element `cAlternate` stecken. Selbstverständlich müssen Sie sich die Struktur vorher mit `FindFirstFile` auf den entsprechenden Teilpfad ausfüllen lassen.

Ähnlich müssen Sie vorgehen, wenn Sie mit Systemen arbeiten, die `GetLongPathName` nicht kennen. Dann müssten Sie aber das Element `cFileName` der `WIN32_FIND_DATA`-Struktur benutzen. In dem nachfolgenden Beispiel habe ich es aber gelassen und verwende dafür einfach nur die API-Funktion `GetLongPathName`.

Listing 7.15
Beispiele\07_Dateien\07_01_Dateien.xls\mdlShortLongPath

```
Private Const MAX_PATH = 260

Private Type FILETIME
        dwLowDateTime As Long
        dwHighDateTime As Long
End Type
```

Listing 7.15 (Forts.)
Beispiele\07_Dateien\07_01_Dateien.xls\mdlShortLongPath

```
Private Type WIN32_FIND_DATA
        dwFileAttributes As Long
        ftCreationTime As FILETIME
        ftLastAccessTime As FILETIME
        ftLastWriteTime As FILETIME
        nFileSizeHigh As Long
        nFileSizeLow As Long
        dwReserved0 As Long
        dwReserved1 As Long
        cFileName As String * MAX_PATH
        cAlternate As String * 14
End Type

Private Declare Function FindFirstFile _
   Lib "kernel32" Alias "FindFirstFileA" ( _
   ByVal lpFileName As String, _
   lpFindFileData As WIN32_FIND_DATA _
   ) As Long

Private Declare Function GetLongPathName _
   Lib "kernel32" Alias "GetLongPathNameA" ( _
   ByVal lpszShortPath As String, _
   ByVal lpszLongPath As String, _
   ByVal cchBuffer As Long _
   ) As Long

Private Declare Function GetShortPathName _
   Lib "kernel32" Alias "GetShortPathNameA" _
   (ByVal lpszLongPath As String, _
   ByVal lpszShortPath As String, _
   ByVal lBuffer As Long) As Long

Private Sub testShortLongName()
   Dim strPathShort  As String
   Dim strPathLong   As String

   strPathShort = GetShortPath("C:\Dokumente und Einstellungen" & _
      "\All Users\")

   strPathLong = GetLongPath(strPathShort)

   MsgBox strPathShort & vbCrLf & strPathLong

End Sub

Public Function GetShortPath(ByVal strFilePath As String) As String
   Dim strBuffer  As String
   Dim strPath    As String
   Dim lngLength  As Long
   Dim varSplit   As Variant
   Dim udtFind    As WIN32_FIND_DATA
   Dim i          As Long

   On Error Resume Next

   ' Puffer anlegen
   strBuffer = String(260, 0)
```

Listing 7.15 (Forts.)
Beispiele\07_Dateien\
07_01_Dateien.xls\
mdlShortLongPath

```
    ' Kurzen Namen holen
    lngLength = GetShortPathName(strFilePath, strBuffer, 259)

    ' Name zurückliefern
    If lngLength Then
       GetShortPath = Left(strBuffer, lngLength)
       Exit Function
    Else
       ' In ein Array einzelner Namen umwandeln
       varSplit = Split(strFilePath, "\")

       ' Laufwerksbezeichnung
       strPath = varSplit(0)

       ' Alle Elemente durchlaufen
       For i = 1 To UBound(varSplit)
          strBuffer = strPath & "\" & varSplit(i)

          If FindFirstFile(strBuffer, udtFind) = 0 Then
             GetShortPath = strFilePath
             Exit Function
          End If

          'Kurzen Namen extrahieren
          strBuffer = udtFind.cAlternate
          strBuffer = Trim(Left(strBuffer, _
             InStr(1, strBuffer, Chr(0)) - 1))

          ' Zusammensetzen
          strPath = strPath & "\" & strBuffer

       Next
    End If
    GetShortPath = strPath

End Function

Public Function GetLongPath(ByVal strFilePath As String) As String
    Dim strBuffer  As String
    Dim lngLength  As Long

    On Error Resume Next

    ' Puffer anlegen
    strBuffer = String(1024, 0)

    ' Langen Namen holen
    lngLength = GetLongPathName(strFilePath, strBuffer, 1023)

    ' Name zurückliefern
    If lngLength Then
       GetLongPath = Left(strBuffer, lngLength)
    Else
       GetLongPath = strFilePath
    End If
End Function
```

7.9 Sonderverzeichnisse

Um an Standardverzeichnisse wie zum Beispiel das *Temp*- oder Programmverzeichnis zu kommen, können Sie die API-Funktion `SHGetSpecialFolderPath` bemühen (Listing 7.16). Die zwei für den praktischen Gebrauch wichtigsten Parameter dieser Funktion sind `lpszPath` und `lngFolder`. Der Puffer `lpszPath` nimmt den gewünschten Pfad auf und der Longwert `lngFolder` gibt an, welcher Pfad geliefert werden soll.

Listing 7.16
Beispiele\07_Dateien\07_01_Dateien.xls\WorkSheets (»Sonderverzeichnisse«)

```
Private Declare Function SHGetSpecialFolderPath _
   Lib "shell32.dll" Alias "SHGetSpecialFolderPathA" ( _
   ByVal hwndOwner As Long, _
   ByVal lpszPath As String, _
   ByVal lngFolder As Long, _
   ByVal fCreate As Long _
   ) As Long

Private Const CSIDL_COMMON_DESKTOPDIRECTORY = &H19
Private Const CSIDL_COMMON_DOCUMENTS = &H2E
Private Const CSIDL_COMMON_FAVORITES = &H1F
Private Const CSIDL_COMMON_PROGRAMS = &H17
Private Const CSIDL_COMMON_STARTMENU = &H16
Private Const CSIDL_COMMON_STARTUP = &H18
Private Const CSIDL_COMMON_TEMPLATES = &H2D
Private Const CSIDL_PERSONAL = &H5
Private Const CSIDL_DESKTOP = &H0
Private Const CSIDL_DESKTOPDIRECTORY = &H10
Private Const CSIDL_FAVORITES = &H6
Private Const CSIDL_FONTS = &H14
Private Const CSIDL_PROGRAM_FILES = &H26
Private Const CSIDL_PROGRAM_FILES_COMMON = &H2B
Private Const CSIDL_SENDTO = &H9
Private Const CSIDL_STARTMENU = &HB
Private Const CSIDL_STARTUP = &H7
Private Const CSIDL_SYSTEM = &H25
Private Const CSIDL_WINDOWS = &H24

Function Spezialverzeichnis(lngWant As Long) As String
   Dim strBuff As String

   strBuff = String(256, 0)
   SHGetSpecialFolderPath 0&, strBuff, lngWant, 0&

   Spezialverzeichnis = Left(strBuff, InStr(1, strBuff, Chr(0)) - 1)

End Function

Private Sub cmdSpecial_Click()
Me.Range("A3") = "DESKTOPDIRECTORY"
Me.Range("B3") = Spezialverzeichnis(CSIDL_COMMON_DESKTOPDIRECTORY)

Me.Range("A4") = "DOCUMENTS"
Me.Range("B4") = Spezialverzeichnis(CSIDL_COMMON_DOCUMENTS)
```

Listing 7.16 (Forts.)
Beispiele\07_Dateien\
07_01_Dateien.xls\WorkSheets
(»Sonderverzeichnisse«)

```
Me.Range("A5") = "FAVORITES"
Me.Range("B5") = Spezialverzeichnis(CSIDL_COMMON_FAVORITES)

Me.Range("A6") = "PROGRAMS"
Me.Range("B6") = Spezialverzeichnis(CSIDL_COMMON_PROGRAMS)

Me.Range("A7") = "STARTMENU"
Me.Range("B7") = Spezialverzeichnis(CSIDL_COMMON_STARTMENU)

Me.Range("A8") = "STARTUP"
Me.Range("B8") = Spezialverzeichnis(CSIDL_COMMON_STARTUP)

Me.Range("A9") = "TEMPLATES"
Me.Range("B9") = Spezialverzeichnis(CSIDL_COMMON_TEMPLATES)

Me.Range("A10") = "PERSONAL"
Me.Range("B10") = Spezialverzeichnis(CSIDL_PERSONAL)

Me.Range("A11") = "DESKTOP"
Me.Range("B11") = Spezialverzeichnis(CSIDL_DESKTOP)

Me.Range("A12") = "DESKTOPDIRECTORY"
Me.Range("B12") = Spezialverzeichnis( _
   CSIDL_DESKTOPDIRECTORY)

Me.Range("A13") = "FAVORITES"
Me.Range("B13") = Spezialverzeichnis(CSIDL_FAVORITES)

Me.Range("A14") = "FONTS"
Me.Range("B14") = Spezialverzeichnis(CSIDL_FONTS)

Me.Range("A15") = "PROGRAM_FILES"
Me.Range("B15") = Spezialverzeichnis(CSIDL_PROGRAM_FILES)

Me.Range("A16") = "PROGRAM_FILES_COMMON"
Me.Range("B16") = Spezialverzeichnis( _
   CSIDL_PROGRAM_FILES_COMMON)

Me.Range("A17") = "SENDTO"
Me.Range("B17") = Spezialverzeichnis(CSIDL_SENDTO)

Me.Range("A18") = "STARTMENU"
Me.Range("B18") = Spezialverzeichnis(CSIDL_STARTMENU)

Me.Range("A19") = "STARTUP"
Me.Range("B19") = Spezialverzeichnis(CSIDL_STARTUP)

Me.Range("A20") = "SYSTEM"
Me.Range("B20") = Spezialverzeichnis(CSIDL_SYSTEM)

End Sub
```

7.10 Dateien kürzen

Um beispielsweise Textdateien an einer bestimmten Textstelle zu kürzen, müssen Sie normalerweise diese Datei öffnen, den Text auslesen, die Datei schließen, löschen, eine neue mit gleichem Namen anlegen und gleichzeitig öffnen, den gekürzten Text hineinschreiben und diese schließen.

Um das näher zu erläutern, ein kleines Beispiel:

Nehmen wir an, Sie führen eine Logdatei, die jeweils die letzten 30 Ereignisse enthält. Um einen neuen Eintrag hinzuzufügen, öffnen Sie diese Datei und lesen den Dateiinhalt in eine Stringvariable ein. Nun schließen Sie die Zeile, die hinzugefügt werden soll, mit einem Wagenrücklauf und Zeilenvorschub (`vbCrLf`) ab und hängen den ausgelesenen Dateiinhalt daran, damit der letzte Eintrag am Anfang ist.

Die Position des ältesten Eintrages finden Sie, indem Sie mit `InStrRev` die Position des letzten Auftretens von `VbCrLf` suchen. An dieser Stelle wird der String gekürzt und in die Datei zurück geschrieben. Ist nun die ursprüngliche Datei länger als der zurück geschriebene Text inklusive der hinzugefügten Zeile, wird die Datei nicht etwa gekürzt, die ursprüngliche Länge bleibt erhalten.

Um solche Effekte zu vermeiden, müssen Sie die Logdatei erst löschen und mit dem neuen Text neu anlegen. Das hat den zusätzlichen Nachteil, dass die Datei jedes mal ein anderes Erstellungsdatum erhält.

Das ist natürlich extrem umständlich, deshalb werden dazu im nachfolgenden Beispiel (Listing 7.17) einfach ein paar API-Funktionen benutzt.

Listing 7.17
Beispiele\07_Dateien\
07_01_Dateien.xls\mdlCut

```
Private Declare Function CreateFile _
   Lib "kernel32" Alias "CreateFileA" ( _
   ByVal lpFileName As String, _
   ByVal dwDesiredAccess As Long, _
   ByVal dwShareMode As Long, _
   ByVal lpSecurityAttributes As Long, _
   ByVal dwCreationDisposition As Long, _
   ByVal dwFlagsAndAttributes As Long, _
   ByVal hTemplateFile As Long _
   ) As Long

Private Declare Function SetFilePointer _
   Lib "kernel32" ( _
   ByVal hFile As Long, _
   ByVal lDistanceToMove As Long, _
   lpDistanceToMoveHigh As Long, _
   ByVal dwMoveMethod As Long _
   ) As Long

Private Declare Function SetEndOfFile _
   Lib "kernel32" ( _
   ByVal hFile As Long _
   ) As Long

Private Declare Function CloseHandle _
   Lib "kernel32" ( _
   ByVal hObject As Long _
   ) As Long
```

Listing 7.17 (Forts.)
Beispiele\07_Dateien\
07_01_Dateien.xls\mdlCut

```
Private Declare Function GetTempPath _
   Lib "kernel32" Alias "GetTempPathA" ( _
   ByVal nBufferLength As Long, _
   ByVal lpBuffer As String _
   ) As Long

Private Declare Function ShellExecute Lib "shell32.dll" _
   Alias "ShellExecuteA" ( _
   ByVal Fensterzugriffsnummer As Long, _
   ByVal lpOperation_wie_Open_oder_Print As String, _
   ByVal lpDateiname_incl_Pfad As String, _
   ByVal lpZusätzliche_Startparameter As String, _
   ByVal lpArbeitsverzeichnis As String, _
   ByVal nGewünschte_Fenstergröße_der_Anwendung As Long) _
   As Long

Private Const SW_SHOWNORMAL = 1

Private Const GENERIC_WRITE = &H40000000
Private Const GENERIC_READ = &H80000000
Private Const OPEN_EXISTING = 3
Private Const FILE_SHARE_WRITE = &H2
Private Const FILE_SHARE_READ = &H1
Private Const FILE_BEGIN = 0
Private Const FILE_CURRENT = 1
Private Const FILE_END = 2


Public Sub testCut()
   Dim lngFile       As Long
   Dim lngFF         As Long
   Dim strFilename   As String
   Dim strPath       As String
   Dim lngRet        As Long
   Dim lngAccess     As Long
   Dim lngShare      As Long
   Dim lngFileLen    As Long
   Dim strBuffer     As String
   Dim i             As Long

   ' Temp Pfad ermitteln
   strPath = String(255, 0)
   lngRet = GetTempPath(255, strPath)

   ' Pfad und Name zusammenfügen
   strFilename = Left(strPath, lngRet) & "~~Kürzen.txt"

   ' Ev. vorhandene Datei löschen
   If Dir(strFilename) <> "" Then Kill strFilename

   ' Freie Dateinummer holen
   lngFF = FreeFile

   ' Logdaten erzeugen
   For i = 1 To 30
      strBuffer = strBuffer & Format(Now() - i, "DDD DD.MM.YYYY") & _
      " Text: " & String(20, "A") & vbCrLf
   Next
```

Listing 7.17 (Forts.)
Beispiele\07_Dateien\
07_01_Dateien.xls\mdlCut

```
    strBuffer = Left(strBuffer, Len(strBuffer) - 2)

    ' Log Datei mit 30 Zeilen erzeugen
    Open strFilename For Binary Access Write As lngFF
       Put #lngFF, , strBuffer
    Close

    ' Neuen Eintrag in Logdatei hinzufügen,
    ' letzten Eintrag löschen
    Open strFilename For Binary Access Read Write As lngFF

       strBuffer = String(LOF(lngFF), " ")

       ' Logdatei auslesen
       Get #lngFF, , strBuffer

       ' Neuen Eintrag hinzu
       strBuffer = Format(Now(), "DDD DD.MM.YYYY") & _
          " Text: " & String(2, "A") & _
          vbCrLf & strBuffer

       ' Position des letzten Eintrages finden und kürzen
       strBuffer = Left(strBuffer, InStrRev(strBuffer, vbCrLf))

       ' Die Datei wird hierbei nicht gekürzt, wenn es
       ' nötig ist, Die Länge der Datei bleibt erhalten,
       ' oder wird erhöht
       Put #lngFF, 1, strBuffer

    Close

    lngAccess = GENERIC_WRITE ' Or GENERIC_READ

    lngShare = FILE_SHARE_WRITE ' Or FILE_SHARE_READ

    lngFileLen = Len(strBuffer) ' Neue Länge

    ' Filehandle holen
    lngFile = CreateFile(strFilename, lngAccess, _
        lngShare, ByVal 0&, OPEN_EXISTING, 0&, 0&)

    ' Position setzen
    lngRet = SetFilePointer(lngFile, lngFileLen, 0&, FILE_BEGIN)

    ' EOF setzen
    lngRet = SetEndOfFile(lngFile)

    ' Filehandle schließen
    CloseHandle lngFile

    ' Datei anzeigen
    ShellExecute 0&, "open", strFilename, vbNullString, _
       vbNullString, SW_SHOWNORMAL

End Sub
```

Der größte Teil des Codes besteht darin, eine Beispieldatei mit 30 Zeilen anzulegen und anschließend einen neuen Eintrag hinzuzufügen.

Um diese erzeugte Textdatei zu kürzen, holt man sich mit der API `CreateFile` ein Handle auf die Datei, setzt mit der API `SetFilePointer` den Pointer an die gewünschte Stelle in der Datei und setzt mit der API `SetEndOfFile` das Dateiende. Mit `CloseHandle` wird anschließend die Datei geschlossen.

Um das zu überprüfen, wird die Textdatei mit dem verknüpften Programm geöffnet. Dazu wird `ShellExecute` benutzt.

Und so sieht der Dateiinhalt aus, wenn man das Programm direkt nach dem Zurückschreiben beendet und die Datei nicht kürzt:

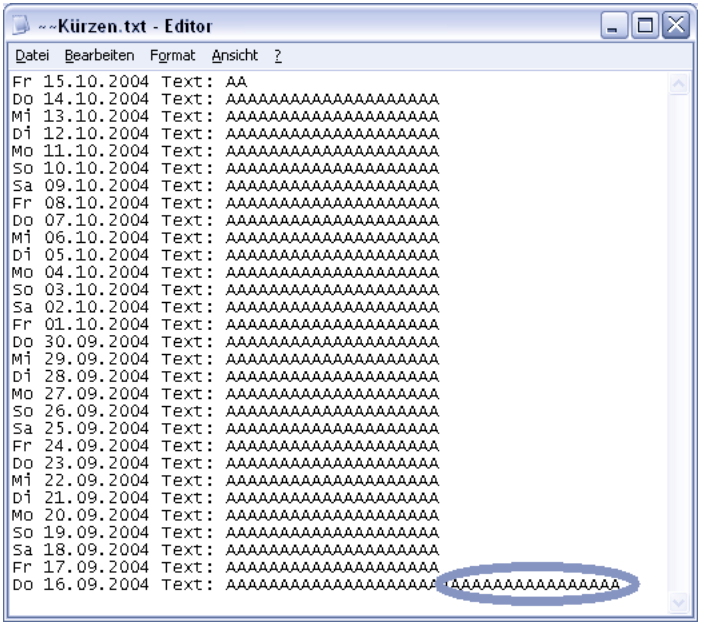


Abbildung 7.3
Dateiinhalt ohne Kürzen

8

Laufwerke

8.1 Was Sie in diesem Kapitel erwartet

In diesem Kapitel wird gezeigt, wie Sie freie Laufwerksbuchstaben, den Typ vorhandener Laufwerke, deren Speicherkapazität, den freien Speicher, die Datenträgerbezeichnung, das Filesystem und die Seriennummer ermitteln können.

Außerdem wird dargestellt, wie Netzlaufwerke programmgesteuert verbunden und wieder getrennt werden können.

8.2 Informationen über Laufwerke

Um an Informationen über verfügbare Laufwerke zu kommen, können Sie auf Funktionalitäten des `FileSystemObjects` zurückgreifen. Wie ich im Verlaufe dieses Buches aber schon mehrfach erwähnt habe, begibt man sich dabei in die Abhängigkeit eines fremden Objektes. Das kann zur Folge haben, dass Ihre, bei Ihnen einwandfrei laufende Mappe auf anderen Rechnern nicht funktioniert. Der Vorteil gegenüber anderen Lösungen ist der, dass der Code sehr kompakt wird, da die eigentliche Funktionalität verborgen wird.

Bei einer Lösung mit der Windows-API müssen Sie etwas mehr Aufwand betreiben. Es sind erst einmal die Deklarationsanweisungen, die einiges an Platz erfordern. Weiterhin müssen Sie sich die eingesetzten Konstanten selbst erstellen, was auch ein paar Zeilen zusätzlichen Codes bedeutet.

Der eigentliche Code, der die Arbeit erledigt, ist aber nicht sehr kompliziert und nimmt auch nicht viel mehr Platz in Anspruch als der bei Benutzung des FSO. Wenn Sie dann noch die Funktionalität in eine Klasse auslagern, verfügen Sie nachher über ein kompaktes Objekt, dass Sie oder andere immer wieder benutzen können, ohne mit den API-Funktionen in Berührung zu kommen. Etwas anderes macht das FSO übrigens auch nicht, es werden lediglich die gleichen API-Funktionen gekapselt.

8.2.1 FileSystemObject

Die Bibliothek Scripting Runtime, die sich in der DLL `ScrRun.Dll` verbirgt, beinhaltet das `FileSystemObject`. Um dieses zu nutzen, muss diese *.DLL* auch auf Ihrem System verfügbar sein. Um die Typen und Konstanten des FSO zu verwenden, benötigen Sie einen Verweis auf die `Microsoft Scripting Runtime`.

Das FSO beinhaltet eine `Drives`-Auflistung mit Objekten vom Typ `Drive`, wobei jedes Element dieser Auflistung ein eigenes Laufwerk repräsentiert. Das `Drive`-Objekt besitzt Eigenschaften, auf die Sie zugreifen können und die die Eigenschaften des jeweiligen Laufwerkes enthalten.

Nachfolgend ein Beispiel (Listing 8.1), das nacheinander die Eigenschaften jedes Laufwerkes ausgibt:

Listing 8.1
Beispiele\08_Dateien\
08_01_Laufwerke.xls\mdlFSO

```
Public Sub DrivesFSO()
   Dim objFSO     As New FileSystemObject
   Dim objDrive   As Drive
   Dim strDrives  As String

   On Error Resume Next

   For Each objDrive In objFSO.Drives
      With objDrive

         ' Laufwerksbuchstabe
         strDrives = "LW="
         strDrives = strDrives & .DriveLetter

         ' Laufwerkstyp
         strDrives = strDrives & vbCrLf & "Typ="
         Select Case .DriveType
           Case 0
               strDrives = strDrives & "Laufwerkstyp unbekannt"
           Case 1
               strDrives = strDrives & "Wechseldatenträger"
           Case 2
               strDrives = strDrives & "Festplatte"
           Case 3
               strDrives = strDrives & "Netzlaufwerk ("
               strDrives = strDrives & .ShareName & ")"
            Case 4
               strDrives = strDrives & "CD-Laufwerk"
           Case 5
             strDrives = strDrives & "RAM-Disk"
         End Select

         ' Filesystem FAT, FAT32, NTFS
         strDrives = strDrives & vbCrLf & "File System="
         strDrives = strDrives & .FileSystem

         ' Gesamter Speicher eines Datenträgers
         strDrives = strDrives & vbCrLf & "Gesamtgröße="
         strDrives = strDrives & .TotalSize
```

Listing 8.1 (Forts.)
Beispiele\08_Dateien\
08_01_Laufwerke.xls\mdlFSO

```
            ' Verfügbarer Speicher eines Datenträgers
            strDrives = strDrives & vbCrLf & "Verfügbar="
            strDrives = strDrives & .AvailableSpace

            ' Bezeichnung eines Datenträgers
            ' z.B. Festplattenlaufwerk_1
            strDrives = strDrives & vbCrLf & "Datenträgerbezeichnung="
            strDrives = strDrives & .VolumeName

            ' Seriennummer eines Datenträgers
            strDrives = strDrives & vbCrLf & "Seriennummer="
            strDrives = strDrives & .Serialnumber

            MsgBox strDrives

        End With
    Next
End Sub
```

8.2.2 API

Die Windows-API bietet mit den Funktionen `GetVolumeInformation`, `GetDiskFreeSpaceEx` und `GetLogicalDrives` Möglichkeiten, die relevanten Informationen über installierte Laufwerke und deren Eigenschaften zu bekommen.

GetVolumeInformation

```
Private Declare Function GetVolumeInformation _
    Lib "kernel32" Alias "GetVolumeInformationA" ( _
    ByVal lpRootPathName As String, _
    ByVal lpVolumeNameBuffer As String, _
    ByVal nVolumeNameSize As Long, _
    lpVolumeSerialNumber As Long, _
    lpMaximumComponentLenght As Long, _
    lpFileSystemFlags As Long, _
    ByVal lpFileSystemNameBuffer As String, _
    ByVal nFileSystemNameSize As Long _
    ) As Long
```

Diese Funktion liefert Informationen über ein Laufwerk. Als Ergebnis liefert sie einen Longwert zurück, der ungleich null ist, wenn die Funktion erfolgreich war.

- `lpRootPathName`

  Mit dem Parameter `lpRootPathName` übergeben Sie einen String mit dem Laufwerksbuchstaben, z.B. X:\.

  Wenn es sich dabei um einen UNC-Namen in der Form \\Server\Freigabename\... handelt, übergeben Sie dazu den String bis einschließlich den abschließenden Backslash hinter dem Freigabenamen.

- `lpVolumeNameBuffer`

  Der Parameter `lpVolumeNameBuffer` muss ein String sein, der groß genug ist, um die Laufwerksbezeichnung aufzunehmen.

- `nVolumeNameSize`

  Der Parameter `nVolumeNameSize` ist ein Long mit der Größe des Buffers, der für die Laufwerksbezeichnung übergeben wird.

- `lpVolumeSerialNumber`

  Der Parameter `lpVolumeSerialNumber` ist ein Longwert, der die Seriennummer aufnimmt.

- `lpMaximumComponentLength`

  Der Parameter `lpMaximumComponentLength` ist ein Longwert, der die maximale Länge eines Pfades zurückgibt, die abhängig vom Filesystem ist.

- `lpVolumeNameBuffer`

  Der Parameter `lpVolumeNameBuffer` muss ein Stringpuffer sein, der groß genug ist, um die Laufwerksbezeichnung aufzunehmen.

- `nVolumeNameSize`

  Der Parameter `nVolumeNameSize` ist ein `Long` mit der Größe des Buffers für die Laufwerksbezeichnung.

GetDiskFreeSpaceEx

```
Private Declare Function GetDiskFreeSpaceEx _
   Lib "kernel32" Alias "GetDiskFreeSpaceExA" ( _
   ByVal lpRootPathName As String, _
   lpFreeBytesAvailableToCaller As Currency, _
   lpTotalNumberOfBytes As Currency, _
   lpTotalNumberOfFreeBytes As Currency _
   ) As Long
```

Die API-Funktion `GetDiskFreeSpaceEx` liefert Speicherinformationen der Laufwerke. Als Ergebnis liefert diese Funktion einen Longwert zurück, der ungleich null ist, wenn die Funktion erfolgreich war.

- `lpRootPathName`

  Der Parameter `lpRootPathName` muss ein String sein, der ein Verzeichnis auf dem Laufwerk enthält, z.B. *C:\Windows*. Ist dieser Parameter `vbNullString`, so wird das Laufwerk mit dem aktuellen Verzeichnis angenommen.

- `lpFreeBytesAvailableToCaller`

  Der Parameter `lpFreeBytesAvailableToCaller` muss ein `Currency` sein, der die Größe des verfügbaren Speicherplatzes für den aktuellen Benutzer aufnimmt. Der Typ in der API ist eigentlich eine 64-Bit-Ganzzahl. Das gibt es in VBA aber nicht. Dafür gibt es in VBA den Datentyp `Currency`, der Zahlen als 64-Bit-Ganzzahl speichert. Der Wert selber ist die gespeicherte Ganzzahl, dividiert durch 10.000, um Nachkommastellen zuzulassen. Daher muss der zurückgelieferte Wert mit 10.000 multipliziert werden.

- `lpTotalNumberOfBytes`

  Der Parameter `lpTotalNumberOfBytes` muss ein `Currency` sein, der die Größe des gesamten Speicherplatzes aufnimmt. Beim Typ gilt das Gleiche wie beim Parameter `lpFreeBytesAvailableToCaller`.

- `lpTotalNumberOfFreeBytes`

  Der Parameter `lpTotalNumberOfFreeBytes` muss ein `Currency` sein, der die Größe des gesamten freien Speicherplatzes aufnimmt. Beim Typ gilt das Gleiche wie beim Parameter `lpFreeBytesAvailableToCaller`.

GetLogicalDrives

```
Private Declare Function GetLogicalDrives Lib "kernel32" () As Long
```

Die API-Funktion `GetLogicalDrives` liefert einen Longwert. Dieser Longwert enthält die Information, welche Laufwerke vorhanden sind. Für jedes gesetzte der verfügbaren 32 Bit existiert ein Laufwerk.

Ist Bit 0 gesetzt, bedeutet das, dass ein Laufwerk A vorhanden ist, ist Bit 25 gesetzt, existiert ein Laufwerk mit dem Buchstaben Z.

WNetGetConnection

```
Private Declare Function WNetGetConnection _
   Lib "mpr.dll" Alias "WNetGetConnectionA" ( _
   ByVal lpszLocalName As String, _
   ByVal lpszRemoteName As String, _
   cbRemoteName As Long _
   ) As Long
```

Diese Funktion liefert den Namen der Netzwerkressource. Als Ergebnis liefert diese Funktion einen Longwert zurück, der gleich null ist, wenn die Funktion erfolgreich war.

- Rückgabewerte

  ```
  Private Const ERROR_BAD_DEVICE = 1200&
  ```

  Ungültiger Parameter `lpLocalName`.

  ```
  Private Const ERROR_CONNECTION_UNAVAIL = 1201&
  ```

  Das normalerweise verbundene Gerät ist momentan nicht verbunden.

  ```
  Private Const ERROR_EXTENDED_ERROR = 1208&
  ```

  Ein netzwerkspezifischer Fehler ist aufgetreten. Für Einzelheiten muss die Funktion `WnetEnumResource` benutzt werden.

  ```
  Private Const ERROR_MORE_DATA = 234
  ```

  Der Puffer `lpRemoteName` ist zu klein. Der Parameter `lpnLength` enthält jetzt die benötigte Länge.

  ```
  Private Const ERROR_NO_NET_OR_BAD_PATH = 1203&
  ```

  Kein Provider kennt die Verbindung.

  ```
  Private Const ERROR_NO_NETWORK = 1222&
  ```

  Das Netzwerk ist nicht verfügbar.

  ```
  Private Const ERROR_NOT_CONNECTED = 2250&
  ```

  Das im Parameter `lpLocalName` Gerät ist keine Verbindung.

- `lpszLocalName`

  Der Parameter `lpszLocalName` enthält einen String mit dem Laufwerksbuchstaben, z.B. X:\.

- `lpszRemoteName`

  Der Parameter `lpszRemoteName` ist ein Puffer als String in der benötigten Länge.

- `cbRemoteName`

  Der Parameter `cbRemoteName` gibt die Länge des Puffers an.

GetDriveType

```
Private Declare Function GetDriveType _
   Lib "kernel32" Alias "GetDriveTypeA" ( _
   ByVal nDrive As String _
   ) As Long
```

Diese Funktion liefert den Typ des Laufwerks zurück, und zwar einen der folgenden Werte:

```
Private Const DRIVE_CDROM = 5
Private Const DRIVE_FIXED = 3
Private Const DRIVE_RAMDISK = 6
Private Const DRIVE_REMOTE = 4
Private Const DRIVE_REMOVABLE = 2
```

Der Parameter `nDrive` muss ein String sein, der den Laufwerksbuchstaben enthält, z.B. »A:\«.

Nachfolgend ein Beispiel (Listing 8.2), das nacheinander die Eigenschaften jedes Laufwerkes ausgibt:

Listing 8.2
Beispiele\08_Dateien\
08_01_Laufwerke.xls\mdlAPI

```
Private Declare Function WNetGetConnection _
   Lib "mpr.dll" Alias "WNetGetConnectionA" ( _
   ByVal lpszLocalName As String, _
   ByVal lpszRemoteName As String, _
   cbRemoteName As Long _
   ) As Long

Private Declare Function GetLogicalDrives _
   Lib "kernel32" () As Long

Private Declare Function GetDriveType _
   Lib "kernel32" Alias "GetDriveTypeA" ( _
   ByVal nDrive As String _
   ) As Long

Private Declare Function GetVolumeInformation _
   Lib "kernel32" Alias "GetVolumeInformationA" ( _
   ByVal lpRootPathName As String, _
   ByVal lpVolumeNameBuffer As String, _
   ByVal nVolumeNameSize As Long, _
   lpVolumeSerialNumber As Long, _
   lpMaximumComponentLength As Long, _
   lpFileSystemFlags As Long, _
```

Listing 8.2 (Forts.)
Beispiele\08_Dateien\
08_01_Laufwerke.xls\mdlAPI

```
    ByVal lpFileSystemNameBuffer As String, _
    ByVal nFileSystemNameSize As Long _
    ) As Long

Private Declare Function GetDiskFreeSpaceEx _
    Lib "kernel32" Alias "GetDiskFreeSpaceExA" ( _
    ByVal lpRootPathName As String, _
    lpFreeBytesAvailableToCaller As Currency, _
    lpTotalNumberOfBytes As Currency, _
    lpTotalNumberOfFreeBytes As Currency _
    ) As Long

Private Const DRIVE_CDROM = 5
Private Const DRIVE_FIXED = 3
Private Const DRIVE_RAMDISK = 6
Private Const DRIVE_REMOTE = 4
Private Const DRIVE_REMOVABLE = 2

Public Sub DrivesAPI()
    Dim i                   As Long
    Dim strLW               As String
    Dim lngLW               As Long
    Dim strName             As String
    Dim lngSerial           As Long
    Dim curFreeSpace        As Currency
    Dim curAvailableSpace   As Currency
    Dim curTotalSize        As Currency
    Dim lngFlags            As Long
    Dim strFilesystem       As String
    Dim strDrives  As String

    'Jedes gesetzte Bit von lngLW ist
    'ein Laufwerk (32 Bit)
    lngLW = GetLogicalDrives()

    For i = 65 To 90
        'Wenn Bit gesetzt ist, ist entsprechender
        'Laufwerksbuchstabe vorhanden
        If lngLW And 2 ^ (i - 65) Then

          'Buffer init, Variablen leeren
          strName = String(256, 0)
          strFilesystem = String(256, 0)
          lngSerial = 0
          strLW = ""
          lngFlags = 0
          curAvailableSpace = 0
          curTotalSize = 0
          curFreeSpace = 0
          strLW = Chr(i) & ":\"

          ' Laufwerksbuchstabe
          strDrives = "LW="
          strDrives = strDrives & Chr(i)

          ' Laufwerkstyp
          strDrives = strDrives & vbCrLf & "Typ="
```

Listing 8.2 (Forts.)
Beispiele\08_Dateien\
08_01_Laufwerke.xls\mdlAPI

```
            Select Case GetDriveType(strLW)
               'Laufwerkstyp
               Case DRIVE_CDROM
                  strDrives = strDrives & "CD-Rom"
               Case DRIVE_FIXED
                  strDrives = strDrives & "Festplatte"
               Case DRIVE_RAMDISK
                  strDrives = strDrives & "Ramdisk"
               Case DRIVE_REMOTE
                  strDrives = strDrives & "Netzlaufwerk ("
                  'UNC-Pfad holen, weil Netzlaufwerk
                  strDrives = strDrives & PfadNachUnc(strLW) & ")"
               Case DRIVE_REMOVABLE
                  strDrives = strDrives & "Wechsellaufwerk"
               Case Else
                  strDrives = strDrives & "Andere"
            End Select

            'Laufwerksinfos holen
            GetVolumeInformation strLW, strName, 255, _
               lngSerial, 0, lngFlags, strFilesystem, 255

            ' Filesystem FAT, FAT32, NTFS
            strDrives = strDrives & vbCrLf & "File System="
            strDrives = strDrives & Left$(strFilesystem, InStr(1, _
               strFilesystem, Chr(0)) - 1)

            ' Laufwerksinformationen holen
            GetDiskFreeSpaceEx strLW, curAvailableSpace, _
                curTotalSize, curFreeSpace

            ' Gesamter Speicher eines Datenträgers
            strDrives = strDrives & vbCrLf & "Gesamtgröße="
            strDrives = strDrives & curTotalSize * 10000

            ' Verfügbarer Speicher eines Datenträgers
            strDrives = strDrives & vbCrLf & "Verfügbar="
            strDrives = strDrives & curAvailableSpace * 10000

            ' Freier Speicher eines Datenträgers
            strDrives = strDrives & vbCrLf & "Frei="
            strDrives = strDrives & curFreeSpace * 10000

            ' Bezeichnung eines Datenträgers
            ' z.B. Festplattenlaufwerk_1
            strDrives = strDrives & vbCrLf & "Name="
            strDrives = strDrives & _
               Left$(strName, InStr(1, strName, Chr(0)) - 1)

            ' Seriennummer eines Datenträgers
            strDrives = strDrives & vbCrLf & "Seriennummer="
            strDrives = strDrives & lngSerial

            ' Ausgabe
            MsgBox strDrives
         End If
      Next
   End Sub
```

Listing 8.2 (Forts.)
Beispiele\08_Dateien\
08_01_Laufwerke.xls\mdlAPI

```
Private Function PfadNachUnc(ByVal strPathname As String) As String
   Dim varDummy   As Variant
   Dim strUNC     As String
   Dim strLW      As String
   Dim strPath    As String

   On Error GoTo fehlerbehandlung

   'Es wird nur der Buchstabe und der Doppelpunkt gebraucht
   strLW = Left(strPathname, 2)

   'Der Rest wird zwischengespeichert und später angehängt
   strPath = Right(strPathname, Len(strPathname) - 2)

   'Nur wenn strLW
   If InStr(1, strLW, ":") = 2 Then

       'Buffer bereitstellen
       strUNC = String(1001, 0)

       'UNC-Pfad holen
       varDummy = WNetGetConnection(strLW, strUNC, 1000)

       'Funktion auf Erfolg testen
       If varDummy <> 0 Then strUNC = strPathname: GoTo _
           fehlerbehandlung

       'Zwischengespeicherten Rest dranhängen
       strUNC = Left(strUNC, InStr(1, strUNC, _
           Chr(0)) - 1) & strPath

   Else

       'War kein Netzlaufwerk
       strUNC = strPathname

   End If
fehlerbehandlung:
   PfadNachUnc = strUNC
End Function
```

PfadNachUnc

Diese benutzerdefinierte Funktion macht aus einem Pfad, der ein verbundenes Netzlaufwerk enthält, einen Pfad in der UNC-Notation (`\\Server\Freigabename\Pfad`). Dazu wird die API-Funktion `WnetGetConnection` benutzt, an die der Pfad als Parameter `lpszLocalName`, ein Stringpuffer als Parameter `lpszRemoteName`, der den UNC-Pfad aufnimmt, und als Parameter `cbRemoteName` die Größe des Puffers übergeben wird.

8.3 Freies Netzlaufwerk ermitteln

Um Netzlaufwerke zu verbinden, benötigen Sie einen Laufwerksbuchstaben, der noch nicht belegt ist. Die API-Funktion `GetLogicalDrives` liefert die Informationen dazu (Listing 8.3).

Folgendermaßen wird die benutzerdefinierte Funktion `GetFreeDriveLetter` verwendet:

Listing 8.3
Beispiele\08_Dateien\08_01_Laufwerke.xls\mdlFreeDriveletter

```
Private Sub TestFreeDriveLetter ()
   Dim strMsg As String
   strMsg = " 1. Freier Laufwerksbuchstabe = "
   strMsg = strMsg & GetFreeDriveLetter & vbCrLf
   strMsg = strMsg & " Alle Freie Laufwerksbuchstaben = "
   strMsg = strMsg & GetFreeDriveLetter(True) & vbCrLf
   MsgBox strMsg
End Sub
```

Und hier die Funktion selbst:

```
Private Declare Function GetLogicalDrives _
    Lib "kernel32" () As Long

Public Function GetFreeDriveLetter( _
   Optional blnAll As Boolean _
   ) As String

   Dim lngLW As Long
   Dim i As Long

   lngLW = GetLogicalDrives()
   For i = 97 To 122
      If (lngLW And 2 ^ (i - 97)) = 0 Then
         GetFreeDriveLetter = GetFreeDriveLetter & UCase(Chr(i))
         If Not blnAll Then Exit For
      End If
   Next

End Function
```

8.4 Netzlaufwerke verbinden und trennen

Wie beim Einholen von Informationen über Laufwerke gibt es auch hier zwei Lösungsansätze. Einmal kapselt eine fremde Komponente die Funktionalität der API und wird vom `Windows Script Host` (`WSH`) zur Verfügung gestellt. Wie bei allen anderen Fremdprogrammen sollten Sie sich aber nicht unbedingt darauf verlassen, diese auch auf jedem System vorzufinden.

Mit der API können Sie genauso wie mit dem `WSH` Netzlaufwerke verbinden und auch wieder trennen.

8.4.1 Windows Script Host

Zum Benutzen des WSH müssen Sie `CreateObject` benutzen oder in der VBE einen Verweis darauf setzen. Nachfolgend die Objekterstellung mittels `CreateObject`:

```
Set WshNetwork = CreateObject("WScript.Network")
```

Das Setzen eines Verweises auf den WSH veranschaulicht die folgende Abbildung (Abbildung 8.1):

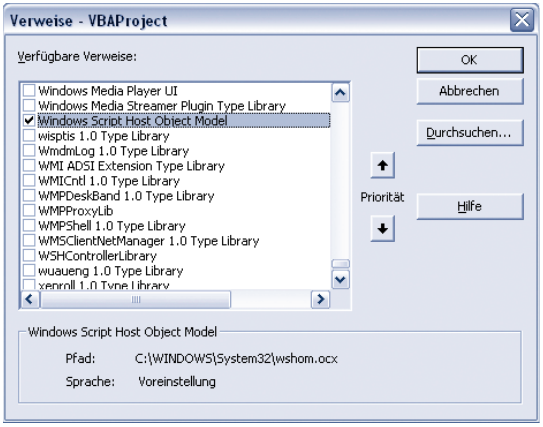


Abbildung 8.1
Verweis WSH

Wenn Sie Ihre Mappe weitergeben wollen, sollten Sie am besten auf den Einsatz des WSH verzichten und die API-Lösung einsetzen.

Mit dem WSH können Sie ohne Probleme Netzlaufwerke verbinden und auch wieder trennen (Listing 8.4). Der erste Parameter der benutzerdefinierten Funktion `MapNetworkDriveWSH` zum Verbinden ist der `UNC`-Pfad, der zweite der Laufwerksbuchstabe, der dritte ist ein optionaler Wahrheitswert, der angibt, ob die Änderung dauerhaft sein soll. Der vierte und fünfte optionale Parameter ist der Benutzername und das Passwort, unter dem der Zugriff erfolgen soll.

Beim Trennen wird der benutzerdefinierten Funktion `UnmapNetworkDriveWSH` als erster Parameter der Laufwerksbuchstabe übergeben. Der zweite entscheidet darüber, ob die Verbindung auch getrennt wird, wenn noch darauf zugegriffen wird. Der dritte ist ein optionaler Wahrheitswert, der angibt, ob die Änderung dauerhaft sein soll.

Listing 8.4
Beispiele\08_Dateien\08_01_Laufwerke.xls\mdlMapWSH

```
Private Sub testMapWSH()

   MapNetworkDriveWSH _
      "\\laptop\mail", _
      "Z", _
      True

   UnmapNetworkDriveWSH _
      "Z", _
      True, _
      True

End Sub
```

Listing 8.4 (Forts.)
Beispiele\08_Dateien\
08_01_Laufwerke.xls\
mdlMapWSH

```
Public Function MapNetworkDriveWSH( _
   ByVal strUNCPath As String, _
   ByVal strDriveLetter As String, _
   Optional ByVal blnPersistant As Boolean, _
   Optional ByVal strUser As String, _
   Optional ByVal strPasswort As String _
   ) As Boolean

   Dim objNetwork As New wshNetwork

   ' Dim objNetwork As Object
   ' Set objNetwork = CreateObject("WScript.Network")

   strDriveLetter = Left(strDriveLetter, 1) & ":"
   objNetwork.MapNetworkDrive strDriveLetter, strUNCPath, _
      blnPersistant, strUser, strPasswort
End Function

Public Function UnmapNetworkDriveWSH( _
   ByVal strDriveLetter As String, _
   Optional ByVal blnForce As Boolean, _
   Optional ByVal blnPersistant As Boolean _
   ) As Boolean

   Dim objNetwork As New wshNetwork

   ' Dim objNetwork As Object
   ' Set objNetwork = CreateObject("WScript.Network")

   strDriveLetter = Left(strDriveLetter, 1) & ":"
   objNetwork.RemoveNetworkDrive strDriveLetter, _
      blnForce, blnPersistant
End Function
```

8.4.2 WNetAddConnection2, WNetCancelConnection2

Ein großer Teil des Codes (Listing 8.5) wird dabei von den Deklarationsanweisungen in Anspruch genommen. Das ist dann auch schon der größte Unterschied zwischen diesen zwei Vorgehensweisen.

Der erste Parameter der benutzerdefinierten Funktion `MapNetworkDriveAPI` zum Verbinden ist der `UNC`-Pfad (`\\Server\Freigabename\Pfad`), der zweite der Laufwerksbuchstabe, der dritte ist ein optionaler Wahrheitswert, der angibt, ob die Änderung dauerhaft sein soll. Der vierte und fünfte optionale Parameter ist der Benutzername und das Passwort, unter dem der Zugriff erfolgen soll.

Beim Trennen wird der benutzerdefinierten Funktion `UnmapNetworkDriveAPI` als erster Parameter der Laufwerksbuchstabe übergeben. Der zweite entscheidet darüber, ob die Verbindung auch dann getrennt wird, wenn noch darauf zugegriffen wird. Der dritte ist ein optionaler Wahrheitswert, der angibt, ob die Änderung dauerhaft sein soll.

Listing 8.5
Beispiele\08_Dateien\
08_01_Laufwerke.xls\mdlMapAPI

```
Private Declare Function WNetAddConnection2 _
   Lib "mpr.dll" Alias "WNetAddConnection2A" ( _
   lpNetResource As NETRESOURCE, _
   ByVal lpPassword As String, _
   ByVal lpstrUserName As String, _
   ByVal dwFlags As Long _
   ) As Long

Private Declare Function WNetCancelConnection2 _
   Lib "mpr.dll" Alias "WNetCancelConnection2A" ( _
   ByVal lpName As String, _
   ByVal dwFlags As Long, _
   ByVal fForce As Long _
   ) As Long

Private Const CONNECT_UPDATE_PROFILE = &H1
Private Const RESOURCE_GLOBALNET = &H2
Private Const RESOURCETYPE_ANY = &H0
Private Const RESOURCEDISPLAYTYPE_SHARE = &H3
Private Const RESOURCEUSAGE_CONNECTABLE = &H1

Private Type NETRESOURCE
   dwScope As Long
   dwType As Long
   dwDisplayType As Long
   dwUsage As Long
   lpLocalName As String
   lpstrUNCPath As String
   lpComment As String
   lpProvider As String
End Type

Private Sub testMapAPI()

   MapNetworkDriveAPI _
      "\\laptop\mail", _
      "Z", _
      True

   UnmapNetworkDriveAPI _
      "Z", _
      True, _
      True

End Sub

Public Function MapNetworkDriveAPI( _
   strUNCPath As String, _
   strDriveLetter As String, _
   Optional ByVal blnPersistant As Boolean, _
   Optional ByVal strUser As String, _
   Optional ByVal strPasswort As String _
   ) As Boolean

   Dim lngRet        As Long
   Dim udtNetres     As NETRESOURCE
   Dim lngPersistant As Long
```

Listing 8.5 (Forts.)
Beispiele\08_Dateien\
08_01_Laufwerke.xls\mdlMapAPI

```
    ' An Laufwerksbuchstaben Doppelpunkt hängen
    strDriveLetter = Left(strDriveLetter, 1) & ":"

    ' Die Struktur ausfüllen
    With udtNetres
       .dwScope = RESOURCE_GLOBALNET
       .dwType = RESOURCETYPE_ANY
       .dwDisplayType = RESOURCEDISPLAYTYPE_SHARE
       .dwUsage = RESOURCEUSAGE_CONNECTABLE
       .lpstrUNCPath = strUNCPath
       .lpLocalName = strDriveLetter
    End With

    ' Benutzername oder mit vbNullString ignorieren
    If strUser = "" Then strUser = vbNullString

    ' Passwort oder mit vbNullString ignorieren
    If strPasswort = "" Then strPasswort = vbNullString

    ' Passwort oder mit vbNullString ignorieren
    If blnPersistant Then _
       lngPersistant = CONNECT_UPDATE_PROFILE

    ' Verbinden
    lngRet = WNetAddConnection2(udtNetres, strPasswort, _
    strUser, lngPersistant)

    ' Rückgabewert auswerten, ob erfolgreich verbunden
    If lngRet = 0 Then MapNetworkDriveAPI = True

End Function

Public Function UnmapNetworkDriveAPI( _
   ByVal strDriveLetter As String, _
   Optional ByVal blnForce As Boolean, _
   Optional ByVal blnPersistant As Boolean _
   ) As Boolean

   Dim lngRet     As Long
   Dim lngFlags   As Long

   If blnPersistant Then lngFlags = CONNECT_UPDATE_PROFILE

   ' An den Laufwerksbuchstaben einen Doppelpunkt hängen
   strDriveLetter = Left(strDriveLetter, 1) & ":"

   ' Trennen
   lngRet = WNetCancelConnection2( _
      strDriveLetter, lngFlags, blnForce)

   ' Rückgabewert auswerten, ob erfolgreich getrennt
   If lngRet = 0 Then UnmapNetworkDriveAPI = True

End Function
```

9

Datum und Zeit

9.1 Was Sie in diesem Kapitel erwartet

In diesem Kapitel bekommen Sie einige Funktionen vorgestellt, die sich mit dem Datum und der Zeit beschäftigen.

Darunter ist eine Funktion zum Berechnen des Ostertages, von dem direkt einige andere kirchliche Feiertage abhängen.

Ein weiteres kleines Beispiel befasst sich mit der Berechnung der Kalenderwoche nach EN 28601, denn die in Excel eingebaute Tabellenfunktion und auch andere oft benutzte Funktionen liefern nicht im jedem Fall richtige Ergebnisse. Darauf aufbauend berechnet eine weitere kleine Funktion den Montag einer übergebenen Kalenderwoche.

Mit einer anderen Funktion berechnen Sie das Alter an einem bestimmten Tag. Auch ein Thema ist die Eingabe von Datum und Zeit ohne Punkte und Doppelpunkte. Die derart eingegebenen Zahlen werden mit dem vorliegenden Code automatisch in ein echtes Datum oder in eine Zeit umgewandelt.

Schließlich liefern zwei benutzerdefinierte Funktionen die Zeiten von Sonnenauf- und Sonnenuntergang an einem übergebenen Tag.

9.2 Ostern und Feiertage

Ostern ist ein sehr wichtiger Tag, wenn es darum geht, Feiertage zu berechnen. Viele andere kirchliche Feiertage eines Jahres sind abhängig von diesem Tag.

Folgende Fakten zum Berechnen des Ostertages werden zugrunde gelegt:

- Ostern ist der erste Sonntag nach dem ersten Frühlingsvollmond.
- Der erste Frühlingsvollmond ist der erste Vollmond, der am 21.3 oder danach stattfindet.

- Ostern muss zwischen dem 22. März und dem 25. April (jeweils einschließlich) liegen.
- Der synodische Monat dauert 29,5306 Tage.
- 225 synodische Monate dauern 6939,688 Tage und 19 tropische Jahre 6939,602 Tage. Diese zwei Zyklen sind nahezu gleich lange, dieser metonische Zyklus bedeutet, dass sich die Tage eines Jahres, an denen Vollmond herrscht, im 19-Jahre-Zyklus wiederholen.

Und so (Listing 9.1) wird der Ostersonntag nach der Schwimmerschen Osterformel berechnet:

Listing 9.1
Beispiele\09_DatumZeit\09_01_DateTime.xls\mdlDateTime

```
Function Ostern(J As Integer) As Date '1900-2100
   Ostern = CDate(Int((Abs(Abs(J - 2015) - 47.5) = _
      13.5) + 0.9 + (DateSerial(J, 3, 21) + _
      ((204 - 11 * (J Mod 19)) Mod 30)) / 7) * 7 + 1)
End Function
```

Mit `(J Mod 19)` finden Sie das Jahr in dem metonischen Zyklus heraus. Im ersten Jahr ist Vollmond am 14.4. Des Weiteren können Sie feststellen, dass in den Folgejahren der Vollmond jeweils 11 Tage vorher ist. Wenn der Termin vor dem 21. März liegt, werden 30 Tage hinzugezählt. Brauchen Sie den Offset zum 21. März, liefert `((204 - 11 * (J Mod 19)) Mod 30)` das richtige Ergebnis. Danach wird das Ergebnis durch 7 geteilt, 0,9 hinzugezählt und die Nachkommastellen werden abgeschnitten. Mit 7 multipliziert und eins hinzugezählt ist man beim Ostersonntag.

Leider passen nicht alle Jahre in das Schema. Die Jahre 1954, 1981, 2049 und 2076 bereiten Probleme und man kommt auf den Ostermontag. Die Berechnung des Terms `(Abs(Abs(Jahr - 2015) - 47.5) = 13.5)` liefert in diesen Jahren das Ergebnis -1, sonst 0. Das wird zu dem Ergebnis hinzugezählt und auch diese Unzulänglichkeit wurde beseitigt.

Trösten Sie sich, wenn Sie das nicht kapieren, sogar mir als Entwickler der Funktion fällt es nicht leicht, aber es funktioniert hervorragend.

Übrigens ist das Osterfest auch mit dafür verantwortlich, dass 1582 eine Kalenderreform durchgeführt wurde:

Cäsar führte 46 v. Chr. den julianischen Kalender mit einem Schalttag alle vier Jahre ein. Nach Cäsar wurde die Schaltjahresregel fehlerhaft angewendet. Augustus musste dann 8 n. Chr. den Kalender korrigieren. Da das julianische Jahr um 11 Minuten und 14 Sekunden länger als das tropische Jahr war, entfernte sich im Laufe der Jahrhunderte der wahre Frühlingsbeginn vom 21. März und damit verlor das Osterfest den gewollten Bezug zum Passahfest.

1582 wurde von Papst Gregor eine Kalenderreform durchgeführt. Auf den 4. Okt 1582 folgte sofort der 15. Okt 1582. Er führte auch ein, das Schalttage, die durch 100, aber nicht durch 400 teilbar sind, wegfallen.

9.2.1 Feiertage

Nachfolgend eine Funktion (Listing 9.2), die einen Wahrheitswert liefert, der Auskunft darüber gibt, ob es sich bei einem übergebenen Datum um einen Feiertag handelt.

Listing 9.2
Beispiele\09_DatumZeit\
09_01_DateTime.xls\mdlDateTime

```
Public Function IsFeiertag(dteDate As Date) As Boolean
   Dim dteEastern       As Date
   Dim lngIndex         As Long
   Static Feiertage()   As Variant
   Dim intYear          As Integer

   If Year(dteDate) <> intYear Then
      intYear = Year(dteDate)
      dteEastern = CDate(Int((Abs(Abs(intYear - 2015) - 47.5) = _
         13.5) + 0.9 + (DateSerial(intYear, 3, 21) + _
         ((204 - 11 * (intYear Mod 19)) Mod 30)) / 7) * 7 + 1)
      ReDim Feiertage(1 To 13)
      Feiertage(1) = DateSerial(intYear, 1, 1)
      Feiertage(2) = dteEastern - 2 'Karfreitag
      Feiertage(3) = dteEastern + 1 'Ostermontag
      Feiertage(4) = dteEastern + 39 'Himmelfahrt
      Feiertage(5) = dteEastern + 49 'Pfingstsonntag
      Feiertage(6) = dteEastern + 50 'Pfingstmontag
      Feiertage(7) = DateSerial(intYear, 5, 1) 'Maifeiertag
      Feiertage(8) = DateSerial(intYear, 10, 3) 'Tag_der_Einheit
      Feiertage(9) = DateSerial(intYear, 12, 25) '1. Weihnachtsf.
      Feiertage(10) = DateSerial(intYear, 12, 26) '2. Weihnachtsf.

      'Wenn Tarifvertrag es vorsieht
      Feiertage(11) = DateSerial(intYear, 12, 24) 'Heiligabend
      Feiertage(12) = DateSerial(intYear, 12, 31) 'Silvester

      'Nicht überall
      'Mittwoch zwischen dem 16. und 22.11
      Feiertage(13) = DateSerial(intYear, 11, 22) - _
         (DateSerial(intYear, 11, 18) Mod 7) 'Böse-Bube-Tag

   End If

   For lngIndex = 1 To 13
      If dteDate = Feiertage(lngIndex) _
         Then IsFeiertag = True: Exit For
   Next

End Function
```

Hier noch einmal eine Funktion für den Buß- und Bettag (Listing 9.3):

Listing 9.3
Beispiele\09_DatumZeit\
09_01_DateTime.xls\mdlDateTime

```
Public Function BussUndBettag(IntYear As Integer)

   ' Buß- und Bettag ist der Mittwoch zwischen
   ' 16. und 22. November
  BussUndBettag = DateSerial(IntYear, 11, 22) - _
   WeekDay(DateSerial(IntYear, 11, 25), vbSunday)
End Function
```

9.2.2 Wochenende oder Feiertag

Die folgende Funktion (Listing 9.4) benutzt die Funktion `IsFeiertag` und liefert einen Wahrheitswert, der Auskunft darüber gibt, ob es sich bei einem übergebenen Datum um einen Feiertag oder um das Wochenende handelt.

Listing 9.4
Beispiele\09_DatumZeit\
09_01_DateTime.xls\mdlDateTime

```
Public Function IsWochenendeOderFeiertag(dteDate As Date)
   If dteDate = 0 Then Exit Function
   If (WeekDay(dteDate) = 1) Or _
      (WeekDay(dteDate) = 7) Or _
      IsFeiertag(dteDate) Then _
      IsWochenendeOderFeiertag = True
End Function
```

Diese Funktion ist nützlich, wenn Sie in einem Tabellenblatt diese Tage farblich kennzeichnen möchten. Dazu wird die bedingte Formatierung (hier in Zelle A3) eingesetzt.

Wählen Sie den Menüpunkt FORMAT | BEDINGTE FORMATIERUNG.

Bedingung 1: Formel ist `=IsWochenendeOderFeiertag(A3)`

Setzen Sie diese oder die Funktion `IsFeiertag` aber nicht allzu häufig bei der bedingten Formatierung ein, eine Neuberechnung verschlingt recht viel Zeit.

9.3 Weitere Zeitfunktionen

9.3.1 Kalenderwoche

Viele Funktionen, auch die in Excel eingebaute Funktion `Kalenderwoche` aus dem Analyse-Add-In, liefern falsche Werte nach der Europäischen Norm EN 28601. Darunter auch diese, häufig verwendete benutzerdefinierte Funktion:

```
Public Function KalWocheFalsch(dteDate As Date) As Integer
   KalWocheFalsch = Format(dteDate, "ww", vbMonday, vbFirstFourDays)
End Function
```

Hier ein paar Tage, an denen solche Funktionen falsche Werte liefern:

31.12.1951, 30.12.1963, 29.12.2003, 31.12.2007

Folgende benutzerdefinierte Funktion liefert dagegen richtige Werte (Listing 9.5):

Listing 9.5
Beispiele\09_DatumZeit\
09_01_DateTime.xls\mdlDateTime

```
Public Function Kalwoche(dteDate As Date) As Integer
   Kalwoche = ((DatePart("ww", dteDate, 2, 2) = 53) _
      And (Day(dteDate) >= 29) And (dteDate Mod 7 = 2)) _
      * 52 + DatePart("ww", dteDate, 2, 2)
End Function
```

Es gibt auch noch ein paar andere Funktionen, die das Gleiche machen und dabei noch etwas kürzer sind. Unter *groups.google.de* finden Sie einige, die genial kurz, aber leider nicht von mir sind.

Zu den Kalenderwochen nach EN 28601 gibt es noch Folgendes anzumerken:

Wenn der 1. Januar eines Jahres auf einen Montag, Dienstag, Mittwoch oder Donnerstag fällt, gehört er zur ersten Kalenderwoche. Wenn dieser Tag ein Freitag, Samstag oder Sonntag ist, zählt er zur letzten Kalenderwoche des vorherigen Jahres.

Der 29., 30. und 31.12. des vorherigen Jahres gehören schon zur ersten Kalenderwoche des neuen Jahres, wenn der 31.12. auf einen Montag, Dienstag oder Mittwoch fällt.

9.3.2 Montag der Woche

Die folgende Funktion (Listing 9.6) liefert den Montag einer übergebenen Kalenderwoche des angegebenen Jahres:

Listing 9.6
Beispiele\09_DatumZeit\
09_01_DateTime.xls\mdlDateTime

```
Public Function MontagDerWoche( _
   lngWeek As Long, lngYear) As Date

   Dim t As Date

   t = DateSerial(lngYear, 1, 1) + (7 - WeekDay( _
      DateSerial(lngYear, 1, 1), 3))

   MontagDerWoche = t + 7 * (lngWeek - (((DatePart("ww", _
      t, 2, 2) = 53) And (Day(t) >= 29) And (WeekDay(t, 2) _
      = 1)) * 52 + DatePart("ww", t, 2, 2)))

End Function
```

9.3.3 Lebensalter

Nachfolgende Funktionen liefern das Alter an einem bestimmten Tag (Listing 9.7).

Listing 9.7
Beispiele\09_DatumZeit\
09_01_DateTime.xls\mdlDateTime

```
Public Function Lebensalter( _
   dteBirtday As Date, dteDay As Date _
   ) As Integer

   Lebensalter = (DateSerial(2000, Month(dteBirtday), _
      Day(dteBirtday)) > DateSerial(2000, Month(dteDay), _
      Day(dteDay))) + Year(dteDay) - Year(dteBirtday)
End Function2

Public Function Lebensalter1( _
   dtmBirtday As Date, dtmDay As Date _
   ) As Integer

   Lebensalter1 = DateDiff("YYYY", dtmBirtday, dtmDay)
End Function
```

9.4 Datums- und Zeiteingabe

Wenn Sie ein Datum eingeben wollen, und zwar ohne zusätzliche Zeichen wie den Punkt, hilft Ihnen die folgende Prozedur weiter (Listing 9.8). Sie macht z.B. aus 240160 den 24.01.1960, und zwar als echtes Datum, nicht nur als Anzeige oder Text.

Listing 9.8
Beispiele\09_DatumZeit\
09_01_DateTime.xls\mdlDateTime

```
Public Sub AusZahlDatum(ByVal Target As Excel.Range)
   Dim varDummy As Variant
   Dim intDay As Integer
   Dim intMonth As Integer
   Dim intYear As Integer

   On Error GoTo fehlerbehandlung

   varDummy = Target.Value2

   ' Abbrechen, wenn keine Zahl oder ein Datum
   If (IsNumeric(varDummy) = False) And _
      (IsDate(varDummy) = False) Then _
      Exit Sub

   ' Abbrechen, wenn Zahl nicht umgewandelt werden kann
   If (Mid$(varDummy, 5, 4) < 1000) And _
      (varDummy < 10000 Or varDummy > 999999) Then _
      Exit Sub

   ' Formatieren
   varDummy = Format(CStr(varDummy), "000000")

   ' in ein Datum umwandeln
   intDay = Mid$(varDummy, 1, 2)
   intMonth = Mid$(varDummy, 3, 2)
   intYear = Mid$(varDummy, 5, 4)
   varDummy = DateSerial(intYear, intMonth, intDay)

   ' Ereignisse ausschalten
   Application.EnableEvents = False

   ' Datum eintragen
   Target.Value = varDummy

   ' Formatieren
   Target.NumberFormat = "dd.mm.yyyy"

fehlerbehandlung:
   ' Ereignisse einschalten
   Application.EnableEvents = True
End Sub
```

Wenn Sie eine Zeit ohne einen Doppelpunkt eingeben wollen, hilft Ihnen die folgende Prozedur weiter (Listing 9.9). Sie macht z.B. aus 1420 die Zeit 14:20, und zwar als echte Zeit, nicht nur als Anzeige oder Text.

Listing 9.9
Beispiele\09_DatumZeit\
09_01_DateTime.xls\mdlDateTime

```
Public Sub AusZahlZeit(ByVal Target As Excel.Range)
   Dim varDummy As Variant

   On Error GoTo fehlerbehandlung

   varDummy = Target.Value2
   If varDummy = "" Then Exit Sub

   ' Abbrechen, wenn keine Zahl oder ein Datum
   If Not (IsNumeric(varDummy) Or IsDate(varDummy)) Then Exit Sub

   ' Formatieren
   varDummy = String(4 - Len(varDummy), Asc("0")) & varDummy

   ' In Zeit umwandeln
   varDummy = TimeSerial(Left$(varDummy, Len(varDummy) - 2), _
      Right$(varDummy, 2), 0)

   ' Ereignisse ausschalten
   Application.EnableEvents = False

   ' Zeit eintragen
   Target.Value = varDummy

   ' Formatieren
   Target.NumberFormat = "hh:mm"

fehlerbehandlung:
   ' Ereignisse einschalten
   Application.EnableEvents = True
End Sub
```

Diese beiden vorherigen Prozeduren werden im `Change`-Ereignis (Listing 9.10) eines Tabellenblattes aufgerufen. Die Spalte A wird dabei in ein Datum, die Spalte B in eine Zeit umgewandelt.

Listing 9.10
Beispiele\09_DatumZeit\
09_01_DateTime.xls\mdlDateTime

```
Private Sub Worksheet_Change(ByVal Target As Excel.Range)
   Dim Zielbereich As Range

   If Target.Count > 1 Then Exit Sub

   'Überprüfen, ob die richtige Zelle für Datum geändert wird
   Set Zielbereich = Application.Intersect(Range("A:A"), Target)
   If Not (Zielbereich Is Nothing) Then
       AusZahlDatum Target
   End If

   'Überprüfen, ob die richtige Zelle für Zeit geändert wird
   Set Zielbereich = Application.Intersect(Range("B:B"), Target)
   If Not (Zielbereich Is Nothing) Then
       AusZahlZeit Target
   End If

End Sub
```

9.5 Sonnenauf- und Sonnenuntergang

Wenn die Sonne am höchsten steht, ist Mittag, und zwar 12:00 Uhr mittlere Ortszeit.

Leider stimmt diese Aussage nicht so ganz. Aus verschiedenen Gründen weicht die wahre Ortszeit von der mittleren Ortszeit ab. Die Abweichung liegt in der Größenordnung von +/– 15 Minuten. Um das zu kompensieren, könnten Sie eine Liste mit den Abweichungen für jeden Tag verwenden, aber hier wird eine Zeitgleichung benutzt.

Des Weiteren brauchen Sie die Deklination der Sonne für den Stichtag. Im Prinzip ist die Deklination der Breitengrad, über dem die Sonne an diesem Tag zur Mittagszeit senkrecht steht.

Mit dem Längen- und Breitengrad des aktuellen Standorts, der Deklination und der Zeitabweichung kann man nun die Zeiten berechnen (Listing 9.11). Zu beachten ist, dass alle Winkel im Bogenmaß benötigt werden. Leider kennt VBA kein `Arccos`, deshalb muss das mit anderen Winkelfunktionen mühsam nachgebildet werden. Man könnte zwar die entsprechende Tabellenfunktion benutzen, aber dann könnten diese Funktionen nicht mehr ohne weiteres in anderen Office-Anwendungen eingesetzt werden.

Listing 9.11
Beispiele\09_DatumZeit\
09_01_DateTime.xls\mdlSun

```
Private Sub TestSun()
   Dim strMorning As String
   Dim strEvening As String
   Dim strMsg     As String
   Dim dtmTime    As String

   dtmTime = ToSummerWinter(Date + Sonnenaufgang(8.7, 50.12, Now))
   strMorning = "Sonnenaufgang : " & Format(dtmTime, "hh:nn")

   dtmTime = ToSummerWinter(Date + Sonnenuntergang(8.7, 50.12, Now))
   strEvening = "Sonnenuntergang : " & Format(dtmTime, "hh:nn")

   MsgBox strMorning & vbCrLf & strEvening, , _
      Format(Now, "DDD DD.MM.YYYY")
End Sub

Public Function Sonnenaufgang( _
    Längengrad As Double, _
    Breitengrad As Double, _
    Datum As Date) As Date

   Dim dblDeklination   As Double
   Dim dblDiffMorning   As Double
   Dim dblDayOfYear     As Double
   Dim dblHeight        As Double
   Dim dblDiffEvening   As Double
   Dim dblDummy         As Double
   Const Pi = 3.141592653

   'Sonnenaufgang bei -50 Bogenminuten
   dblHeight = (-50 / 60) * Pi / 180
```

Listing 9.11 (Forts.)
Beispiele\09_DatumZeit\
09_01_DateTime.xls\mdlSun

```
    'Tag des Jahres
    dblDayOfYear = Datum - DateSerial(Year(Datum), 1, 0)

    Breitengrad = Breitengrad * Pi / 180
    Längengrad = Längengrad * Pi / 180

    'Breitengrad, über dem die Sonne mittags senkrecht steht
    dblDeklination = 0.40954 * Sin(0.0172 * (dblDayOfYear - 79.35))

    'Differenzen zum Mittag in Stunden berechnen
    dblDummy = (Sin(dblHeight) - Sin(Breitengrad) * _
        Sin(dblDeklination)) / (Cos(Breitengrad) * Cos(dblDeklination))
    dblDiffEvening = 12 * (Atn((dblDummy * -1) / _
       Sqr(dblDummy * -1 * dblDummy + 1)) + 2 * Atn(1)) / Pi
    dblDiffMorning = -0.1752 * Sin(0.03343 * dblDayOfYear + _
       0.5474) - 0.134 * Sin(0.018234 * dblDayOfYear - 0.1939)

    Sonnenaufgang = (12 - dblDiffEvening - dblDiffMorning + _
        (15 - Längengrad * 180 / Pi) * 4 / 60) / 24

End Function

Public Function Sonnenuntergang( _
    Längengrad As Double, _
    Breitengrad As Double, _
    Datum As Date) As Date

   Dim dblDeklination   As Double
   Dim dblDiffMorning   As Double
   Dim dblDayOfYear     As Double
   Dim dblHeight        As Double
   Dim dblDiffEvening   As Double
   Dim dblDummy         As Double
   Const Pi = 3.141592653

   'Sonnenuntergang bei -50 Bogenminuten
   dblHeight = (-50 / 60) * Pi / 180

   'Tag des Jahres
   dblDayOfYear = Datum - DateSerial(Year(Datum), 1, 0)

   Breitengrad = Breitengrad * Pi / 180
   Längengrad = Längengrad * Pi / 180

   'Breitengrad, über dem die Sonne mittags senkrecht steht
   dblDeklination = 0.40954 * Sin(0.0172 * (dblDayOfYear - 79.35))

   'Differenzen zum Mittag in Stunden berechnen
   dblDummy = (Sin(dblHeight) - Sin(Breitengrad) * _
       Sin(dblDeklination)) / (Cos(Breitengrad) * Cos(dblDeklination))
   dblDiffEvening = 12 * (Atn((dblDummy * -1) / _
      Sqr(dblDummy * -1 * dblDummy + 1)) + 2 * Atn(1)) / Pi
   dblDiffMorning = -0.1752 * Sin(0.03343 * dblDayOfYear + _
      0.5474) - 0.134 * Sin(0.018234 * dblDayOfYear - 0.1939)
```

Listing 9.11 (Forts.)
Beispiele\09_DatumZeit\
09_01_DateTime.xls\mdlSun

```
    Sonnenuntergang = (12 + dblDiffEvening - dblDiffMorning + _
       (15 - Längengrad * 180 / Pi) * 4 / 60) / 24

End Function

Public Function ToSummerWinter(dtmDateTime As Date) As Date
  Dim dtmBegin As Date
  Dim dtmEnd   As Date
  Dim lngYear  As Long

  lngYear = Year(dtmDateTime)

  dtmBegin = DateSerial(lngYear, 4, 0) - (WeekDay( _
    DateSerial(lngYear, 4, 0), 2) Mod 7) + TimeSerial(1, 0, 0)
  dtmEnd = DateSerial(lngYear, 11, 0) - (WeekDay( _
    DateSerial(lngYear, 11, 0), 2) Mod 7) + TimeSerial(1, 0, 0)

  If dtmDateTime > dtmBegin And dtmDateTime < dtmEnd Then
    ToSummerWinter = dtmDateTime + TimeSerial(1, 0, 0)
  Else
    ToSummerWinter = dtmDateTime
  End If

End Function
```

10

Grafik

10.1 Was Sie in diesem Kapitel erwartet

Ein Thema in diesem Kapitel befasst sich mit dem Erstellen einer Bilderschau. Dabei werden die Bilder erst beim Überfahren mit der Maus angezeigt und über einen Hyperlink kann das Bild dann geöffnet werden.

Häufig werden Icons für eigene Menüs benötigt. Ein Beispiel zeigt, wie Sie diese aus fremden Bibliotheken und ausführbaren Dateien extrahieren und für Ihre eigene Zwecke weiterverwenden können.

Ein weiteres Beispiel beschäftigt sich damit, programmgesteuert die Bildschirmeinstellungen auszulesen und zu ändern. Nebenbei wird noch eine Möglichkeit gezeigt, wie Sie Windows neu starten können.

Windows enthält einen kompletten Satz Spielkarten. Diese können auch in Excel benutzt werden und als Objekt in ein Tabellenblatt eingefügt werden, wie in einem weiteren Beispiel gezeigt wird. Es ist auch möglich, diese Spielkarten einzeln als *Bitmap*-Datei zu exportieren.

Fortschrittsanzeigen können auf verschiedene Arten realisiert werden. Hier wird eine Möglichkeit vorgestellt, die Statusleiste dafür zu verwenden. Dabei kann dort ein echter Balken in allen möglichen Farben gezeichnet werden. Es ist zusätzlich möglich, beliebige Schriftarten und Farben für die Textausgabe zu verwenden. Gleichzeitig lernen Sie eine einfache Möglichkeit kennen, alle installierten Schriftarten auszugeben.

10.2 Bilderschau

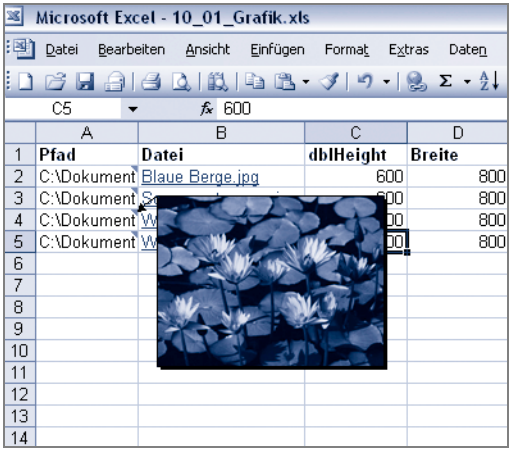
Wenn Sie mit Excel eine Übersicht der Bilder eines Verzeichnisses haben wollen, können Sie das Einfügen und Formatieren der Bilder aufzeichnen und diesen Code als Grundlage für den automatisierten Vorgang benutzen. Das Aufzeichnen ist übrigens der beste Weg, wie Sie an die notwendigen Informationen zum Benutzen von Eigenschaften und Methoden unbekannter Objekte kommen und man sollte sich nicht schämen, diesen auch zu benutzen.

Leider bringen Bilder, die in eine Tabelle eingebettet sind, viel Verdruss. Jedes neue Formatieren der Tabelle kann die Position der Bilder ändern und außerdem nehmen die Bilder relativ viel Platz weg, sodass das Ganze doch recht unübersichtlich wird.

Einen Ausweg liefern Kommentare, bei denen als Hintergrund eine Grafik verwendet werden kann. Erst, wenn Sie mit dem Mauszeiger darüber stehen, wird wie bei einem Kommentar das Bild eingeblendet (Abbildung 10.1). Nun ist es aber mühselig, so etwas von Hand zu machen, deshalb liest das nachfolgende Beispiel (Listing 10.1) ein gewähltes Verzeichnis samt dessen Unterverzeichnissen aus und fügt für jedes JPG-Bild eine Zeile mit Informationen sowie einen Kommentar mit dem Bild in ein Tabellenblatt ein.

Sie sollten es aber nicht übertreiben, was die Anzahl der Bilder angeht, Sie kommen sonst recht schnell zu Dateigrößen, bei denen das Arbeiten mit der Mappe extrem langsam wird.

Abbildung 10.1
Kommentare als Bildercontainer



Listing 10.1
Beispiele\10_Grafik\
10_01_Grafik.xls\mdlPicture

```
Private Declare Function GetShortPathName _
   Lib "kernel32" Alias "GetShortPathNameA" ( _
   ByVal lpszLongPath As String, _
   ByVal lpszShortPath As String, _
   ByVal lBuffer As Long _
   ) As Long

Public Sub Bildervorschau()
   Dim strPath    As String
   Dim strFile    As String
   Dim lngRow     As Long
   Dim dblWidth   As Double
   Dim dblHeight  As Double
   Dim dblRate    As Double
   Dim objComment As Comment
   Dim colFiles   As New Collection
   Dim varFiles   As Variant
   Dim ShortPath  As String
```

Listing 10.1 (Forts.)
Beispiele\10_Grafik\
10_01_Grafik.xls\mdlPicture

```
    ' Ordner über Dialog wählen
    strPath = VBGetFolder("Ordner wählen", "C:\")

    ' Verlassen, wenn nichts gewählt
    If strPath = "" Then Exit Sub

    ' Nach .JPG Files suchen
    myFilesearch strPath, colFiles, "jpg"

    With Worksheets("Bilderschau")

        ' Altes Löschen
        .Range("A2:D65535").Clear

        lngRow = 1

        ' Alle .JPG Files nacheinander durchlaufen
        For Each varFiles In colFiles

            ' Größe ermitteln
            JPEGSize varFiles, dblWidth, dblHeight

            ' Kurzen Dateipfad ermitteln
            ShortPath = String(251, 0)
            GetShortPathName varFiles, ShortPath, 250
            ShortPath = Left(ShortPath, InStr(1,
                ShortPath, Chr(0)) - 1)

            ' Pfad, Dateiname und Größe eintragen
            lngRow = lngRow + 1
            .Cells(lngRow, 1) = varFiles
            .Cells(lngRow, 2) = Dir(varFiles)
            .Cells(lngRow, 3) = dblHeight
            .Cells(lngRow, 4) = dblWidth
            dblRate = dblWidth / dblHeight

            ' Einen Hyperlink auf die Datei setzen
            ActiveSheet.Hyperlinks.Add Anchor:=.
                Cells(lngRow, 2), Address:=ShortPath

            ' Kommentar einfügen
            Set objComment = .Cells(lngRow, 1).AddComment

            ' In den Kommentar das Bild in der Höhe 100
            ' maßstabsgerecht einfügen
            With objComment
                .Shape.Fill.UserPicture ShortPath
                .Shape.Height = 100
                .Shape.Width = .Shape.Height * dblRate
            End With

        Next
    End With
End Sub
```

Listing 10.1 (Forts.) Beispiele\10_Grafik\10_01_Grafik.xls\mdlPicture

```
Private Sub myFilesearch( _
   ByVal strStart As String, _
   ByRef colList As Collection, _
   Optional ByVal strFilter As String)

   Dim astrFolder()  As String
   Dim i             As Long
   Dim strCurFolder  As String
   Dim strFile       As String

   'Erst einmal 100 Unterverzeichnisse annehmen
   ReDim astrFolder(1 To 100)

   If Left(strFilter, 1) <> "*" Then strFilter = "*" & strFilter

   If Right$(strStart, 1) <> "\" Then

       'Nachschauen, ob übergebener Pfad auch einen
       'Backslash enthält. Wenn nicht, dann anhängen
       strStart = strStart & "\"

   End If

   strCurFolder = strStart

   ' Alle Dateien liefern
   strStart = strStart & "*"

   ' Suche mit Dir initialisieren
   strFile = Dir(strStart, vbSystem Or _
      vbHidden Or vbDirectory Or vbNormal)

   Do While strFile <> ""
   ' So lange durchlaufen, wie
   ' durch Dir() etwas geliefert wird

       If GetAttr(strCurFolder & strFile) And vbDirectory Then
       'wenn Datei ein Verzeichnis ist,
           If Right$(strFile, 1) <> "." Then
                ' und zwar ein untergeordnetes (Punkte sind
                ' gleiche oder übergeordnete Verzeichnisse)

                i = i + 1

                If i > UBound(astrFolder) Then

                   'Wenn Array zu klein ist, anpassen
                    ReDim Preserve astrFolder(1 To i + 1)

                End If

                'dann ein Array mit Verzeichnissen füllen.
                astrFolder(i) = strFile

           End If
```

Listing 10.1 (Forts.)
Beispiele\10_Grafik\
10_01_Grafik.xls\mdlPicture

```
      Else
      'Handelt es sich um eine Datei,

         If LCase(strFile) Like LCase(strFilter) Then
            'und entspricht sie noch den strFilterbedingungen,
            'dann den Pfad an die Collection colList hängen.
            colList.Add strCurFolder & strFile, _
               strCurFolder & strFile

         End If
      End If
      strFile = Dir$()
   Loop

   ' Keine Unterverzeichnisse vorhanden, dann beenden
   If i = 0 Then Exit Sub

   ' Array anpassen
   ReDim Preserve astrFolder(1 To i)

   'Jetzt erst werden die Unterverzeichnisse abgearbeitet,
   'weil Dir$ mit Rekursionen nicht klarkommt.
   For i = 1 To UBound(astrFolder)

      'Jetzt ruft sich diese Prozedur noch einmal auf.
      myFilesearch strCurFolder & astrFolder(i), colList, strFilter

   Next

End Sub

Private Function JPEGSize( _
   ByVal strFile As String, _
   dblWidth As Double, _
   dblHeight As Double _
   ) As Boolean

   Dim lngFF         As Long
   Dim bytFlags      As Byte
   Dim lngOffs       As Long
   Dim bytA          As Byte
   Dim bytB          As Byte
   Dim lngPointer    As Long

   lngFF = FreeFile
   lngPointer = 1

   ' Datei öffnen
   Open strFile For Binary Access Read As lngFF

      Get lngFF, 2, bytFlags
      Get lngFF, 5, bytA
      Get lngFF, 6, bytB

      ' Nächster Frame
      lngOffs = CLng(bytA) * 256 + CLng(bytB)
```

Listing 10.1 (Forts.)
Beispiele\10_Grafik\
10_01_Grafik.xls\mdlPicture

```
        ' Pointer auf Beginn
        lngPointer = 6
        Do
           If (bytFlags = &HC2) Or (bytFlags = &HC0) Then
           ' Nur diese sind relevant
               ' Breite
              Get lngFF, lngPointer + 4, bytA
              Get lngFF, , bytB
              dblWidth = CLng(bytA) * 256 + CLng(bytB)

              ' Höhe
              Get lngFF, lngPointer + 2, bytA
              Get lngFF, , bytB
              dblHeight = CLng(bytA) * 256 + CLng(bytB)

              JPEGSize = True

              ' Verlassen
              Exit Do

           End If

           ' Pointer auf nächsten
           lngPointer = lngPointer + lngOffs - 2

           Get lngFF, lngPointer + 1, bytA
           If bytA <> 255 Then Exit Do

           Get lngFF, , bytFlags

           lngPointer = lngPointer + 2

           Get lngFF, lngPointer + 1, bytA

           Get lngFF, , bytB

           lngOffs = CLng(bytA) * 256 + CLng(bytB)

           lngPointer = lngPointer + 2
        Loop While bytFlags <> &HD9
     Close
End Function
```

Zum Auswählen des Verzeichnisses, das nach Bildern durchsucht werden soll, wird ein Dialog (`VBGetFolder`) zur Verzeichnisauswahl gestartet, der Code und die Informationen dazu sind in Kapitel 6 zu finden.

Ist das Verzeichnis ausgewählt, werden anschließend alle Unterverzeichnisse nach JPGs durchsucht. Danach wird die Größe jedes einzelnen Bildes ermittelt, um in der Mappe die Seitenverhältnisse richtig darzustellen. Dazu wird in der Funktion `JPEGSize` die Datei binär geöffnet und die Informationen werden ausgelesen. Selbstverständlich muss man dabei die Dateistruktur kennen, aber es würde den Rahmen sprengen, diese hier zu erläutern. Es gibt im Internet hervorragende Seiten, welche die Dateistrukturen der gebräuchlichsten Dateiformate beschreiben. Nehmen Sie den Code also einfach so hin und benutzen Sie ihn als eine Art Wegbeschreibung, um zu den Informationen zu gelangen.

Anschließend wird der Pfad, der Dateiname und die Größe ins Tabellenblatt eingetragen. Zusätzlich wird ein Hyperlink auf die Datei erzeugt und ein Kommentar mit Bild eingefügt. Da der Pfad für die Grafiken zumindest unter Excel 97 auf 99 Zeichen begrenzt ist, werden die 8+3-Datei- und Pfadnamen benutzt. Die API-Funktion `GetShortPathName` erledigt diese Aufgabe.

10.3 Icons extrahieren

In vielen Dateien auf ihrem Rechner stecken ein oder mehrere Icons. Bei den Dateien, die Icons enthalten, handelt es sich in den meisten Fällen um *DLLs* oder *EXE* Dateien. Weitere Aspiranten sind Dateien mit den Endungen *.OCX*, *.RES* und *.SCR*. Mit dem hier vorliegenden Code haben Sie die Möglichkeit, sich die in diesen Dateien enthaltene Icons anzuschauen, ausgewählte als Bitmaps in die Zwischenablage zu kopieren oder sie in ein Tabellenblatt zu kopieren. Lohnenswerte Dateien (in Klammern finden Sie die Anzahl der Icons) finden Sie unter `C:\WINDOWS\SYSTEM32` in den Dateien *ICONLIB.DLL* (ca. 86), *MORICONS.DLL* (ca. 110), *PIFMGR.DLL* (ca. 38), *SETUPAPI.DLL* (ca. 37), *PROGMAN.EXE* (ca. 50), *SHELL32.DLL* (ca. 240) und unter *C:\WINDOWS\SYSTEM* suchen Sie am besten mal nach *COOL.DLL* (ca. 45).

Ein Icon in der Zwischenablage können Sie für selbst hinzugefügte Menüpunkte sehr schön benutzen. Folgende Zeilen (Listing 10.2) fügen dem Kontextmenü der Zellen einen Menüpunkt hinzu und weisen diesem das Icon der Zwischenablage zu. Das Kontextmenü einer Zelle ist dabei das Menü, das aufgeht, wenn ein rechter Mausklick auf eine Zelle erfolgt. Bei der Betätigung des neuen Menüpunktes wird die Prozedur `DoIt` aufgerufen, die lediglich signalisiert, dass etwas passiert.

Listing 10.2
Beispiele\10_Grafik\
10_02_Icon.xls\mdlKontex

```
Public Sub MyKontexMenu()
   Dim cmdCommand As CommandBarButton
   On Error Resume Next

   ' Kontexmenü rechte Maustaste in Zelle
   With CommandBars("Cell")

      ' Menüpunkt hinzufügen
      Set cmdCommand = .Controls.Add(msoControlButton)

      With cmdCommand
         ' Beschriftung
         .Caption = "Do it"

         ' Aussehen
         .FaceId = 3

         ' Aktion beim Klick (Prozedur)
         .OnAction = "DoIt"

         ' Icon aus Zwischenablage als Icon im Menü benutzen
         .PasteFace
      End With
   End With
End Sub
```

Listing 10.2 (Forts.)
Beispiele\10_Grafik\
10_02_Icon.xls\mdlKontex

```
Public Sub ResetKontexmenu()
    ' Menü zurücksetzen
   CommandBars("Cell").Reset
End Sub

Public Sub DoIt()
    ' Anzeigen, dass etwas passiert
   MsgBox "Do it"
End Sub
```

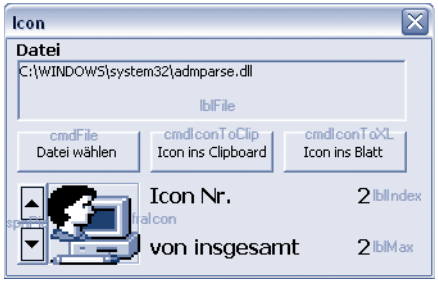
Achtung

Denken Sie daran, dass Sie in den wenigsten Fällen das Recht dazu haben, fremde Icons für eigene Zwecke zu benutzen oder diese gar weiterzugeben.

Nachfolgend der Code (Listing 10.3) für das Extrahieren von Icons aus einer Datei. Legen Sie dazu eine UserForm an, fügen Sie drei CommandButtons, einen SpinButton, ein Rahmensteuerelement und sechs Labels ein.

Den Aufbau und die Namen entnehmen Sie der Abbildung 10.2.

Abbildung 10.2
Aufbau der UserForm mit den Namen der Steuerelemente



Nachfolgend der Code im Klassenmodul der UserForm:

Listing 10.3
Beispiele\10_Grafik\
10_01_Grafik.xls\ufIcon

```
Private Declare Function ExtractIcon _
   Lib "shell32.dll" Alias "ExtractIconA" ( _
   ByVal hInst As Long, _
   ByVal lpszExeFileName As String, _
   ByVal nIconIndex As Long _
   ) As Long

Private Declare Function DestroyIcon _
   Lib "user32" ( _
   ByVal lngHandleIcon As Long _
   ) As Long

Private Declare Function DrawIconEx _
   Lib "user32" ( _
   ByVal lngDC As Long, _
   ByVal xLeft As Long, _
   ByVal yTop As Long, _
   ByVal lngHandleIcon As Long, _
   ByVal cxWidth As Long, _
   ByVal cyWidth As Long, _
```

Listing 10.3 (Forts.)
Beispiele\10_Grafik\
10_01_Grafik.xls\ufIcon

```
    ByVal istepIfAniCur As Long, _
    ByVal hbrFlickerFreeDraw As Long, _
    ByVal diFlags As Long _
    ) As Long

Private Declare Function FindWindow _
    Lib "user32" _
    Alias "FindWindowA" ( _
    ByVal lpClassName As String, _
    ByVal lpWindowName As String _
    ) As Long

Private Declare Function OpenClipboard _
    Lib "user32" ( _
    ByVal hwnd As Long _
    ) As Long

Private Declare Function EmptyClipboard _
    Lib "user32" () As Long

Private Declare Function SetClipboardData _
    Lib "user32" ( _
    ByVal wFormat As Long, _
    ByVal hMem As Long _
    ) As Long

Private Declare Function CloseClipboard _
    Lib "user32" () As Long

Private Declare Function GetDC _
    Lib "user32" ( _
    ByVal hwnd As Long _
    ) As Long

Private Declare Function GetDeviceCaps _
    Lib "gdi32" ( _
    ByVal lngDC As Long, _
    ByVal nIndex As Long _
    ) As Long

Private Declare Function ReleaseDC _
    Lib "user32" ( _
    ByVal hwnd As Long, _
    ByVal lngDC As Long _
    ) As Long

Private Declare Function DeleteDC _
    Lib "gdi32" ( _
    ByVal lngDC As Long _
    ) As Long

Private Declare Function SelectObject _
    Lib "gdi32" ( _
    ByVal lngDC As Long, _
    ByVal hObject As Long _
    ) As Long
```

Listing 10.3 (Forts.)
Beispiele\10_Grafik\
10_01_Grafik.xls\ufIcon

```
Private Declare Function CreateSolidBrush _
   Lib "gdi32" ( _
   ByVal crColor As Long _
   ) As Long

Private Declare Function FillRect _
   Lib "user32" ( _
   ByVal lngDC As Long, _
   lpRect As RECT, _
   ByVal lngHandleBrush As Long _
   ) As Long

Private Declare Function CreateCompatibleDC _
   Lib "gdi32" ( _
   ByVal lngDC As Long _
   ) As Long

Private Declare Function CreateBitmap _
   Lib "gdi32" ( _
   ByVal nWidth As Long, ByVal nHeight As Long, _
   ByVal nPlanes As Long, _
   ByVal nBitCount As Long, _
   lpBits As Any _
   ) As Long

Private Declare Function DeleteObject _
   Lib "gdi32" ( _
   ByVal hObject As Long _
   ) As Long

Private Declare Function GetWindowRect _
   Lib "user32" ( _
   ByVal hwnd As Long, _
   lpRect As RECT _
   ) As Long

Private Declare Function GetWindowLong _
   Lib "user32" _
   Alias "GetWindowLongA" ( _
   ByVal hwnd As Long, _
   ByVal nIndex As Long _
   ) As Long

Private Declare Function GetWindow _
   Lib "user32" ( _
   ByVal hwnd As Long, _
   ByVal wCmd As Long _
   ) As Long
Private Const BITSPIXEL = 12
Private Const GWL_HINSTANCE = (-6)
Private Const CF_BITMAP = 2
Private Const GW_CHILD = 5
Private Const GW_HWNDFIRST = 0
Private Const GW_HWNDNEXT = 2
Private Const DI_MASK = &H1
Private Const DI_IMAGE = &H2
Private Const DI_NORMAL = DI_MASK Or DI_IMAGE
```

Listing 10.3 (Forts.)
Beispiele\10_Grafik\
10_01_Grafik.xls\ufIcon

```
Private Type RECT
        Left As Long
        Top As Long
        Right As Long
        Bottom As Long
End Type

Private mstrPath           As String
Private mlngHandleFrame    As Long
Private mlngInstanceHandle As Long

Private Sub Extract(lngIndex As Long)
   Dim lngFrameDC    As Long
   Dim lngHandleIcon As Long
   Dim udtDimension  As RECT

   ' Icon aus Datei extrahieren
   lngHandleIcon = ExtractIcon(mlngInstanceHandle, _
      mstrPath, lngIndex)

   ' Verlassen, wenn kein Icon verfügbar
   If (lngHandleIcon = 0) Or (lngHandleIcon = 1) Then _
   Exit Sub

   ' Abmessungen des Rahmensteuerelements ermitteln
   GetWindowRect mlngHandleFrame, udtDimension

   ' DC des Rahmensteuerelements ausleihen
   lngFrameDC = GetDC(mlngHandleFrame)

   With udtDimension
      ' Icon malen
      DrawIconEx lngFrameDC, 0, 0, lngHandleIcon, _
         .Right - .Left, .Bottom - .Top, 0, 0, DI_NORMAL
   End With

   ' Icon zerstören
   DestroyIcon lngHandleIcon

   ' DC zurückgeben
   ReleaseDC mlngHandleFrame, lngFrameDC

End Sub

Private Function IconToClip(lngIndex As Long) As Boolean
   Dim lngHandleIcon    As Long
   Dim lngRet           As Long
   Dim lngScrDC         As Long
   Dim lngMemoryDC      As Long
   Dim lngHandleBMP     As Long
   Dim lngHandleBrush   As Long
   Dim udtDimension     As RECT
   Dim lngColorDepth    As Long
   Dim lngDC            As Long
   Dim bytRed           As Byte
   Dim bytGreen         As Byte
   Dim bytBlue          As Byte
```

Listing 10.3 (Forts.)
Beispiele\10_Grafik\
10_01_Grafik.xls\ufIcon

```
' Icon aus Datei extrahieren
lngHandleIcon = ExtractIcon(mlngInstanceHandle, _
   mstrPath, lngIndex)

' Verlassen, wenn kein Icon verfügbar
If (lngHandleIcon = 0) Or (lngHandleIcon = 1) Then _
Exit Function

' DC vom Screen ausleihen
lngScrDC = GetDC(lngDC)

' Einen kompatiblen DC zum Screen anlegen
lngMemoryDC = CreateCompatibleDC(lngScrDC)

With udtDimension

   .Right = 32 ' Breite
   .Bottom = 32 ' Höhe

   ' Hintergrundfarbe weiß
   bytRed = 255
   bytGreen = 255
   bytBlue = 255

   ' Farbtiefe vom Screen ermitteln
   lngColorDepth = GetDeviceCaps(lngScrDC, BITSPIXEL)

   ' Eine Dummy-Bitmap erzeugen
   lngHandleBMP = CreateBitmap(.Right, .Bottom, 1, _
     lngColorDepth, ByVal 0&)

   ' Die Dummy-Bitmap in den Speicher-DC stellen
   lngHandleBMP = SelectObject(lngMemoryDC, lngHandleBMP)

   'Pinsel erzeugen
   lngHandleBrush = CreateSolidBrush( _
      RGB(bytRed, bytGreen, bytBlue))

   'Den Hintergrund in den DC mit dem Pinsel malen
   FillRect lngMemoryDC, udtDimension, lngHandleBrush

   ' Pinsel löschen
   DeleteObject lngHandleBrush

   ' Icon in den DC malen
   DrawIconEx lngMemoryDC, 0, 0, lngHandleIcon, _
      32, 32, 0, 0, DI_NORMAL

End With

' Die erzeugte Dummy-Bitmap in den DC stellen
' und das vom Icon zurückbekommen
lngHandleBMP = SelectObject(lngMemoryDC, lngHandleBMP)

' Erzeugten Speicher-DC löschen
DeleteDC lngMemoryDC
```

Listing 10.3 (Forts.)
Beispiele\10_Grafik\
10_01_Grafik.xls\ufIcon

```
    ' Screen DC zurückgeben
    ReleaseDC lngDC, lngScrDC

    ' Clipboard öffnen
    lngRet = OpenClipboard(0&)

    ' Clipboard leeren
    lngRet = EmptyClipboard()

    'Bitmap in das Clipboard stellen
    lngRet = SetClipboardData(CF_BITMAP, lngHandleBMP)

    ' Bei Erfolg als Funktionsergebnis True zurückgeben
    If lngRet <> 0 Then IconToClip = True

    ' Clipboard schließen
    lngRet = CloseClipboard

    ' Icon zerstören
    DestroyIcon lngHandleIcon

End Function

Private Sub cmdFile_Click()
   Dim strFile       As String
   Dim lngIconsCount As Long
   Const GETCOUNT    As Long = -1

   ' Anfangspfad zur Suche auf das
   ' Windows-Systemverzeichnis setzen
   ChDrive "c:"
   ChDir Environ$("systemroot") & "\system32"

   ' Dialog zur Dateiauswahl
   strFile = Application.GetOpenFilename( _
      "Bibliotheken (*.dll),*.dll," & _
      "Exedateien (*.exe),*.exe," & _
      "Alle Dateien (*.*),*.*")

   ' Keine Datei ausgewählt, dann abbrechen
   If Dir$(strFile, vbReadOnly Or vbSystem Or vbHidden) _
       = "" Then Exit Sub

   ' Anzahl der Icons ermitteln
   lngIconsCount = ExtractIcon(mlngInstanceHandle, _
      strFile, GETCOUNT)

   If lngIconsCount > 0 Then

      ' Icons gefunden
      mstrPath = strFile

      ' mstrPath eintragen
      lblFile.Caption = strFile
```

Listing 10.3 (Forts.)
Beispiele\10_Grafik\
10_01_Grafik.xls\ufIcon

```
        ' Spinbutton Max setzen
        spnPic.Max = lngIconsCount - 1
        spnPic_Change

    Else

        ' Keine Icons gefunden
        mstrPath = ""
        lblFile.Caption = "Kein Icon :  " & strFile
        spnPic.Max = 0

    End If

    ' SpinButton Min und Wert festlegen
    spnPic.Min = 0
    spnPic.Value = 0

    ' Anzahl Icons anzeigen
    lblMax = lngIconsCount

End Sub

Private Function GetFrameHwnd() As Long
    Dim strCaption As String
    Dim strMyGUID  As String
    Dim lngHandle  As Long

    ' Wenn es geht, ohne Klassennamen arbeiten
    ' Hier wird das Fenster mittels einer
    ' eindeutigen Caption gesucht
    strMyGUID = "Disch griige mer aach"

    ' Fenstertext zwischenspeichern
    strCaption = Me.Caption

    ' Fenstertext kurzzeitig ändern
    Me.Caption = strMyGUID

    ' Zugriffsnummer auf die Form
    lngHandle = FindWindow(vbNullString, strMyGUID)

    ' Zugriffsnummer auf den Clientbereich der Form
    lngHandle = GetWindow(lngHandle, GW_CHILD)

    ' Zugriffsnummer Frame
    lngHandle = GetWindow(lngHandle, GW_CHILD)

    ' Fenstertext zurücksetzen
    Me.Caption = strCaption

    ' Handle zurückgeben
    GetFrameHwnd = lngHandle
End Function

Private Sub cmdIconToClip_Click()
    IconToClip spnPic.Value
End Sub
```

Listing 10.3 (Forts.)
Beispiele\10_Grafik\
10_01_Grafik.xls\ufIcon

```
Private Sub cmdIconToXL_Click()

   On Error Resume Next

   If IconToClip(spnPic.Value) Then
      Worksheets(1).Paste
   End If

End Sub

Private Sub spnPic_Change()

   ' Vorher gemaltes Icon im Rahmen löschen
   fraIcon.Repaint

   ' Neues Icon hineinmalen
   Extract spnPic.Value

   ' Textbox mit aktuellem Iconindex füllen
   lblngIndex = spnPic.Value + 1

End Sub

Private Sub UserForm_Initialize()

   'Das Handle des Rahmensteuerelements ermitteln
   mlngHandleFrame = GetFrameHwnd()

   'Das Instanz-Handle der Form ermitteln
   mlngInstanceHandle = GetWindowLong(mlngHandleFrame, _
      GWL_HINSTANCE)

End Sub
```

UserForm_Initialize

Wird dieses Ereignis beim Initialisieren der Form ausgelöst, wird zuerst mit der Funktion `GetFrameHwnd` das Handle des Rahmensteuerelements `fraIcon` gesucht, das sich im Clientfenster der UserForm befindet. Die auf Klassenebene deklarierte Variable `mlngHandleFrame` nimmt das gefundene Fensterhandle auf. Anschließend ermitteln Sie mit der API `GetWindowLong` das Instanzhandle des gefundenen Fensters und machen dieses über die Variable `mlngInstanceHandle` klassenweit verfügbar.

spnPic_Change

Wird dieses Ereignis beim Ändern des Spinbuttons ausgelöst, stoßen Sie das Neuzeichnen des Rahmensteuerelements mit der Methode `Repaint` an. Dadurch wird der sichtbare `DeviceContext` dieses Fensters mit dem Inhalt eines im Hintergrund geführten Speicher-DC überschrieben.

Leider ist der einzige DC, an dessen Handle man herankommt, der sichtbare, deshalb kann man auch nur diesen manipulieren. Das bringt einige Nachteile, die ich nicht verschweigen möchte. Beim Neuzeichnen, sei es durch die

Methode `Repaint` oder automatisch ausgelöst durch das Überstreichen eines anderen Fensters, wird die intern im Speicher mitgeführte Zeichenfläche (DC) als Bildquelle benutzt und alles, was mühselig in den sichtbaren DC gemalt wurde, rigoros überschrieben.

Das hat aber den positiven Nebeneffekt, dass das Löschen nicht sehr schwer fällt. Das Rahmensteuerelement ist übrigens eines der wenigen echten Fenster in der Gruppe der Steuerelemente von Excel, nur deshalb habe ich es auch benutzt. Die meisten anderen Steuerelemente sind dagegen nur grafische Elemente ohne Fensterhandle und DC.

Anschließend wird die Prozedur `Extract` aufgerufen, welche das als Index übergebene Icon in das Rahmensteuerelement zeichnet. Der Label `lblIndex` zeigt anschließend die Nummer des Icons an.

cmdIconToXL_Click

Beim Klick auf den Button `cmdIconToXl` wird die Funktion `IconToClip` aufgerufen, die das als Index übergebene Icon als Bitmap ins Clipboard (Zwischenablage) befördert. War die Funktion erfolgreich, was Sie an dem Rückgabewert der Funktion erkennen können, wird mit der Methode `Paste` der Inhalt in das Tabellenblatt eins befördert. Die Fehlerbehandlung habe ich aus keinem konkretem Anlass eingefügt, sondern nur deshalb, weil ich in anderen Fällen schon einmal Probleme mit der Methode `Paste` hatte.

cmdIconToClip_Click

Beim Klick auf den Button `cmdIconToClip` wird die Funktion `IconToClip` aufgerufen, die das als Index übergebene Icon als Bitmap ins Clipboard befördert.

GetFrameHwnd

Wird diese Funktion aufgerufen, liefern Sie als Funktionsergebnis das Handle des Rahmensteuerelements `fraIcon` zurück, das sich im Clientfenster der UserForm befindet. Dazu wird erst einmal mit `FindWindow` das Handle der UserForm über eine eindeutige Caption ermittelt. Wenn Sie dieses Handle haben, suchen Sie mit der API `GetWindow` und dem mit übergebenen Parameter `GW_CHILD` das Kindfenster der UserForm. Dieses erst enthält als weiteres Kindfenster das Rahmensteuerelement, weshalb noch einmal `GetWindow` mit dem Parameter `GW_CHILD` eingesetzt wird.

cmdFile_Click

Beim Klick auf den Button `cmdFile` wird der Dialog `GetOpenFilename` zur Dateiauswahl aufgerufen. Vorher wird mit `ChDir` das aktuelle Verzeichnis auf das `System32`-Verzeichnis im Windowsordner gesetzt, damit der Dialog in diesem Verzeichnis beginnt. Anschließend wird mit der Funktion `ExtractIcon` die Anzahl der Icons in dieser Datei ermittelt und falls mindestens eins vorhanden ist, die klassenweit gültige Variable `mstrPath` mit dem Pfad inklusive Dateinamen gefüllt.

Anschließend erhält das Label `lblFile` als Caption den Pfad und der Spinbutton `spnPic` bekommt als Maximumwert die Anzahl der Icons in der Datei zugewiesen. Auch das Label `lblMax` soll die Anzahl der Icons anzeigen und bekommt diesen als Caption zugewiesen. Damit auch gleich nach der Dateiauswahl ein Icon sichtbar wird, rufen Sie `spnPic_Change` auf.

Die Datei *user32.dll* enthält auf jeden Fall ein paar Icons.

IconToClip

Diese Funktion wird aufgerufen, um ein Icon aus einer Datei zu extrahieren und ins Clipboard als Bitmap zu bringen.

Dazu benötigen Sie zuerst einmal ein Handle auf das gewünschte Icon. Das erledigt die API-Funktion `ExtractIcon`, der als erster Parameter das Instanzhandle der aktuellen Anwendung übergeben wird. Als zweiter Parameter wird die Datei inklusive komplettem Pfad und als dritter Parameter die Nummer des Icons in Form eines Longwertes übergeben. Als Funktionsergebnis erhalten Sie das Handle, im Fehlerfall wird Null zurückgeliefert.

Als Nächstes benötigen Sie einen `DeviceContext`, der mit dem `Screen` kompatibel ist. Dazu leihen Sie sich den `DC` vom Screen aus, indem Sie die Funktion `GetDC` mit dem auf null gesetzten Parameter `hwnd` aufrufen. Um das zu realisieren, bleibt die Variable `hDC` auf null.

An die API-Funktion `CreateCompatibleDC` wird anschließend der ausgeliehene ScreenDC übergeben und diese liefert ein Handle auf den erzeugten Speicher-DC zurück, der nun mit dem Screen kompatibel ist. Diese grafische Oberfläche ist nicht sichtbar und existiert nur im Speicher, besitzt aber die gleichen Eigenschaften wie der sichtbare Screen.

Um diesen DC mit Leben zu füllen, stellen Sie mit `SelectObject` eine Bitmap hinein. Diese Bitmap wird vorher mit `CreateBitmap` erzeugt, hat eine Abmessung von 32 x 32 Pixel und benötigt so viele Bits für die Farbinformationen, wie in der Variablen `lngColorDepth` steht. Die aktuelle Farbtiefe des `ScreenDC` wird mit der API `GetDeviceCaps` und dem auf den `ScreenDC` gesetzten Parameter `hdc` ermittelt. Als Parameter `nIndex` wird dabei die Konstante `BITSPIXEL` verwendet.

Um den Hintergrund in einer bestimmten Farbe darzustellen, erzeugen Sie einen soliden Pinsel in der gewünschten Farbe, was mit `CreateSolidBrush` geschieht. Dann malen Sie mit diesem Pinsel und mithilfe der Funktion `FillRect` ein gefülltes Rechteck in den DC, das die Größe des Bitmaps hat. Anschließend können Sie den Pinsel mit `DeleteObject` wieder löschen.

Schließlich wird noch mit `DrawIconEx` das eigentliche Icon in der gewünschten Größe in den DC gemalt. Als Sie das erzeugte Bitmap mit `SelectObject` in den `DC` gestellt haben, wurde das ursprünglich dort vorhandene Objekt als Handle zurückgeliefert. Dies müssen Sie nun mit `SelectObject` wieder rückgängig machen, indem Sie das alte Objekt dort wieder hineinstellen. Dann können Sie den Speicher-DC wieder löschen und anschließend wird der ausgeliehene `ScreenDC` zurückgegeben.

Nun muss nur noch die Bitmap in die Zwischenablage geschoben werden. Dazu öffnen Sie mit `OpenClipboard` die Zwischenablage, leeren diese mit `EmptyClipboard`, schieben mit `SetClipboardData` die Bitmap dort hinein und schließen das Clipboard mit `CloseClipboard` wieder.

Anschließend wird mit `DestroyIcon` das extrahierte Icon zerstört und der Rückgabewert der Funktion auf Wahr gestellt.

Extrahieren

Diese Prozedur wird aufgerufen, um ein Icon aus einer Datei zu extrahieren und im Rahmensteuerelement darzustellen.

Dazu benötigen Sie erst einmal ein Handle auf das gewünschte Icon. Das erledigt die Funktion `ExtractIcon`, der als erster Parameter das Instanzhandle der aktuellen Anwendung übergeben wird. Als zweiter Parameter wird die Datei inklusive komplettem Pfad und als dritter Parameter die Nummer des Icons in Form eines Longwertes übergeben. Als Funktionsergebnis bekommen Sie das Handle auf das Icon, im Fehlerfall wird Null zurückgeliefert.

Zum Malen benötigen Sie als Zeichenfläche den `Devicekontext` des Rahmensteuerelements. Dazu leihen Sie sich diesen `DC` mit der Funktion `GetDC` und dem auf diesen Fensterhandle gesetzten Parameter `hwnd` aus. Anschließend wird mit der Funktion `DrawIconEx` das Icon in der Größe des Steuerelements in den `DC` gemalt. Diese Größe wurde vorher mittels `GetWindowRect` ermittelt. Zum Schluss wird mit `DestroyIcon` das Icon zerstört und mit `ReleaseDC` der DC an den ursprünglichen Besitzer zurückgegeben.

10.4 Bildschirmauflösung ändern

Mithilfe des nachfolgenden Beispieles können Sie die Auflösung, Farbtiefe und Frequenz programmgesteuert ändern. Sie sollten sich aber darüber im Klaren sein, dass das Ändern dieser Einstellungen ausschließlich dem jeweiligen Benutzer überlassen bleiben sollte. Bei Spielen könnte das programmgesteuerte Verändern der Auflösung noch akzeptiert werden, aber auch nur, wenn die Auflösung nach unten geändert wird und am Schluss die alten Einstellungen zurückgesetzt werden, aber selbst dann sollten Sie sich beim Anwender rückversichern.

Ein Programm, das solche Einstellungen ungefragt ändert, würde bei mir sofort von der Platte fliegen. Bei einer Beschädigung von Hardware kämen noch die Schadensersatzansprüche hinzu. Wenn Sie in einem Dialog aber eindeutig darauf hinweisen, spricht andererseits nichts dagegen, dem Benutzer die Arbeit abzunehmen, die er sonst wesentlich umständlicher selbst durchführen würde.

Achtung

Jede Änderung der Auflösung nach oben oder die Erhöhung der Bildwiederholfrequenz kann einen angeschlossenen Monitor zerstören, selbst wenn die Grafikkarte diese Einstellungen problemlos unterstützt.

Manche Einstellungen verlangen auch einen Neustart des Rechners. Das Herunterfahren des Rechners ist unter NT und deren Nachfolgern aber nicht jedem Prozess erlaubt, weshalb man sich diese Rechte erst einmal für den laufenden Prozess verschaffen muss. Voraussetzung dazu ist allerdings, dass der gerade angemeldete Benutzer selbst ausreichende Berechtigungen dazu hat.

Um das Abmelden, Herunterzufahren, Neustarten und Auszuschalten zu realisieren, habe ich eine kleine Klasse geschrieben, die das alles kapselt. Das hat den schönen Nebeneffekt, dass Sie diese Klasse auch in anderen Anwendungen einsetzen können.

Nachfolgend der Code für das Auslesen von Informationen über die Bildschirmeinstellungen und zum programmgesteuerten Ändern derselben.

10.4.1 Die UserForm

Legen Sie dazu eine UserForm an, fügen Sie drei CommandButtons, eine Listbox und fünf Labels ein. Den Aufbau und die Namen entnehmen Sie der Abbildung 10.3.

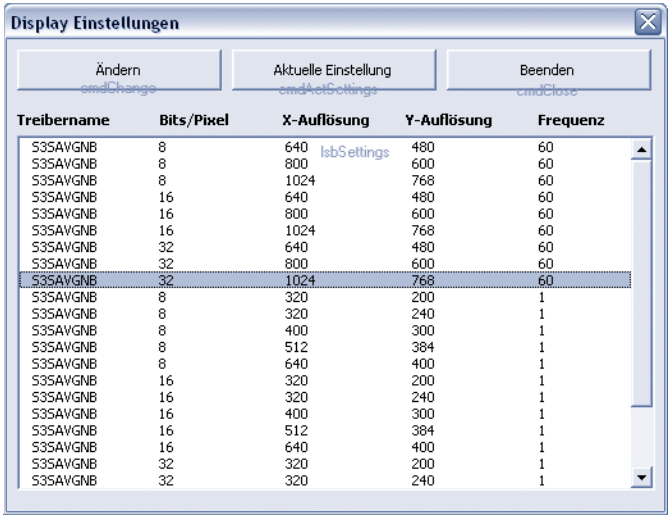


Abbildung 10.3
Änderung der Bildschirmeinstellungen (mit Namen der Steuerelemente)

Listing 10.4
Beispiele\10_Grafik\
10_01_Displaysettings.xls\
ufDisplay

```
Private Declare Function EnumDisplaySettings _
   Lib "user32" Alias "EnumDisplaySettingsA" ( _
   ByVal lpszDeviceName As String, _
   ByVal iModeNum As Long, _
   lpDevMode As Any _
   ) As Boolean

Private Declare Function ChangeDisplaySettings _
   Lib "user32" Alias "ChangeDisplaySettingsA" ( _
   lpDevMode As Any, _
   ByVal dwFlags As Long _
   ) As Long

Private Declare Function GetDeviceCaps _
   Lib "gdi32" ( _
   ByVal hdc As Long, _
   ByVal nIndex As Long _
   ) As Long

Private Declare Function GetDC _
   Lib "user32" ( _
    ByVal hwnd As Long _
    ) As Long

Private Declare Function ReleaseDC _
   Lib "user32" ( _
    ByVal hwnd As Long, _
    ByVal hdc As Long _
    ) As Long

Private Const CCDEVICENAME = 32
Private Const CCFORMNAME = 32

Private Const DM_BITSPERPEL = &H40000
Private Const DM_PELSWIDTH = &H80000
Private Const DM_PELSHEIGHT = &H100000
Private Const DM_DISPLAYFREQUENCY = &H400000

Private Const DISP_CHANGE_SUCCESSFUL = 0
Private Const DISP_CHANGE_RESTART = 1

Private Const CDS_UPDATEREGISTRY = &H1
Private Const CDS_TEST = &H4

Private Type DEVMODE
    dmDeviceName As String * CCDEVICENAME
    dmSpecVersion As Integer
    dmDriverVersion As Integer
    dmSize As Integer
    dmDriverExtra As Integer
    dmFields As Long
    dmOrientation As Integer
    dmPaperSize As Integer
    dmPaperLength As Integer
    dmPaperWidth As Integer
    dmScale As Integer
    dmCopies As Integer
```

Listing 10.4 (Forts.)
Beispiele\10_Grafik\
10_01_Displaysettings.xls\
ufDisplay

```
    dmDefaultSource As Integer
    dmPrintQuality As Integer
    dmColor As Integer
    dmDuplex As Integer
    dmYResolution As Integer
    dmTTOption As Integer
    dmCollate As Integer
    dmFormName As String * CCFORMNAME
    dmUnusedPadding As Integer
    dmBitsPerPel As Integer
    dmPelsWidth As Long
    dmPelsHeight As Long
    dmDisplayFlags As Long
    dmDisplayFrequency As Long
End Type

Private Const DRIVERVERSION = 0
Private Const HORZRES As Long = 8
Private Const VERTRES As Long = 10
Private Const BITSPIXEL As Long = 12
Private Const VREFRESH As Long = 116

Private mudtDevmodeAkt As DEVMODE
Private mlngActSettings As Long

Private Sub cmdChange_Click()
   ChangeSettings
End Sub

Private Sub cmdActSettings_Click()

   'Listindex auf die aktuelle Einstellung
   lsbSettings.ListIndex = mlngActSettings

End Sub

Private Sub cmdClose_Click()

   Unload Me

End Sub

Private Sub lsbSettings_DblClick( _
   ByVal Cancel As MSForms.ReturnBoolean)

   ChangeSettings

End Sub

Private Sub UserForm_Initialize()
   Dim udtDev                    As DEVMODE
   Dim lngRet                    As Long
   Dim avarDummy()               As Variant
   Dim colDummy                  As New Collection
   Dim varItem                   As Variant
   Dim avarActSettings(0 To 4)   As Variant
```

Listing 10.4 (Forts.)
Beispiele\10_Grafik\
10_01_Displaysettings.xls\
ufDisplay

```
Dim lngScrDC                    As Long
Dim i                           As Long

ReDim avarDummy(0 To 4)

lngScrDC = GetDC(0&)  ' Screen DC ausleihen

' Aktuelle Einstellungen vom Screen holen
avarActSettings(1) = GetDeviceCaps(lngScrDC, BITSPIXEL)
avarActSettings(2) = GetDeviceCaps(lngScrDC, HORZRES)
avarActSettings(3) = GetDeviceCaps(lngScrDC, VERTRES)
avarActSettings(4) = GetDeviceCaps(lngScrDC, VREFRESH)

ReleaseDC 0, lngScrDC 'Screen DC zurückgeben

Do

   ' Nacheinander alle möglichen Einstellungen holen
   ' Beide Varianten möglich, bei der ersten können
   ' noch andere Displayeinstellungen geholt werden
   lngRet = EnumDisplaySettings("\\.\Display1", i, udtDev)
   lngRet = EnumDisplaySettings(vbNullString, i, udtDev)

   ' Schleife verlassen, wenn keine geliefert wurden
   If lngRet = 0 Then Exit Do

   ' Den Index erhöhen
   i = i + 1

   With udtDev

      ' Einstellungen in ein eindimensionales Array
      avarDummy(0) = Left(.dmDeviceName, InStr(1, _
         .dmDeviceName, Chr(0)) - 1)
      avarDummy(1) = .dmBitsPerPel
      avarDummy(2) = .dmPelsWidth
      avarDummy(3) = .dmPelsHeight
      avarDummy(4) = .dmDisplayFrequency

      'Aktuelle Einstellungen ermitteln
      If avarDummy(1) = avarActSettings(1) Then _
         If avarDummy(2) = avarActSettings(2) Then _
            If avarDummy(3) = avarActSettings(3) Then _
               If avarDummy(4) = avarActSettings(4) Then _
                  mudtDevmodeAkt = udtDev

      ' Das Array einer Collection hinzufügen
      colDummy.Add avarDummy

   End With
Loop

' Array für die Listbox dimensionieren. Die zweite Dimension
' legt die Anzahl der Spalten fest. ColumnCount muss auch
' entsprechend gesetzt sein
ReDim avarDummy(0 To colDummy.Count - 1, 0 To 4)
```

Listing 10.4 (Forts.)
Beispiele\10_Grafik\
10_01_Displaysettings.xls\
ufDisplay

```
    i = 0
    'Alle Einstellmöglichkeiten durchlaufen
    For Each varItem In colDummy

        'Werte ins Array eintragen
        avarDummy(i, 0) = varItem(0)
        avarDummy(i, 1) = varItem(1)
        avarDummy(i, 2) = varItem(2)
        avarDummy(i, 3) = varItem(3)
        avarDummy(i, 4) = varItem(4)

        'Aktuellen Listindex ermitteln
        If avarDummy(i, 1) = avarActSettings(1) Then _
           If avarDummy(i, 2) = avarActSettings(2) Then _
              If avarDummy(i, 3) = avarActSettings(3) Then _
                 If avarDummy(i, 4) = avarActSettings(4) Then _
                    mlngActSettings = i
        i = i + 1
    Next

    'Daten an die Listbox übergeben
    lsbSettings.List() = avarDummy

    'Listindex auf die aktuelle Einstellung
    lsbSettings.ListIndex = mlngActSettings

End Sub

Sub ChangeSettings()
    Dim udtDev As DEVMODE
    Dim lngRet As Long
    Dim strMsg As String

    Dim objShutDown As clsWinEnd

    'Abfrage, ob wirklich geändert werden soll
    strMsg = "Sind sie sicher, dass sie" & _
       " die Einstellungen auf" & vbCrLf
    strMsg = strMsg & "Bits/Pixel    :" & _
       lsbSettings.List(lsbSettings.ListIndex, 1) & vbCrLf
    strMsg = strMsg & "X-Res         :" & _
       lsbSettings.List(lsbSettings.ListIndex, 2) & vbCrLf
    strMsg = strMsg & "Y-Res         :" & _
       lsbSettings.List(lsbSettings.ListIndex, 3) & vbCrLf
    strMsg = strMsg & "Frequenz   :" & _
       lsbSettings.List(lsbSettings.ListIndex, 4) & vbCrLf
    strMsg = strMsg & "ChangeSettings wollen?"

    If MsgBox(strMsg, vbOKCancel, "Bildschirmeinstellungen") <> _
       vbOK Then Exit Sub

    'Es soll geändert werden
    With mudtDevmodeAkt

        'Angeben, was geändert werden soll
        .dmFields = DM_PELSWIDTH Or DM_PELSHEIGHT Or _
           DM_BITSPERPEL Or DM_DISPLAYFREQUENCY
```

Listing 10.4 (Forts.)
Beispiele\10_Grafik\
10_01_Displaysettings.xls\
ufDisplay

```
        'Auflösung X
        .dmPelsWidth = lsbSettings.List( _
           lsbSettings.ListIndex, 2)

        'Auflösung Y
        .dmPelsHeight = lsbSettings.List( _
           lsbSettings.ListIndex, 3)

        'Bits/Pixel
        .dmBitsPerPel = lsbSettings.List( _
           lsbSettings.ListIndex, 1)

        'Frequenz
        .dmDisplayFrequency = lsbSettings.List( _
           lsbSettings.ListIndex, 4)

        'Änderung durchführen, aber vorher testen
        lngRet = ChangeDisplaySettings(mudtDevmodeAkt, CDS_TEST)

       Select Case lngRet

          Case DISP_CHANGE_SUCCESSFUL
             ' Erfolgreich geändert
             lngRet = ChangeDisplaySettings( _
                udtDev, CDS_UPDATEREGISTRY)
             MsgBox "Geändert", vbOKOnly, "Erfolg"
                mlngActSettings = lsbSettings.ListIndex

          Case DISP_CHANGE_RESTART
             ' Änderung erfordert Neustart
             lngRet = MsgBox("Neustarten?", vbYesNo, _
                "Änderung wird erst bei Neustart wirksam")

             If lngRet = vbYes Then
                ' Neustart wurde erlaubt

                '  Objekt objShutDown (clsWinEnde) bemühen
                Set objShutDown = New clsWinEnd

                '  Neustarten
                objShutDown.Neustarten

             End If

          Case Else

             MsgBox "Nicht geändert", vbOKOnly, "Fehler"

       End Select

    End With

End Sub
```

cmdChange_Click

Bei einem Klick auf den Button `cmdChange` wird die Prozedur `ChangeSettings` aufgerufen.

cmdActSettings_Click

Bei einem Klick auf den Button `cmdActSettings` wird der Listindex der Listbox `lsbSettings` auf den Eintrag mit den momentan herrschenden Einstellungen gesetzt.

cmdClose_Click

Bei einem Klick auf den Button `cmdClose` wird die UserForm entladen.

lsbSettings_DblClick

Bei einem Doppelklick in die Listbox `lsbSettings` wird die Prozedur `ChangeSettings` aufgerufen.

UserForm_Initialize

Die Prozedur `UserForm_Initialize` dient dazu, beim Initialisieren der Form eine Liste mit allen möglichen Bildschirmeinstellungen zu erzeugen und diese in die Listbox einzutragen.

Um erst einmal die aktuellen Einstellungen auszulesen, leihen Sie sich den `Devicekontext` (`DC`) des Screens mit `GetDC` aus und holen sich mittels der API `GetDeviceCaps` die gewünschten Einstellungen. Die API `GetDeviceCaps` benötigt als Argument den `DC`, dessen Eigenschaften ausgelesen werden sollen und als zweites Argument einen Wert, der darüber entscheidet, welche Eigenschaft ausgelesen wird. Den ausgeliehenen DC können Sie anschließend sofort zurückgeben.

Mittels `EnumDisplaySettings` können Sie nun nacheinander alle möglichen Einstellungen der Grafikkarte auslesen. Dazu wird dieser Funktion ein Index in die Liste der möglichen Einstellung übergeben. In einer `Do-Loop`-Schleife erhöhen Sie so lange den Index und lesen die Einstellungen aus, bis von der Funktion eine Null zurückgeliefert wird. Das ist das Zeichen für einen Fehler.

Die Einstellungen selbst stecken jeweils in einer Struktur des Typs `DEVMODE`, die als Parameter mit übergeben wurde. Ein eindimensionales Array nimmt diese Einstellungen anschließend auf und wird als neues Element in einer Collection gespeichert. Bei Übereinstimmung mit den momentanen Einstellungen wird die `DEVMODE`-Struktur als klassenweit gültige Struktur mit dem Namen `mudtDevmodeAkt` gespeichert.

Wenn Sie alle Einstellungen ausgelesen haben, wird daraus ein zweidimensionales Array erzeugt, das als Eigenschaft `List` an die Listbox übergeben wird. Die Zeilen in der Listbox werden durch die erste Dimension, die Spalten durch die zweite Dimension des Arrays bestimmt. Anschließend muss die Eigenschaft `ColumnCount` auf die Spaltenanzahl gesetzt werden, die nachher angezeigt werden sollen, in diesem Fall sind das fünf.

Schließlich wird noch der Eintrag in der Listbox ausgewählt, der den aktuellen Einstellungen entspricht. Dieser Index wurde ermittelt, indem beim Erzeugen des zweidimensionalen Arrays die aktuelle Einstellung mit der gerade bearbeiteten verglichen wurde. Bei Übereinstimmung wird die Variable `mlngActSettings` auf den Index gesetzt.

ChangeSettings

Die Prozedur `ChangeSettings` dient dazu, die Einstellungen für das Display zu ändern. Nach einer Rückfrage, ob Sie die Einstellungen tatsächlich ändern wollen, werden einzelne Elemente der klassenweit gültigen Struktur `mudtDevmodeAkt` so geändert, dass sie die neuen Werte enthalten. Das ist die Auflösung in x- und y-Richtung – die Farbtiefe und die Frequenz –, die anderen Elemente bleiben unangetastet. Als Quelle dafür wird die markierte Zeile in der Listbox verwendet, wobei die Werte in den Spalten 2 bis 5 benutzt werden.

Mit der API `ChangeDisplaySettings` wird die Änderung durchgeführt. Dieser Funktion wird die modifizierte `DEVMODE`-Struktur mit der Konstanten `CDS_Test` übergeben, die bewirkt, dass die neue Einstellung erst getestet wird.

Als Ergebnis wird ein Longwert zurückgeliefert, der Auskunft darüber gibt, ob die Funktion erfolgreich war und ob der Rechner neu gestartet werden muss. Wurde `DISP_CHANGE_SUCCESSFUL` zurückgeliefert, kann die Änderung in die Registry eingetragen werden, was durch den Aufruf von `ChangeDisplaySettings` mit dem auf `CDS_UPDATEREGISTRY` gesetzten zweiten Parameter geschieht.

Ist ein Neustart erforderlich – in diesem Fall wird `DISP_CHANGE_RESTART` zurückgeliefert –, erzeugen Sie eine Instanz der Klasse `clsWinEnd` und rufen die Methode `Reboot` auf.

10.4.2 Klasse clsWinEnd

In eine Klasse (Listing 10.5) mit Namen `clsWinEnd`, die zum Herunterfahren und Neustarten des Rechners dient, gehört folgender Code:

Listing 10.5
Beispiele\10_Grafik\
10_01_Displaysettings.xls\
clsWinEnd

```
Private Declare Function GetCurrentProcess _
   Lib "kernel32" () As Long

Private Declare Function OpenProcessToken _
   Lib "advapi32" ( _
   ByVal ProcessHandle As Long, _
   ByVal DesiredAccess As Long, _
   TokenHandle As Long _
   ) As Long

Private Declare Function LookupPrivilegeValue _
   Lib "advapi32" Alias "LookupPrivilegeValueA" ( _
   ByVal lpSystemName As String, _
   ByVal lpName As String, _
   lpLuid As LUID _
   ) As Long
```

Listing 10.5 (Forts.)
Beispiele\10_Grafik\
10_01_Displaysettings.xls\
clsWinEnd

```
Private Type OSVERSIONINFO
   dwOSVersionInfoSize As Long
   dwMajorVersion As Long
   dwMinorVersion As Long
   dwBuildNumber As Long
   dwPlatformId As Long
   szCSDVersion As String * 128
End Type

Private Declare Function GetVersionEx _
   Lib "kernel32" Alias "GetVersionExA" ( _
   lpVersionInformation As OSVERSIONINFO _
   ) As Long

Private Declare Function AdjustTokenPrivileges _
   Lib "advapi32" ( _
   ByVal TokenHandle As Long, _
   ByVal DisableAllPrivileges As Long, _
   NewState As TOKEN_PRIVILEGES, _
   ByVal BufferLength As Long, _
   PreviousState As TOKEN_PRIVILEGES, _
   ReturnLength As Long _
   ) As Long

Private Declare Function ExitWindowsEx _
   Lib "user32" ( _
   ByVal uFlags As Long, _
   ByVal dwReserved As Long _
   ) As Long

Private Const TOKEN_ADJUST_PRIVILEGES = &H20
Private Const TOKEN_QUERY = &H8

Private Type LUID
   LowPart As Long
   HighPart As Long
End Type

Private Type TOKEN_PRIVILEGES
    PrivilegeCount As Long
    TPLuid As LUID
    Attributes As Long
End Type

Private Const EWX_FORCE = 4
Private Const EWX_FORCEIFHUNG = &H10
Private Const EWX_LOGOFF = 0
Private Const EWX_POWEROFF = &H8
Private Const EWX_REBOOT = 2
Private Const EWX_SHUTDOWN = 1

Private Const SE_SHUTDOWN_NAME = "SeShutdownPrivilege"
Private Const SE_PRIVILEGE_ENABLED = &H2

Private mlngForce As Long
```

Listing 10.5 (Forts.)
Beispiele\10_Grafik\
10_01_Displaysettings.xls\
clsWinEnd

```
Private Sub ChangeShutdownPrivileges()
   Dim udtTPOld         As TOKEN_PRIVILEGES
   Dim udtTPNew         As TOKEN_PRIVILEGES
   Dim lngCurToken      As Long
   Dim lngNeededBuffer  As Long
   Dim udtLUID          As LUID

   ' Handle zum eigenen Prozesstoken holen,
   ' das die Zugriffsberechtigungen regelt
   OpenProcessToken GetCurrentProcess(), ( _
      TOKEN_ADJUST_PRIVILEGES Or TOKEN_QUERY _
      ), lngCurToken

   With udtTPNew

      ' Die LUID für SE_SHUTDOWN_NAME auslesen
      LookupPrivilegeValue vbNullString, _
         SE_SHUTDOWN_NAME, udtLUID

      ' Und in die Struktur eintragen
      .TPLuid = udtLUID

      .PrivilegeCount = 1


      ' Gewünschtes Privileg für diesen Prozess eintragen
      .Attributes = SE_PRIVILEGE_ENABLED

   End With

   ' Shutdown-Privileg für diesen Prozess setzen
   AdjustTokenPrivileges lngCurToken, False, udtTPNew, _
      Len(udtTPOld), udtTPOld, lngNeededBuffer

End Sub

Private Sub DoIt(lngAction As Long)

   If Not (IsWindows9X) Then
      ' NT-Familie
      ChangeShutdownPrivileges
   End If

   ExitWindowsEx lngAction Or mlngForce, 0&
End Sub

Private Function IsWindows9X()
   Dim udtOSVERSION As OSVERSIONINFO

   With udtOSVERSION

      ' Größe der Struktur
      .dwOSVersionInfoSize = Len(udtOSVERSION)

      ' Infos über die Version holen
      GetVersionEx udtOSVERSION
```

Listing 10.5 (Forts.)
Beispiele\10_Grafik\
10_01_Displaysettings.xls\
clsWinEnd

```
        ' Ergebnis zurückliefern
        If .dwPlatformId = 1 Then _
            IsWindows9X = True

    End With
End Function

Public Sub LogOff()
    DoIt EWX_LOGOFF
End Sub

Public Sub PowerOff()
    DoIt EWX_POWEROFF
End Sub

Public Sub ShutDown()
    DoIt EWX_SHUTDOWN
End Sub

Public Sub Reboot()
    DoIt EWX_REBOOT
End Sub

Public Property Get KillProcess() As Boolean
    KillProcess = mlngForce And EWX_FORCE
End Property
Public Property Let KillProcess( _
    ByVal vNewValue As Boolean)
    If vNewValue Then
        mlngForce = mlngForce Or EWX_FORCE
    Else
        mlngForce = mlngForce And Not (EWX_FORCE)
    End If
End Property

Public Property Get KillHangingProcess() As Boolean
    KillHangingProcess = mlngForce And EWX_FORCEIFHUNG
End Property
Public Property Let KillHangingProcess( _
    ByVal vNewValue As Boolean)
    If vNewValue Then
        mlngForce = mlngForce Or EWX_FORCEIFHUNG
    Else
        mlngForce = mlngForce And (Not EWX_FORCEIFHUNG)
    End If
End Property
```

LogOff

Die Methode `LogOff` dient dazu, einen angemeldeten Benutzer abzumelden. Dabei wird die Prozedur `DoIt` aufgerufen und die Konstante `EWX_LOGOFF` mit übergeben.

PowerOff

Die Methode `PowerOff` dient dazu, einen Rechner herunterzufahren und auszuschalten, wenn es die Hardware zulässt. Dabei wird die Prozedur `DoIt` aufgerufen und die Konstante `EWX_POWEROFF` mit übergeben.

ShutDown

Die Methode `ShutDown` dient dazu, einen Rechner herunterzufahren. Dabei wird die Prozedur `DoIt` aufgerufen und die Konstante `EWX_SHUTDOWN` mit übergeben.

Reboot

Die Methode `Reboot` dient dazu, einen Rechner herunterzufahren und neu zu starten. Dabei wird die Prozedur `DoIt` aufgerufen und die Konstante `EWX_REBOOT` mit übergeben.

KillProcess

Wird die Eigenschaft `KillProcess` gesetzt, wird zum Beenden anderer Prozesse nicht die Message `WM_QUERYENDSESSION` und `WM_ENDSESSION` abgesetzt, sodass Datenverlust entstehen kann. Die klassenweit gültige Variable `mlngForce` enthält anschließend das gesetzte Flag `EWX_FORCE`. Wenn die Eigenschaft auf `False` gesetzt wird, wird das Flag gelöscht.

KillHangingProcess

Wird die Eigenschaft `KillHangingProcess` gesetzt, werden hängende Prozesse abgeschossen, wenn sie nicht auf die Message `WM_QUERYENDSESSION` und `WM_ENDSESSION` reagieren. Die klassenweit gültige Variable `mlngForce` enthält anschließend das gesetzte Flag `EWX_FORCEIFHUNG`. Wenn die Eigenschaft auf `False` gesetzt wird, wird das Flag gelöscht.

DoIt

Die Prozedur `DoIt` dient dazu, die API-Funktion `ExitWindowsEx` auszuführen. Da der ausführende Prozess unter NT und dessen Nachfolgern auch die Berechtigung dafür besitzen muss, wird mit der Funktion `IsWindows9X` überprüft, welches Betriebssystem läuft. Läuft 9X, wird die Funktion `ExitWindowsEx` sofort ausgeführt, im anderen Fall werden erst mit der Prozedur `ChangeShutdownPrivileges` die Privilegien geändert.

IsWindows9X

Die Funktion `IsWindows9X` dient dazu, einen Wahrheitswert zu liefern, der angibt, ob es sich bei dem aktuell laufenden System um ein 9X-System handelt. Dafür wird die API-Funktion `GetVersionEx` benutzt. Diese API-Funktion füllt die Struktur `OSVERSIONINFO` aus, welche nach dem Aufruf die eigentlichen Informationen zum Betriebssystem liefert. Ist in dieser Struktur das Element

dwPlatformId auf eins gesetzt, handelt es sich um ein 9X-System und die Funktion gibt den Wahrheitswert TRUE zurück.

ChangeShutdownPrivileges

Die Prozedur ChangeShutdownPrivileges dient dazu, das Privileg des aktuellen Prozesses so zu ändern, dass ein Shutdown erlaubt ist. Dazu wird sich mit der Funktion OpenProcessToken ein Handle mit den Berechtigungen TOKEN_ADJUST_PRIVILEGES und TOKEN_QUERY (Anpassen und Abfragen) auf den Prozesstoken geholt. Das dazu notwendige Prozesshandle holen Sie sich mit der Funktion GetCurrentProcess, welche ein in diesem Prozess gültiges Handle liefert.

Anschließend holen Sie sich mit der API-Funktion LookupPrivilegeValue und dem auf SE_SHUTDOWN_NAME gesetzten Parameter lpName eine Struktur LUID, die nach der Rückkehr eine eindeutige 64-Bit-ID des Shutdown-Privilegs enthält. In der Struktur TOKEN_PRIVILEGES wird die LUID an das Element TPLuid übergeben. Danach wird das Element Attributes mit Wert SE_PRIVILEGE_ENABLED gefüllt, um anzugeben, dass das Privileg gesetzt wird. Mittels der API-Funktion AdjustTokenPrivileges wird die Änderung des Privilegs vollzogen.

10.5 Spielkarten

Manche Spiele, darunter auch solche, die zusammen mit dem Betriebssystem installiert werden, verwenden Spielkarten. Diese Karten stecken in einer Datei namens *Cards.dll*, die auf Ihrem Rechner vorhanden sein dürfte, zumindest wenn Sie sich in der Microsoft-Welt bewegen. Aus dieser Bibliothek müssen sie nur herausgekitzelt (Listing 10.6) und als *Bitmap*-Datei gespeichert oder in das Tabellenblatt eingefügt werden.

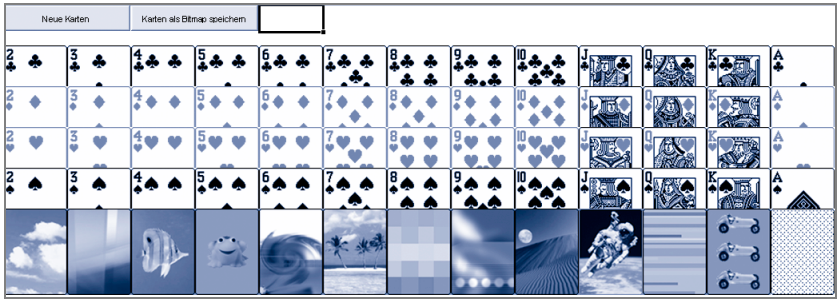


Abbildung 10.4
Karten in der cards.dll

Zum Auswählen des Verzeichnisses, das als Zielverzeichnis für die exportierten Spielkarten verwendet werden soll, wird ein Dialog (VBGetFolder) zur Verzeichnisauswahl gestartet. Der Code und die Informationen dazu sind in Kapitel 6 zu finden.

Listing 10.6
Beispiele\10_Grafik\
10_01_Cards.xls\mdlCards

```
Private Type BITMAPINFOHEADER
   biSize As Long
   biWidth As Long
   biHeight As Long
   biPlanes As Integer
   biBitCount As Integer
   biCompression As Long
   biSizeImage As Long
   biXPelsPerMeter As Long
   biYPelsPerMeter As Long
   biClrUsed As Long
   biClrImportant As Long
End Type

Private Type BITMAPFILEHEADER
   bfType As Integer
   bfSize As Long
   bfReserved1 As Integer
   bfReserved2 As Integer
   bfOffBits As Long
End Type

Private Type BITMAP
   bmType As Long
   bmWidth As Long
   bmHeight As Long
   bmWidthBytes As Long
   bmPlanes As Integer
   bmBitsPixel As Integer
   bmBits As Long
End Type

Private Type RGBQUAD
   rgbBlue As Byte
   rgbGreen As Byte
   rgbRed As Byte
   rgbReserved As Byte
End Type

Private Type BITMAPINFO_8
   bmiHeader As BITMAPINFOHEADER
   bmiColors(255) As RGBQUAD
End Type

Private Declare Function GetDIBits_8 _
   Lib "gdi32" Alias "GetDIBits" ( _
   ByVal aHDC As Long, _
   ByVal lngBMP As Long, _
   ByVal nStartScan As Long, _
   ByVal nNumScans As Long, _
   lpBits As Any, _
   lpbi As BITMAPINFO_8, _
   ByVal wUsage As Long _
   ) As Long
```

Listing 10.6 (Forts.)
Beispiele\10_Grafik\
10_01_Cards.xls\mdlCards

```
Private Declare Function SelectObject _
   Lib "gdi32" ( _
   ByVal hdc As Long, _
   ByVal hObject As Long _
   ) As Long

Private Declare Function CreateCompatibleDC _
   Lib "gdi32" ( _
   ByVal hdc As Long _
   ) As Long

Private Declare Function DeleteDC _
   Lib "gdi32" ( _
   ByVal hdc As Long _
   ) As Long

Private Declare Function GetObject _
   Lib "gdi32" _
   Alias "GetObjectA" ( _
   ByVal hObject As Long, _
   ByVal nCount As Long, _
   lpObject As Any _
   ) As Long

Private Declare Function FreeLibrary& _
   Lib "kernel32" ( _
   ByVal hLibModule As Long _
   )

Private Declare Function DeleteObject _
   Lib "gdi32" ( _
    ByVal hObject As Long _
    ) As Long

Private Declare Function FindWindow _
   Lib "user32" Alias "FindWindowA" ( _
   ByVal lpClassName As String, _
   ByVal lpWindowName As String _
   ) As Long

Private Declare Sub Sleep _
   Lib "kernel32" ( _
   ByVal dwMilliseconds As Long _
   )

Private Declare Function LoadBitmapBynum _
   Lib "user32" Alias "LoadBitmapA" ( _
   ByVal hInstance As Long, _
   ByVal lpBitmapName As Long _
   ) As Long

Private Declare Function LoadLibrary _
   Lib "kernel32" Alias "LoadLibraryA" ( _
   ByVal lpLibFileName As String _
   ) As Long
```

Listing 10.6 (Forts.)
Beispiele\10_Grafik\
10_01_Cards.xls\mdlCards

```
Private Declare Function CloseClipboard _
   Lib "user32" () As Long

Private Declare Function OpenClipboard _
   Lib "user32" ( _
   ByVal hwnd As Long _
   ) As Long

Private Declare Function EmptyClipboard _
   Lib "user32" () As Long

Private Declare Function SetClipboardData _
   Lib "user32" ( _
   ByVal wFormat As Long, _
   ByVal hMem As Long _
   ) As Long

Private Const BI_RGB As Long = 0
Private Const CF_BITMAP = 2

Public Sub InsertCards()
   Dim lngCounter    As Long
   Dim lngCounter1   As Long
   Dim lngModul      As Long
   Dim lngBMP        As Long
   Dim lngMyWnd      As Long
   Dim lngIndex      As Long
   Dim lngRow        As Long
   Dim lngColumn     As Long
   Dim lngDummy      As Long
   Dim objCard       As Shape

   With ActiveSheet

      ' Alle "Bild"-Shapes auf dem Blatt löschen
      For lngCounter = .Shapes.Count To 1 Step -1
         If InStr(1, .Shapes(lngCounter).Name, "Bild") Then
            .Shapes(lngCounter).Delete
         End If
      Next

      ' Excel-Fensterhandle für Clipboard ermitteln
      lngMyWnd = FindWindow(vbNullString, Application.Caption)

   ' Library in den Prozessraum einblenden
   lngModul = LoadLibrary("Cards.dll")
   If lngModul = 0 Then lngModul = LoadLibrary("Cards32.dll")
   If lngModul = 0 Then Exit Sub

   ' Alle 4 Farben und die Rückseiten der Karten durchlaufen
   For lngCounter1 = 1 To 5

      ' Alle Karten einer Farbe durchlaufen
      ' und alle 13 Rückseiten
      For lngCounter = 1 To 13
```

Listing 10.6 (Forts.)
Beispiele\10_Grafik\
10_01_Cards.xls\mdlCards

```
            ' Bitmap löschen, falls eine geladen
            If lngBMP <> 0 Then DeleteObject lngBMP

            ' Clipboard schließen
            CloseClipboard

            ' Clipboard öffnen
            OpenClipboard lngMyWnd

            ' Clipboard leeren
            EmptyClipboard

            ' ZielngRow und ZielngColumn errechnen
            lngRow = lngCounter1 * 3
            lngColumn = lngCounter
            If lngColumn = 13 Then
                lngIndex = (lngCounter1 - 1) * 13 + 1
            Else
                lngIndex = (lngCounter1 - 1) * 13 + lngColumn + 1
            End If

            ' Bitmap laden
            lngBMP = LoadBitmapBynum(lngModul, lngIndex)

            ' Ins Clipboard stellen
            SetClipboardData CF_BITMAP, lngBMP

            ' Ins Blatt einfügen
            .Paste

            ' Position bestimmen, Namen ändern
            Set objCard = .Shapes(.Shapes.Count)
            objCard.Left = .Cells(lngRow, lngColumn).Left
            objCard.Top = .Cells(lngRow, lngColumn).Top
            objCard.Name = "Bild " & lngIndex

            ' Verzerrung zulassen
            ' objCard.LockAspectRatio = msoFalse

            ' Auf Zellenbreite
            objCard.Width = .Cells(lngRow, lngColumn).Width

            ' Auf Zellenhöhe
            ' objCard.Height = .Cells(lngRow, lngColumn).Height

        Next lngCounter
    Next lngCounter1
End With

DeleteObject lngBMP

CloseClipboard

FreeLibrary lngModul
End Sub
```

Listing 10.6 (Forts.)
Beispiele\10_Grafik\
10_01_Cards.xls\mdlCards

```
Public Sub ExportCards()
   Dim lngCounter    As Long
   Dim lngCounter1   As Long
   Dim lngIndex      As Long
   Dim lngModul      As Long
   Dim lngBMP        As Long
   Dim udtBMP        As BITMAP
   Dim lngBufferLen  As Long
   Dim abytPuffer()  As Byte
   Dim lngLen        As Long
   Dim strPath       As String
   Dim udtBMPInfo_8  As BITMAPINFO_8
   Dim hOldBitmap    As Long
   Dim udtFileHeader As BITMAPFILEHEADER
   Dim strDestDir    As String
   Dim lngDummyDC    As Long
   Dim lngFF         As Long
   Dim varColor      As Variant
   Dim varValue      As Variant

   ' Namen der Farben
   varColor = Array("Kreuz", "Karo", "Herz", "Pik", "Rücken")

   ' Kartenwerte
   varValue = Array("As", "Zwei", "Drei", "Vier", "Fünf", _
      "Sechs", "Sieben", "Acht", "Neun", "Zehn", "Bube", _
      "Dame", "König")

   ' Abfrage Speicherordner
   strDestDir = VBGetFolder("Bitte einen Speicherordner" & _
       " für die Karten wählen", "c:" & Environ$("HOMEPATH"))
   If strDestDir = "" Then Exit Sub

   ' DLL verfügbar machen.
   ' Der Referenzzähler der Bibliothek wird um 1 erhöht
   lngModul = LoadLibrary("Cards.dll")
   If lngModul = 0 Then lngModul = LoadLibrary("Cards32.dll")
   If lngModul = 0 Then Exit Sub

   ' Alle 4 Farben und die Rückseiten der Karten durchlaufen
   For lngCounter1 = 1 To 5

      ' Alle Karten einer Farbe durchlaufen
      ' und alle 13 Rückseiten
      For lngCounter = 1 To 13

         'Bitmap löschen, falls eine geladen
         If lngBMP <> 0 Then DeleteObject lngBMP

         ' Bitmap per Nummer aus der .dll laden
         lngIndex = lngIndex + 1
         lngBMP = LoadBitmapBynum(lngModul, lngIndex)

         ' Kompatiblen DC zum Screen erstellen
         lngDummyDC = CreateCompatibleDC(0)
```

Listing 10.6 (Forts.)
Beispiele\10_Grafik\
10_01_Cards.xls\mdlCards

```
'Infos zur Bitmap holen
GetObject lngBMP, Len(udtBMP), udtBMP

' Doppelwortgrenze beim Berechnen
' der Größe des Puffers beachten
lngBufferLen = ((udtBMP.bmWidth + 3) _
    And &HFFFFFFFC) * udtBMP.bmHeight
ReDim abytPuffer(lngBufferLen - 1)

' Die Bitmap in den erzeugten DC stellen
hOldBitmap = SelectObject(lngDummyDC, lngBMP)

' Die Bitmapinfos eintragen
With udtBMPInfo_8.bmiHeader
   .biBitCount = 8
   .biSize = 40
   .biWidth = udtBMP.bmWidth
   .biHeight = udtBMP.bmHeight
   .biPlanes = 1
   .biCompression = BI_RGB
   .biSizeImage = lngBufferLen
End With

' Daten holen
GetDIBits_8 lngDummyDC, lngBMP, 0&, _
   udtBMP.bmHeight, abytPuffer(0), udtBMPInfo_8, 0&
lngLen = Len(udtBMPInfo_8)

' Fileheaderinfos eintragen
With udtFileHeader
   .bfType = &H4D42 ' "BM"
   .bfSize = lngBufferLen + Len(udtFileHeader) + lngLen
   .bfOffBits = Len(udtFileHeader) + lngLen
End With

' Erzeugten DC löschen
DeleteDC lngDummyDC

' Bitmap löschen
DeleteObject lngBMP

' Als BMP speichern
lngFF = FreeFile

' Speichername zusammensetzen
If lngCounter1 = 5 Then
   strPath = varColor(lngCounter1 - 1) & "_" & _
      lngCounter & ".bmp"
Else
   strPath = varColor(lngCounter1 - 1) & "_" & _
      varValue(lngCounter - 1) & ".bmp"
End If
strPath = strDestDir & "\" & strPath

' Falls Datei existiert, löschen
If Dir(strPath) <> "" Then Kill strPath
```

Listing 10.6 (Forts.)
Beispiele\10_Grafik\
10_01_Cards.xls\mdlCards

```
            ' Bitmap-Datei erstellen
            Open strPath For Binary As lngFF
               Put lngFF, , udtFileHeader
               Put lngFF, , udtBMPInfo_8
               Put lngFF, , abytPuffer
            Close

         Next lngCounter
      Next lngCounter1

      ' Der Referenzzähler der Bibliothek wird um 1 vermindert
      FreeLibrary lngModul
End Sub
```

ExportCards

In der Bibliothek *cards.dll* sind von jeder der vier Farben dreizehn Spielkarten und zusätzlich noch einmal dreizehn verschiedene Rückseiten vorhanden.

Zu Beginn der Prozedur wird ein Array mit den Speichernamen der vier Farben und eins mit den dreizehn Werten der Karten angelegt, die später zum eigentlichen Namen kombiniert werden. Danach wird mit der Funktion `VBGetFolder` der Speicherpfad ermittelt.

Daraufhin muss mit `LoadLibrary` die Bibliothek in den Adressbereich der Anwendung eingeblendet werden. Als Funktionsergebnis wird ein Handle auf diese *.dll* geliefert. Mit der Funktion `LoadBitmapBynum` zusammen mit dem Modulhandle der *.dll* laden Sie die Bitmap aus der *.dll* und bekommen ein Handle auf diese Bitmap zurück.

Mit `GetObject` holen Sie sich anschließend die Infos zur geladenen Bitmap. Dann erstellen Sie sich einen `DeviceContext`, der kompatibel zum Screen ist, errechnen die Puffergröße für die Bitmap-Daten und stellen die Bitmap mit `SelectObject` in den DC. Danach füllen Sie die Struktur `udtBMPInfo_8` aus und holen sich mit `GetDIBits_8` die Daten aus dem Device Context. Der Fileheader wird daraufhin ausgefüllt und der Speicherpfad zusammengesetzt. Anschließend wird mit `Open` und `Put` die *Bitmap*-Datei im Zielverzeichnis erstellt.

Zum Schluss wird der `DC` gelöscht und die *.dll* entladen. Damit wird auch der Referenzzähler der *.dll* um eins vermindert.

InsertCards

Mit der Prozedur `InsertCards` werden die Karten als Shape in ein Tabellenblatt eingefügt. Eventuell vorhandene Shapes mit dem Wort »Bild« im Namen werden vorher gelöscht. Dazu wird die `Shapes`-Auflistung durchlaufen und es wird bei jedem Objekt nachgeschaut, ob die Zeichenfolge in der Eigenschaft `Name` enthalten ist.

Um die Karten zu importieren, muss mit `LoadLibrary` die *.dll* in den Adressbereich der Anwendung eingeblendet werden. Als Funktionsergebnis wird ein Handle auf die Bibliothek geliefert.

In einer Schleife greifen Sie nacheinander auf alle vorhandenen Bitmaps in dieser Bibliothek zu. Mit der Funktion `LoadBitmapBynum` und dem Modulhandle der *.dll*-Datei laden Sie die Bitmap und bekommen ein Handle darauf zurück. Eine eventuell bereits geladene Bitmap wird vorher gelöscht.

Dann wird das Clipboard mit der API `CloseClipboard` geschlossen, mit `OpenClipboard` geöffnet, mit `EmptyClipboard` geleert und die Bitmap mit `SetClipboardData` dort hineingestellt.

Jetzt kann die Bitmap mit der Methode `Paste` aus dem Clipboard in das Blatt eingefügt werden. Anschließend wird sie an die Position der Zielzelle geschoben und ihre Breite an die Zellenbreite angepasst. Standardmäßig ist keine Verzerrung zugelassen, sodass die Seitenverhältnisse beibehalten werden.

Wollen Sie die Breite und Höhe unabhängig voneinander anpassen, muss die Eigenschaft `LockAspectRatio` des Shapes auf `msoFalse` gesetzt werden. Das neue Shape wird dann umbenannt und enthält im Namen das Wort »Bild«.

Zum Schluss wird noch die letzte nicht mehr benötigte Bitmap im Speicher gelöscht, das Clipboard geschlossen und die *.dll* entladen.

10.6 Statusleiste als Zeichenfläche

Die Statusleiste ist ein interessantes Objekt. Excel gibt dort die verschiedensten Statusmeldungen aus, aber auch der Benutzer kann dort seine eigenen Meldungen absetzen. Aber leider nur einfarbig und in der voreingestellten Schriftart.

Manchmal wäre auch eine Fortschrittsanzeige in Form eines Fortschrittsbalkens hilfreich, besonders bei länger andauernden Aktionen. Der Benutzer soll ja feststellen, dass seine Anwendung noch nicht völlig abgeschmiert ist.

Es gibt verschiedene Möglichkeiten, dies zu lösen. Möglich ist eine UserForm mit einem Steuerelement oder ein Shape auf einem Tabellenblatt, wobei die Objekte in der Breite anpasst werden und so den aktuellen Fortschritt widerspiegeln. Sogar selbst erzeugte Fenster mit einer Uhr als Maß für den Fortschritt habe ich schon realisiert.

Das ist alles ganz nett, aber der Fortschritt einer Aktion gehört meiner Ansicht nach in die Statuszeile. Dort könnten Sie beispielsweise einen Haufen große I (Buchstabe i) schreiben, für jeden Prozentpunkt ein oder mehrere oder gleich den Fortschritt in Prozent ausgeben. Aber wir wollen uns nicht mit so einfachen Sachen abgeben. Also gehen wir den harten Weg, benutzen ein paar `GDI`-Funktionen (`Graphical Device Interface`) des Betriebssystems und können dabei noch etwas lernen.

Die Statusleiste ist ein Fenster wie viele andere auch und lässt sich auch dementsprechend behandeln. Es stellt eine grafische Oberfläche zur Verfügung, in die Excel die Statusinformationen hineinzeichnet. Diese Oberfläche können Sie aber auch selbst nutzen, indem Sie sich den `DC` (Device Context) ausleihen und nach Belieben darin herummalen.

Der einzige Nachteil ist, dass das Gemalte nicht dauerhaft ist. Es wird nichts automatisch neu gezeichnet. Wenn Sie mit einem anderen Fenster darüber fahren, ist darunter alles weg. Aber eine Fortschrittsanzeige, die laufend neu gezeichnet wird, kann das sicherlich verkraften.

Dagegen gelten die vorgenommenen Einstellungen für den DC, wie zum Beispiel die Attribute der Schrift, so lange, bis sie explizit geändert werden. Aber das macht Excel nicht oft, lediglich die Schriftfarbe wird häufiger zurückgesetzt. Deshalb habe ich der Klasse die Möglichkeit gegeben, die Einstellungen beim Beenden zurückzusetzen, aber auch alles so zu belassen.

10.6.1 Die Klasse clsStatusleiste

Diese Klasse (Listing 10.7) kapselt die gesamte Funktionalität und stellt nur ein paar öffentliche Eigenschaften und Methoden zur Verfügung, mit denen Sie alles komfortabel steuern können.

Listing 10.7
Beispiele\10_Grafik\
10_05_Statusbar.xls\
clsStatusleiste

```
Private Declare Function FillRect _
   Lib "user32" ( _
   ByVal hdc As Long, _
   lpRect As RECT, _
   ByVal hBrush As Long _
   ) As Long

Private Declare Function CreateSolidBrush _
   Lib "gdi32" ( _
   ByVal crColor As Long _
   ) As Long

Private Declare Function FrameRect _
   Lib "user32" ( _
   ByVal hdc As Long, _
   lpRect As RECT, _
   ByVal hBrush As Long _
   ) As Long

Private Declare Function DeleteObject _
   Lib "gdi32" ( _
   ByVal hObject As Long _
   ) As Long

Private Declare Function GetDC _
   Lib "user32" ( _
   ByVal hwnd As Long _
   ) As Long

Private Declare Function ReleaseDC _
   Lib "user32" ( _
   ByVal hwnd As Long, _
   ByVal hdc As Long _
   ) As Long
```

Listing 10.7 (Forts.)
Beispiele\10_Grafik\
10_05_Statusbar.xls\
clsStatusleiste

```
Private Declare Function GetWindowRect _
   Lib "user32" ( _
   ByVal hwnd As Long, _
   lpRect As RECT _
   ) As Long

Private Declare Function GetSystemMetrics _
   Lib "user32" ( _
   ByVal nIndex As Long _
   ) As Long

Private Declare Function FindWindowEx _
   Lib "user32" Alias "FindWindowExA" ( _
   ByVal hwnd1 As Long, _
   ByVal hWnd2 As Long, _
   ByVal lpsz1 As String, _
   ByVal lpsz2 As String _
   ) As Long

Private Declare Function GetWindow _
   Lib "user32" ( _
   ByVal hwnd As Long, _
   ByVal wCmd As Long _
   ) As Long

Private Declare Function DrawText _
   Lib "user32" Alias "DrawTextA" ( _
   ByVal hdc As Long, _
   ByVal lpStr As String, _
   ByVal nCount As Long, _
   lpRect As RECT, _
   ByVal wFormat As Long _
   ) As Long

Private Declare Function SetTextColor _
   Lib "gdi32" ( _
   ByVal hdc As Long, _
   ByVal crColor As Long _
   ) As Long

Private Declare Function GetTextColor _
   Lib "gdi32" ( _
   ByVal hdc As Long _
   ) As Long

Private Declare Function CreateFontIndirect _
   Lib "gdi32" Alias "CreateFontIndirectA" ( _
   lpLogFont As LOGFONT _
   ) As Long

Private Declare Function SelectObject _
   Lib "gdi32" ( _
   ByVal hdc As Long, _
   ByVal hObject As Long _
   ) As Long
```

Listing 10.7 (Forts.)
Beispiele\10_Grafik\
10_05_Statusbar.xls\
clsStatusleiste

```
Private Const LF_FACESIZE = 32
Private Const DT_CENTER = &H1
Private Const DT_BOTTOM = &H8
Private Const DT_LEFT = &H0
Private Const DT_RIGHT = &H2
Private Const DT_TOP = &H0
Private Const SM_CXSCREEN = 0
Private Const PS_SOLID = 0
Private Const GW_CHILD = 5&
Private Const FW_BOLD = 700&
Private Const FW_LIGHT = 300&
Private Const FW_MEDIUM = 500&
Private Const FW_NORMAL = 400&
Private Const FW_THIN = 100&

Private Type RECT
        Left As Long
        Top As Long
        Right As Long
        Bottom As Long
End Type

Private Type LOGFONT
        lfHeight As Long
        lfWidth As Long
        lfEscapement As Long
        lfOrientation As Long
        lfWeight As Long
        lfItalic As Byte
        lfUnderline As Byte
        lfStrikeOut As Byte
        lfCharSet As Byte
        lfOutPrecision As Byte
        lfClipPrecision As Byte
        lfQuality As Byte
        lfPitchAndFamily As Byte
        lfFaceName(LF_FACESIZE) As Byte
End Type

Private Type MyOldStatusbar
    Font As Long
    TextColor As Long
End Type

Private mlngTextColor         As Long
Private mstrFont              As String
Private mblnColor             As Boolean
Private mlngTextWidth         As Long
Private mlngTextThickness     As Long
Private mlngTextSize          As Long
Private mlngBar               As Long
Private mlngFrame             As Long
Private mbytOrientation       As Byte
Private mblnItalic            As Boolean
Private mblnStrikeOut         As Boolean
Private mblnUnderline         As Boolean
Private mblnPersistant        As Boolean
Private mstrText              As String
```

Listing 10.7 (Forts.)
Beispiele\10_Grafik\
10_05_Statusbar.xls\
clsStatusleiste

```
Private mudtOldStatus          As MyOldStatusbar
Private mlngStatus             As Long
Private mlngXL                 As Long
Private mlngDCStatus           As Long
Private mlngFont               As Long

Private Sub SetStatus(lngWidthProcent As Long)
   Dim lngBrush              As Long
   Dim strText               As String
   Dim udtRectBar            As RECT
   Dim udtRectFrame          As RECT
   Dim udtFont               As LOGFONT
   Dim abytFontname()        As Byte
   Dim udtRectTextFrame      As RECT
   Dim udtRectAll            As RECT
   Dim lngWidthAll           As Long
   Dim lngWidth              As Long
   Dim lngHeightAll          As Long
   Dim i                     As Long
   Dim lngWidthAvailable     As Long
   Dim lngStatusbar          As Long
   Dim lngReserved           As Long
   Dim lngFreeForText        As Long

   ReDim abytFontname(0)

   ' Schriftart Text
   If mstrFont <> "" Then
      mstrFont = Left$(mstrFont, 31)
      abytFontname = StrConv(mstrFont & _
         Chr$(0), vbFromUnicode)
   End If

   ' Eigenschaften Schriftart setzen
   With udtFont
      .lfItalic = mblnItalic
      .lfStrikeOut = mblnStrikeOut
      .lfUnderline = mblnUnderline
      .lfHeight = mlngTextSize * -1
      .lfWeight = mlngTextThickness
      .lfWidth = mlngTextWidth
      For i = 0 To UBound(abytFontname)
         .lfFaceName(i) = abytFontname(i)
      Next
   End With

   FindStatusHandle ' Handle ermitteln

   ' Beenden, wenn keine Statusleiste gefunden wurde
   If mlngStatus = 0 Then Exit Sub

   ' Größe der Statusleiste ermitteln
   GetWindowRect mlngStatus, udtRectAll
   With udtRectAll
      lngWidthAll = .Right - .Left
      lngHeightAll = .Bottom - .Top
   End With
```

Listing 10.7 (Forts.)
Beispiele\10_Grafik\
10_05_Statusbar.xls\
clsStatusleiste

```
' Hier eventuell etwas experimentieren.
lngReserved = GetSystemMetrics(SM_CXSCREEN) * 0.35
lngWidthAvailable = lngWidthAll - lngReserved
lngFreeForText = 100
If lngWidthAvailable < lngFreeForText Then Exit Sub
lngWidthAvailable = lngWidthAvailable - lngFreeForText
lngWidth = CLng((lngWidthAvailable / 100) * lngWidthProcent)

With udtRectTextFrame
    .Left = lngWidthAvailable + 3
    .Top = 0
    .Right = lngWidthAvailable + 3 + lngFreeForText
    .Bottom = lngHeightAll - 3
End With

With udtRectBar
    .Left = 3
    .Top = 3
    .Right = lngWidth
    .Bottom = lngHeightAll - 2
End With

With udtRectFrame
    .Left = 3
    .Top = 3
    .Right = lngWidthAvailable
    .Bottom = lngHeightAll - 2
End With

' Text in der Statusbar löschen
Application.StatusBar = ""

' Fenster-DC ausleihen
mlngDCStatus = GetDC(mlngStatus)

' Den SetStatus in den DC malen
lngBrush = CreateSolidBrush(mlngBar)
FillRect mlngDCStatus, udtRectBar, lngBrush
DeleteObject lngBrush

' Den Rahmen in den DC malen
lngBrush = CreateSolidBrush(mlngFrame)
FrameRect mlngDCStatus, udtRectFrame, lngBrush
DeleteObject lngBrush

' Die Beschriftung in den DC malen
strText = Format$(lngWidthProcent, "0.00") & " %"
mlngFont = CreateFontIndirect(udtFont)
mudtOldStatus.Font = SelectObject(mlngDCStatus, mlngFont)

' Alte Farbe sichern
mudtOldStatus.TextColor = GetTextColor(mlngDCStatus)

' Wenn  Farbe ausgewählt, neue Farbe setzen
If mblnColor Then SetTextColor mlngDCStatus, Textfarbe
DrawText mlngDCStatus, strText, _
   Len(strText), udtRectTextFrame, mbytOrientation
```

Listing 10.7 (Forts.)
Beispiele\10_Grafik\
10_05_Statusbar.xls\
clsStatusleiste

```
    ' Einstellungen zurücksetzen, wenn gewünscht
    If Not mblnPersistant Then ResetSettings

    ' DC zurückgeben
    ReleaseDC mlngStatus, mlngDCStatus
End Sub

Private Sub SetStatusText(Text As String)
    Dim udtFont             As LOGFONT
    Dim abytFontname()      As Byte
    Dim udtRectTextFrame    As RECT
    Dim udtRectAll          As RECT
    Dim lngWidthAll         As Long
    Dim lngWidth            As Long
    Dim lngHeightAll        As Long
    Dim i                   As Long
    Dim lngWidthAvailable   As Long
    Dim lngStatusbar        As Long
    Dim lngReserved         As Long
    Dim lngFreeForText      As Long

    ReDim abytFontname(0)

    ' Schriftart Text
    If mstrFont <> "" Then
        mstrFont = Left$(mstrFont, 31)
        abytFontname = StrConv(mstrFont & _
            Chr$(0), vbFromUnicode)
    End If

    ' Eigenschaften setzen
    With udtFont
        .lfItalic = mblnItalic
        .lfStrikeOut = mblnStrikeOut
        .lfUnderline = mblnUnderline
        .lfHeight = mlngTextSize * -1
        .lfWeight = mlngTextThickness
        .lfWidth = mlngTextWidth
        For i = 0 To UBound(abytFontname)
            .lfFaceName(i) = abytFontname(i)
        Next
    End With

    FindStatusHandle ' Handle ermitteln

    ' Beenden, wenn keine Statusleiste gefunden wurde
    If mlngStatus = 0 Then Exit Sub

    ' Größe der Statusleiste ermitteln
    GetWindowRect mlngStatus, udtRectAll
    With udtRectAll
        lngWidthAll = .Right - .Left
        lngHeightAll = .Bottom - .Top
    End With

    ' Platz am rechten Rand freilassen
    lngReserved = GetSystemMetrics(SM_CXSCREEN) * 0.35
```

Listing 10.7 (Forts.)
Beispiele\10_Grafik\10_05_Statusbar.xls\clsStatusleiste

```
    ' Das bleibt übrig zum Beschriften
    lngFreeForText = lngWidthAll - lngReserved

    ' Den Textrahmen festlegen
    With udtRectTextFrame
        .Left = 10
        .Top = 0
        .Right = lngFreeForText
        .Bottom = lngHeightAll - 3
    End With

    ' Text in der Statusbar löschen
    Application.StatusBar = ""

    ' Fenster-DC ausleihen
    mlngDCStatus = GetDC(mlngStatus)

    ' Font erzeugen
    mlngFont = CreateFontIndirect(udtFont)

    ' Alte Font sichern, neue Font setzen
    mudtOldStatus.Font = SelectObject(mlngDCStatus, mlngFont)

    ' Alte Farbe sichern
    mudtOldStatus.TextColor = GetTextColor(mlngDCStatus)

    ' Wenn  Farbe ausgewählt, neue Farbe setzen
    If mblnColor Then SetTextColor mlngDCStatus, Textfarbe

    ' Text zeichnen
    DrawText mlngDCStatus, Text, Len(Text), _
       udtRectTextFrame, mbytOrientation

    ' Einstellungen zurücksetzen, wenn gewünscht
    If Not mblnPersistant Then ResetSettings

    ' DC zurückgeben
    ReleaseDC mlngStatus, mlngDCStatus
End Sub

Private Sub FindStatusHandle()

    ' Fenster von Excel finden
    mlngXL = FindWindowEx(0&, 0&, "XLMAIN", Application.Caption)

    If mlngXL = 0 Then mlngXL = FindWindowEx(0&, 0&, _
          vbNullString, Application.Caption)

    ' Fenster der Statusleiste finden
    mlngStatus = FindWindowEx(mlngXL, 0&, "EXCEL4", vbNullString)

    If mlngStatus = 0 Then mlngStatus = GetWindow(mlngXL, GW_CHILD)
End Sub

Private Sub ResetSettings()

     ' Alte Farbe setzen
     SetTextColor mlngDCStatus, mudtOldStatus.TextColor
```

Listing 10.7 (Forts.)
Beispiele\10_Grafik\
10_05_Statusbar.xls\
clsStatusleiste

```
    ' Alte Font setzen
    SelectObject mlngDCStatus, mudtOldStatus.Font

    ' Erzeugte Font löschen
    DeleteObject mlngFont

End Sub
Public Function AnzahlSchriftarten() As Long

   'Liefert die Anzahl verfügbarer Schriften
   With Application.CommandBars.FindControl(Id:=1728)
      AnzahlSchriftarten = .ListCount
   End With

End Function
Public Function SchriftartVomIndex(lIndex As Long) As String

   ' Liefert eine Schriftart über den Index
   With Application.CommandBars.FindControl(Id:=1728)
      If lIndex < 1 Or Index > .ListCount Then Exit Function
      Schriftart = .List(lIndex)
   End With

End Function
Public Function AlleSchriftarten() As Variant
   Dim i          As Long
   Dim astrFont() As String

   ' Liefert ein Array mit den verfügbaren Schriften
   With Application.CommandBars.FindControl(Id:=1728)
      ReDim astrFont(1 To .ListCount)
      For i = 1 To .ListCount
         astrFont(i) = .List(i)
      Next
      AlleSchriftarten = astrFont
   End With

End Function
Public Sub Statustextausgabe(strStatustext As String)
    SetStatusText strStatustext
End Sub
Public Sub Fortschrittsausgabe(lngProcent As Long)
   If lngProcent > 100 Then lngProcent = 100
   If lngProcent < 0 Then lngProcent = 0
   SetStatus lngProcent
End Sub

Public Property Get EinstellungenLassen() As Boolean
   EinstellungenLassen = mblnPersistant
End Property
Public Property Let EinstellungenLassen(ByVal vNewValue As Boolean)
   mblnPersistant = vNewValue
End Property

Public Property Get Unterstrichen() As Boolean
   Unterstrichen = mblnUnderline
End Property
```

Listing 10.7 (Forts.)
Beispiele\10_Grafik\
10_05_Statusbar.xls\
clsStatusleiste

```
Public Property Let Unterstrichen(ByVal vNewValue As Boolean)
   mblnUnderline = vNewValue
End Property

Public Property Get Durchgestrichen() As Boolean
   Durchgestrichen = mblnStrikeOut
End Property
Public Property Let Durchgestrichen(ByVal vNewValue As Boolean)
   mblnStrikeOut = vNewValue
End Property

Public Property Get Kursiv() As Boolean
   Kursiv = mblnItalic
End Property
Public Property Let Kursiv(ByVal vNewValue As Boolean)
   mblnItalic = vNewValue
End Property

Public Property Get Ausrichtung() As Byte
   Ausrichtung = mbytOrientation
End Property
Public Property Let Ausrichtung(ByVal vNewValue As Byte)
   ' Ausrichtung Text, default links
   Select Case vNewValue
      Case 2
         mbytOrientation = DT_CENTER
      Case 3
         mbytOrientation = DT_RIGHT
      Case Else
         mbytOrientation = DT_LEFT
   End Select
End Property

Public Property Get Schriftgrösse() As Long
   Schriftgrösse = mlngTextSize
End Property
Public Property Let Schriftgrösse(ByVal vNewValue As Long)
   mlngTextSize = vNewValue
End Property

Public Property Get Schriftdicke() As Byte
   Textfarbe = mlngTextThickness
End Property
Public Property Let Schriftdicke(ByVal vNewValue As Byte)
   ' Schriftdicke Text, default FW_THIN
   Select Case vNewValue
      Case Is >= 7
         mlngTextThickness = FW_BOLD
      Case Is >= 5
         mlngTextThickness = FW_MEDIUM
      Case 4
         mlngTextThickness = FW_NORMAL
      Case 3
         mlngTextThickness = FW_LIGHT
      Case Else
         mlngTextThickness = FW_THIN
   End Select
End Property
```

Listing 10.7 (Forts.)
Beispiele\10_Grafik\
10_05_Statusbar.xls\
clsStatusleiste

```
Public Property Get Textfarbe() As Long
   Textfarbe = mlngTextColor
End Property
Public Property Let Textfarbe(ByVal vNewValue As Long)
   mlngTextColor = vNewValue
   mblnColor = True
End Property

Public Property Get Schriftart() As String
   Schriftart = mstrFont
End Property
Public Property Let Schriftart(ByVal vNewValue As String)
   mstrFont = vNewValue
End Property

Public Property Get Schriftweite() As Long
   Schriftweite = mlngTextWidth
End Property
Public Property Let Schriftweite(ByVal vNewValue As Long)
   mlngTextWidth = vNewValue
End Property

Public Property Get Rahmenfarbe() As Long
   Rahmenfarbe = mlngFrame
End Property
Public Property Let Rahmenfarbe(ByVal vNewValue As Long)
   mlngFrame = vNewValue
End Property

Public Property Get FarbeFortschrittsbalken() As Long
   FarbeFortschrittsbalken = mlngBar
End Property
Public Property Let FarbeFortschrittsbalken(ByVal vNewValue As Long)
   mlngBar = vNewValue
End Property

Private Sub Class_Initialize()
   mlngTextThickness = FW_THIN
   mbytOrientation = DT_LEFT
End Sub

Private Sub Class_Terminate()
   If Not mblnPersistant Then ResetSettings
End Sub
```

ResetSettings

Es werden die vorherigen Einstellungen wiederhergestellt. Dazu werden die API-Funktionen `SetTextColor` und `SelectObject` benutzt. Mit `SetTextColor` wird die alte Textfarbe des `DC` zurückgesetzt und mit `SelectObject` wird die vorherige Schriftart in den `DC` gestellt. Anschließend wird der nicht mehr benötigte Font gelöscht.

FindStatusHandle

Es wird das Handle der Statusleiste gesucht. Dazu ermitteln Sie zuerst das Handle von Excel. Dies ist notwendig, weil die Statusleiste ein Kindfenster (`Child`) von Excel ist. Mit der API-Funktion `FindWindow` können leider nur Top-Level-Fenster gefunden werden, also solche, die auf der obersten Ebene liegen. Die API-Funktion `FindWindowEx` findet auch Fenster, die darunter liegen. Dazu benötigen Sie aber das Handle des darüber liegenden Fensters. Bei Top-Level-Fenstern ist das der Desktop mit dem Fensterhandle `Null`.

Mit dem Klassennamen `XLMAIN` und dem Fenstertext (`Application.Caption`) wird das Hauptfenster von Excel ermittelt. Der Klassenname hat sich bis zum Jahr 2003 nicht verändert und ich nehme nicht an, dass sich das in Zukunft ändert. Falls doch, muss auf den Klassennamen verzichtet werden und stattdessen die Konstante `vbNullString` eingesetzt werden.

Mit dem Fensterhandle von Excel können Sie sich nun an das Ermitteln des Statusleistenhandles machen. Hier wird der Klassenname `EXCEL4` eingesetzt. Falls der Klassenname doch irgendwann nicht mehr stimmen sollte, wird über `GetWindow(XLFenster, GW_CHILD)` das Handle geholt. Die Statusleiste ist das erste Kindfenster von Excel, zumindest war es in den bisherigen Versionen der Fall.

SetStatusText

Diese Prozedur gibt einen Text mit allen eingestellten Attributen in der Statusleiste aus.

Wurde eine Schriftart übergeben, wird der Text inklusive einem abschließendem `Chr(0)` in ein ANSI-Array umgewandelt. Dazu wird die Funktion `StrConv` benutzt.

Danach wird eine Struktur vom Typ `LOGFONT` mit den gesetzten Eigenschaften ausgefüllt.

Nachdem durch den Aufruf von `FindStatusHandle` das Fensterhandle ermittelt wurde, holen Sie sich die Größe der Statusleiste. Dazu wird die API-Funktion `GetWindowRect` benutzt, die eine Struktur vom Typ `RECT` mit den Infos ausfüllt.

Der rechte Rand der Statusleiste sollte frei bleiben. Es sieht nicht schön aus, dort etwas hineinzumalen. Beim Testen habe ich festgestellt, dass etwa 35% der Bildschirmbreite frei bleiben sollte, und zwar unabhängig von der Fenstergröße. Das hat zur Folge, dass unter Umständen kein Platz zum Malen mehr frei ist.

Danach wird der Text der Statusleiste gelöscht. Mit `CreateFontIndirect` und der am Anfang ausgefüllten `LOGFONT`-Struktur wird anschließend eine Schrift mit den gewünschten Attributen erzeugt. Diese wird mit dem `DC` des Fensters verknüpft, den Sie sich zuvor mit der API `GetDC` ausgeliehen haben. Die alten Attribute werden als Handle unter `mudtOldStatus.Font` gespeichert, damit die ursprünglichen Einstellungen ohne Probleme zurückgesetzt werden können.

Die alte Textfarbe wird mit der API `GetTextColor` ermittelt und unter `mudtOldStatus.TextColor` gespeichert. Dann wird mit `SetTextColor` die neue Textfarbe

gesetzt und anschließend mit `DrawText` der Text gemalt. Zum Schluss werden die alten Attribute zurückgesetzt, sofern das überhaupt gewünscht ist. Ganz wichtig ist es, am Ende den `DC` wieder zurückzugeben.

SetStatus

Diese Prozedur gibt den übergebenen Wert als Fortschrittsbalken und Text mit allen eingestellten Attributen in der Statusleiste aus.

Der größte Teil ist genauso wie in der Prozedur `SetStatusText`. Der Unterschied besteht in dem Malen des Fortschrittsbalkens.

Mit der API `CreateSolidBrush` erzeugen Sie sich einen Pinsel mit den gewünschten Eigenschaften und malen den Balken mit der API `FillRect` in der prozentualen Länge in den DC.

Mit der API `CreateSolidBrush` erzeugen Sie anschließend einen Pinsel mit den gewünschten Eigenschaften für den Rahmen und malen diesen mit der API `FrameRect` in den DC.

AlleSchriftarten

Beim Aufruf dieser Funktion wird ein Array mit den Namen aller verfügbaren Schriftarten geliefert.

Dazu wird das Control `CommandBars.FindControl(Id:=1728)` von Excel als Lieferant benutzt.

AnzahlSchriftarten

Beim Aufruf dieser Funktion wird die Anzahl verfügbarer Schriften geliefert.

SchriftartVomIndex

Beim Aufruf dieser Funktion wird der Name einer Schrift geliefert, die in der Liste der verfügbaren Schriften an der entsprechenden Stelle steht.

Statustextausgabe

Beim Aufruf dieser Methode wird der übergebene Text mit den entsprechenden Attributen in der Statuszeile ausgegeben.

Fortschrittsausgabe

Beim Aufruf dieser Methode wird der übergebene Prozentwert als Fortschrittsbalken und als Prozentwert mit den entsprechenden Attributen in der Statuszeile ausgegeben.

EinstellungenLassen

Wird diese Eigenschaft auf Wahr gesetzt, werden die ursprünglichen Attribute der Statusleiste nicht mehr wiederhergestellt.

Unterstrichen

Wird diese Eigenschaft auf Wahr gesetzt, werden die ausgegebenen Texte unterstrichen dargestellt.

Durchgestrichen

Wird diese Eigenschaft auf Wahr gesetzt, werden die ausgegebenen Texte durchgestrichen dargestellt.

Kursiv

Wird diese Eigenschaft auf Wahr gesetzt, werden die ausgegebenen Texte kursiv dargestellt.

Ausrichtung

Diese Eigenschaft legt fest, ob der ausgegebene Text linksbündig, zentriert oder rechtsbündig dargestellt wird. Der Wert 2 gilt für zentriert, 3 für rechtsbündig und jeder andere Wert stellt auf linksbündig ein.

Schriftgrösse

Diese Eigenschaft legt fest, in welcher Schriftgröße der ausgegebene Text dargestellt wird.

Schriftdicke

Diese Eigenschaft legt fest, in welcher Strichstärke der ausgegebene Text dargestellt wird.

Die gesetzten Werte habe ich wie folgt den Konstanten zugeordnet:

- größer 7 = `FW_BOLD`
- größer 5 = `FW_MEDIUM`
- 4 = `FW_NORMAL`
- 3 = `FW_LIGHT`
- andere = `FW_THIN`

Textfarbe

Diese Eigenschaft legt fest, in welcher RGB-Farbe der ausgegebene Text dargestellt wird.

Schriftart

Diese Eigenschaft legt fest, in welcher Schriftart der ausgegebene Text dargestellt wird.

Schriftweite

Diese Eigenschaft legt fest, in welcher Schriftbreite der ausgegebene Text dargestellt wird.

Rahmenfarbe

Diese Eigenschaft legt fest, in welcher RGB-Farbe der Rahmen um den Fortschrittsbalken dargestellt wird.

FarbeFortschrittsbalken

Diese Eigenschaft legt fest, in welcher RGB-Farbe der Fortschrittsbalken dargestellt wird.

10.6.2 Prozeduren zum Testen der Klasse

Mit den folgenden drei Prozeduren können Sie die Klasse testen.

Fortschrittsanzeige

In diesem Beispiel (Listing 10.8) zur Benutzung der Klasse wird ein Fortschrittsbalken mit Prozentanzeige erzeugt.

Listing 10.8
Beispiele\10_Grafik\
10_05_Statusbar.xls\mdlStatus

```
Public Sub TestFortschritt()
   Dim lngCount          As Long
   Dim lngColor          As Long
   Dim lngFrameColor     As Long
   Dim lngRectColor      As Long
   Dim strFont           As String
   Dim objStatus         As New clsStatusleiste

   ' Farbe Text RGB(Rotanteil , Grünanteil , Blauanteil)
   lngColor = RGB(255, 0, 0)

   ' Farbe Rahmen RGB(Rotanteil , Grünanteil , Blauanteil)
   lngFrameColor = RGB(0, 255, 0)

   ' Farbe Balken RGB(Rotanteil , Grünanteil , Blauanteil)
   lngRectColor = RGB(0, 0, 255)

   ' Schriftart
   strFont = "Arial Black"

   With objStatus
      For lngCount = 1 To 10
         ' Ausrichtung links=1, Mitte=2, rechts=3
         .Ausrichtung = 2
         .Durchgestrichen = False
         .EinstellungenLassen = True
         .Kursiv = True
         ' Schriftart Leer=Default
         .Schriftart = strFont
         ' Schriftdicke 0-7
         .Schriftdicke = 4
         ' Schriftgröße 6-14 sinnvoll
         .Schriftgrösse = 12
         .Textfarbe = lngColor
         .Rahmenfarbe = lngFrameColor
         .FarbeFortschrittsbalken = lngRectColor
```

Listing 10.8 (Forts.)
Beispiele\10_Grafik\
10_05_Statusbar.xls\mdlStatus

```
            .Unterstrichen = True
            .Fortschrittsausgabe lngCount * 10
            .Schriftweite = 0
            Application.Wait (Now + TimeSerial(0, 0, 1))
        Next
    End With

' Statusbar normal setzen
Application.StatusBar = False

End Sub
```

Und hier (Abbildung 10.5) sehen Sie das Ergebnis einer Fortschrittsanzeige in der Statusleiste.

Abbildung 10.5
Fortschrittsanzeige

Vorhandene Schriftarten auflisten

Mithilfe der Klasseneigenschaft `AlleSchriftarten` wird in diesem Beispiel (Listing 10.9) ein Array mit allen installierten Schriftarten erzeugt und die Schriften werden mitsamt den geänderten Zellformaten in ein Tabellenblatt ausgegeben.

Listing 10.9
Beispiele\10_Grafik\
10_05_Statusbar.xls\mdlStatus

```
Public Sub Schriftarten()
    Dim objStatus  As New clsStatusleiste
    Dim i          As Long
    Dim varFonts   As Variant

    On Error Resume Next

    With Worksheets("Tabelle2")

        Application.ScreenUpdating = False
        .Range("A2:B1000").ClearContents

        ' Die Klasse liefert Schriftarten als Variantarray
        varFonts = objStatus.AlleSchriftarten

        ' Alle Elemente des Arrays durchlaufen
        For i = 1 To UBound(varFonts)
            ' Eintragen
            .Cells(i, 1) = varFonts(i)
            .Cells(i, 2) = varFonts(i)

            ' Formatieren
            .Cells(i, 2).Font.Name = varFonts(i)
        Next

    End With

    Application.ScreenUpdating = False
End Sub
```

Schriftformate in der Statusleiste

In diesem Beispielcode (Listing 10.10) wird die Textfarbe und Schriftart der Statusleiste geändert und dort ein Text mit den geänderten Einstellungen ausgegeben.

Listing 10.10
Beispiele\10_Grafik\
10_05_Statusbar.xls\mdlStatus

```
Public Sub TestenStatustext()
   Dim lngCount As Long
   Dim lngColor As Long
   Dim strFont As String
   Dim objStatus As New clsStatusleiste

   ' Farbe Text Rotanteil, Grünanteil, Blauanteil
   lngColor = RGB(0, 0, 255)

   ' Schriftart
   strFont = "Arial Black"

   With objStatus
      For lngCount = 1 To 10
         ' Ausrichtung links=1, Mitte=2, rechts=3
         .Ausrichtung = 1
         .Durchgestrichen = False
         .EinstellungenLassen = True
         .Kursiv = True
         ' Schriftart Leer=Default
         .Schriftart = strFont
         ' Schriftdicke 0-7
         .Schriftdicke = 2
         ' Schriftgröße 6-14 sinnvoll
         .Schriftgrösse = 14
         .Textfarbe = lngColor
         .Unterstrichen = True
         .Schriftweite = 0
         .Statustextausgabe "Durchlauf:" & lngCount
         Application.Wait (Now + TimeSerial(0, 0, 1))
      Next
   End With

   ' Statusbar normal setzen
   Application.StatusBar = False

End Sub
```

Die geänderte Schriftart und Farbe der Statusleiste (Abbildung 10.6):

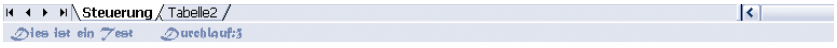


Abbildung 10.6
Statustext mit anderen Textattributen

11

Sound

11.1 Was Sie in diesem Kapitel erwartet

Da heutzutage nahezu jeder Rechner über eine Soundkarte verfügt, ist das Benutzen von Klängen in Anwendungen fast eine Selbstverständlichkeit. Wenn Sie VBA verwenden, sollten Sie wenigstens wissen, was es für Möglichkeiten gibt, Töne zu erzeugen und abzuspielen.

Zuerst werden die `Beep`-Anweisung von VBA und die API-Funktion `Beep` vorgestellt. Anschließend werden Sie erfahren, wie Klänge im *.avi*-Format abgespielt werden können, wobei die `Shell`-Funktion und die API-Funktion `sndPlaySound` zum Einsatz kommen.

Mit der Funktion `mciSendString` können Sie Video-, Audiodateien und auch ganze Audio-CDs abspielen. Sogar die Laufwerksschubladen lassen sich programmgesteuert öffnen und schließen.

Mithilfe der selbst geschriebenen Klasse `clsWave` können Sie mit der Soundkarte Töne als Sinus, Dreieck und Rechteck in Stereo erzeugen und ausgeben, wobei sich dabei Frequenz, Amplitude und Lautstärke einstellen lassen.

Um Töne der Tonleiter in verschiedenen Oktaven abzuspielen, können Sie die Midifunktionen der *winmm.dll* benutzen und haben dabei sogar die Möglichkeit, verschiedene Instrumente nachzuahmen.

11.2 Die Beep-Anweisung

Die einfachste Möglichkeit, einen Ton oder Klang zu erzeugen, besteht darin, die VBA-Anweisung `Beep` zu benutzen. Sie haben damit aber keine Möglichkeit, die Frequenz und Länge des Tons zu beeinflussen. Diese Einstellungen sind von der Hardware, der System-Software und von den Benutzereinstellungen abhängig. Bei Systemen ohne Soundkarte kommt der Systemlautsprecher zum Einsatz, bei Systemen mit Soundkarte der eingestellte Klang. Um auf sich aufmerksam zu machen, reicht der Standard-Beep in vielen Fällen aus.

11.3 Die API-Funktion Beep

Etwas komfortabler als die `Beep`-Anweisung von VBA ist die API-Funktion `Beep`. Als Argument kann dort die Frequenz und die Dauer des Tones angegeben werden. Leider funktioniert das mit der Frequenz und der Dauer nur bei Betriebssystemen der NT-Familie und deren Nachfolgern 2000 und XP. Auf anderen Systemen wird der eingestellte Standard-Beep verwendet.

Die Frequenz kann zwischen 37 und 32.767 Herz eingestellt werden und die Zeitdauer wird in Millisekunden übergeben.

Nachfolgender Code (Listing 11.1) zeigt die Benutzung der API-Funktion `Beep`.

Listing 11.1
Beispiele\11_Sound\
11_01_Sound.xls\mdlBeep

```
Private Declare Function Beep _
   Lib "kernel32" ( _
   ByVal dwFreq As Long, _
   ByVal dwDuration As Long _
   ) As Long

Sub test()
   ' 2000 Herz, Dauer 2 Sekunden
   Beep 2000, 2000
End Sub
```

11.4 Die API-Funktion sndPlaySound

Neben einem einfachen Beep können Besitzer einer Soundkarte auch Klänge im *.wav*-Format abspielen. Dies kann mit der `Shell`-Funktion erreicht werden, wie hier in diesem kleinen Beispiel zu sehen ist:

```
Private Sub ShellPlay()
   Shell "SNDREC32.EXE " & _
      """c:\windows\media\Der Microsoft-Sound.wav""" & _
      " /play /close", vbMinimizedNoFocus
End Sub
```

Der Nachteil dabei ist, dass in der Taskleiste in dieser Zeit das Programm erscheint.

Mit der API-Funktion `sndPlaySound` können Sie das Gleiche ohne den erwähnten Nachteil erledigen.

Listing 11.2
Beispiele\11_Sound\
11_01_Sound.xls\
mdlSndPlaySound

```
Private Declare Function sndPlaySound _
   Lib "winmm.dll" Alias "sndPlaySoundA" ( _
   ByVal lpszSoundName As String, _
   ByVal uFlags As Long _
   ) As Long

Private Const SND_SYNC = &H0
Private Const SND_ASYNC = &H1
Private Const SND_LOOP = &H8
Private Const SND_PURGE = &H40
```

```
Private Sub SoundPlay()
   ' Einmal
   sndPlaySound "c:\windows\media\Der Microsoft-Sound.wav" _
      , SND_ASYNC
   ' Dauernd
   ' sndPlaySound "c:\windows\media\Der Microsoft-Sound.wav" _
      , SND_ASYNC Or SND_LOOP
End Sub

Private Sub SoundStop()
   ' Wiedergabe stoppen
   sndPlaySound 0, SND_PURGE
End Sub
```

Wenn Sie als zweiten Parameter die Konstante `SND_LOOP` mit übergeben, wird der Sound so lange wiederholt, bis das Abspielen mit dem Argument `SND_PURGE` wieder gestoppt wird.

11.5 Die API-Funktion mciSendString

Mithilfe dieser Funktion können Sie komfortabel verschiedene Geräte ansprechen und benutzen. Wenn der richtige String gesendet wird, ist es möglich, Video- oder Audiodateien und auch ganze Audio-CDs abspielen zu lassen. Sie können Vor- und Zurückspulen, bei einer Audio-CD ein bestimmtes Lied ansteuern, eine vorgegebene Stelle anwählen, sogar die Laufwerksschubladen lassen sich programmgesteuert öffnen und schließen.

Listing 11.3
Beispiele\11_Sound\
11_01_Sound.xls\mdlMCI

```
Declare Function mciSendString _
   Lib "winmm.dll" Alias "mciSendStringA" ( _
   ByVal lpstrCommand As String, _
   ByVal lpstrReturnString As String, _
   ByVal uReturnLength As Long, _
   ByVal hwndCallback As Long _
   ) As Long

Private Sub TestCDDoor()
   Dim strCommand       As String
   Dim strAlias         As String
   Dim strDriveLetter   As String

   strDriveLetter = "h"
   strAlias = "LW" & strDriveLetter

   ' Gerät öffnen
   strCommand = "Open " & strDriveLetter & ": Alias " & _
   strAlias & " Type CDAudio"
   MCISend strCommand

   ' Laufwerkschublade öffnen
   strCommand = "Set " & strAlias & " Door Open"
   MCISend strCommand
```

Listing 11.3 (Forts.)
Beispiele\11_Sound\
11_01_Sound.xls\mdlMCI

```
    MsgBox "Schublade Schließen!", vbOK
    ' Laufwerksschublade schließen
    strCommand = "Set " & strAlias & " Door Closed"
    MCISend strCommand

    MsgBox "Beenden!", vbOK
    ' Schließen (Aliasname)
    strCommand = "CLOSE " & strAlias
    MCISend strCommand

End Sub

Private Sub testMCI()
    Dim strCommand    As String
    Dim strAlias      As String

    ' Name, unter dem Sie die geöffnete Verbindung
    ' ansprechen können
    strAlias = "myMCI"

    ' Öffnen
    strCommand = "OPEN CDAudio"
    strCommand = strCommand & _
       " ALIAS " & strAlias
    MCISend strCommand

    strCommand = "Play " & strAlias & " from 3 to 4"
    MCISend strCommand

    ' "PAUSE ALIAS"= Pause
    ' "STOP ALIAS"=Stop
    ' "SEEK ALIAS to start"= Zurück zum Start
    ' "SEEK ALIAS to end"= Zum Ende spulen
    ' "PLAY ALIAS from Tracknummer"= Lied von Audio-CD
    ' "PLAY ALIAS from Tracknummer to Tracknummer"= Lied von bis Audio-CD

    MsgBox "PAUSE!", vbOK
    strCommand = "PAUSE " & strAlias
    MCISend strCommand

    MsgBox "Fortsetzen!", vbOK
    strCommand = "PLAY " & strAlias
    MCISend strCommand

    MsgBox "Stop!", vbOK
    strCommand = "STOP " & strAlias
    MCISend strCommand

    MsgBox "Beenden!", vbOK
    ' Schließen (Aliasname)
    strCommand = "CLOSE " & strAlias
    MCISend strCommand

End Sub
```

Listing 11.3 (Forts.)
Beispiele\11_Sound\
11_01_Sound.xls\mdlMCI

```
Private Sub testMCI1()
   Dim strCommand    As String
   Dim strAlias      As String
   Dim lngPosition   As Long

   ' Name, unter dem Sie die geöffnete Verbindung
   ' ansprechen können
   strAlias = "myMCI"

   ' Öffnen
   strCommand = "OPEN "
   strCommand = strCommand & _
      """c:\windows\media\Der Microsoft-Sound.wav"""
   strCommand = strCommand & _
      " TYPE waveaudio ALIAS " & strAlias
   MCISend strCommand

   ' Vorspulen (Aliasname)
   lngPosition = 4000 ' hier Millisek.
   strCommand = "Seek " & strAlias & " to " & lngPosition
   MCISend strCommand

   ' Abspielen (Aliasname)
   strCommand = "PLAY " & strAlias
   MCISend strCommand

   MsgBox "Beenden!", vbOK
   ' Schließen (Aliasname)
   strCommand = "CLOSE " & strAlias
   MCISend strCommand

End Sub

Public Function MCISend(ByVal strCommand As String) As String
   Dim lngRet  As Long
   Dim strInfo As String

   ' chr(0) anhängen
   strCommand = strCommand + vbNullString

   ' Puffer anlegen
   strInfo = String(256, 0)

   ' Kommando ausführen, Rückgabewert Null, wenn erfolgreich
   lngRet = mciSendString(strCommand, strInfo, 255, 0)

   ' zurückgeben
   MCISend = Left(strInfo, InStr(strInfo, Chr$(0)) - 1)

End Function
```

TestCDDoor

Die Prozedur `TestCDDoor` öffnet und schließt eine Laufwerkstür. Dazu wird das Gerät mit der `Open`-Anweisung im `Commandstring` geöffnet. In diesem String wird auch ein `Alias` definiert, damit später mit diesem Namen auf das geöffnete Gerät zugegriffen werden kann.

Mit `Set` und `Door Open` wird die Schublade geöffnet, mit `Set` und `Door Closed` wird sie geschlossen.

Anschließend wird das Gerät mit `CLOSE` geschlossen.

testMCI

Die Prozedur `testMCI` spielt Lieder einer Audio-CD ab. Wieder wird mit der `Open`-Anweisung eine Verbindung unter einem Alias-Namen hergestellt. Anschließend werden mit der `Play`-Anweisung die Lieder von Track drei bis vier abgespielt.

Wird die Anzeige der Messagebox mit der Meldung »PAUSE!« beendet, wird mit der `Pause`-Anweisung das Abspielen unterbrochen, und zwar so lange, bis die Messagebox mit der Meldung »Fortsetzen!« geschlossen wird. Danach wird das Abspielen mit der `Play`-Anweisung fortgesetzt.

Zum Stoppen des Abspielens und zum Beenden muss jeweils eine angezeigte Messagebox geschlossen werden.

testMCI1

Die Prozedur `testMCI1` spielt die *Wav*-Datei *c:\windows\media\Der Microsoft-Sound.wav* ab. Mit der `Open`-Anweisung wird abermals eine Verbindung unter einem Alias-Namen hergestellt. Anschließend wird mit der `Seek`-Anweisung vorgespult und mit der `Play`-Anweisung der Sound ausgegeben.

Zum Schluss wird die Verbindung mit `CLOSE` getrennt.

MCISend

Diese Funktion kapselt lediglich die API-Funktion `mciSendString` und macht den Code bei häufigem Einsatz etwas kürzer.

11.6 Piepsen

Wie ein einfacher Beep erzeugt wird, haben Sie bereits gesehen. Unter NT können Sie sogar die Frequenz und die Tondauer festlegen. Wenn Sie aber mehr wollen, müssten Sie sich schon eine Audio-Datei erzeugen und abspielen, genauso, wie es Windows mit den Systemklängen macht.

Es ist aber nicht unbedingt komfortabel und recht unflexibel, erst mühsam solch eine Datei zu erzeugen und irgendwo abzuspeichern, besonders, wenn die Arbeitsmappe weitergegeben wird. Glücklicherweise bietet aber die API einiges, um auch ohne eine solche temporäre Datei auszukommen.

Mithilfe der Klasse `clsWave` können mit der Soundkarte Töne erzeugt werden. Denkbar ist im Prinzip jede beliebige Kurvenform. Ich habe im vorliegenden Beispiel aber nur Sinus, Dreieck und Rechteck in Stereo mit einer Abtastrate von 44 KHz und einer Auflösung von 16 Bit pro Sample verwirklicht. Es kann die Kurvenform, Frequenz, Tondauer und Lautstärke für jeden einzelnen Kanal eingestellt werden.

11.6.1 Benutzen der Klasse clsWave

Das Benutzen der Klasse `clsWave` ist eigentlich ganz einfach (Listing 11.4). Sie legen sich eine Instanz der Klasse an, setzen für jeden Kanal die Frequenz, die Kurvenform, die Dauer, die Lautstärke und beginnen die Ausgabe mit der Methode `Tonausgabe`. Mit der Eigenschaft `AnzahlLoops` können Sie festlegen, wie oft die Ausgabe wiederholt wird.

Listing 11.4
Beispiele\11_Sound\
11_01_Sound.xls\mdlTest

```
Private Sub WaveTest()
   Dim objWave As New clsWave

   With objWave
      .LautstärkeLinks = 20
      .LautstärkeRechts = 70
      .LautstärkeEinstellen

      .TondauerLinks = 5000
      .TonfrequenzLinks = 2000
      .KurvenartLinks = 1
      .PegelLinks = 100

      .TondauerRechts = 1000
      .TonfrequenzRechts = 4000
      .KurvenartRechts = 1
      .PegelRechts = 100

      .AnzahlLoops = 1
      .Tonausgabe

   End With
End Sub
```

11.6.2 Die Klasse clsWave

Ein Großteil des folgenden Codes (Listing 11.5) besteht aus Deklarationsanweisungen und öffentlichen Eigenschaften und Methoden. Also nicht gleich über den Umfang erschrecken, die eigentlichen Funktionen sind gar nicht so wild und werden anschließend auch noch einmal erklärt.

Listing 11.5
Beispiele\11_Sound\
11_01_Sound.xls\clsWave

```
Private Declare Function waveOutSetVolume _
   Lib "winmm.dll" ( _
   ByVal uDeviceID As Long, _
   ByVal dwVolume As Long _
   ) As Long

Private Declare Function waveOutGetVolume _
   Lib "winmm.dll" ( _
   ByVal uDeviceID As Long, _
   lpdwVolume As Long _
   ) As Long

Private Declare Function waveOutReset _
   Lib "winmm.dll" ( _
   ByVal hWaveOut As Long _
   ) As Long
```

Listing 11.5 (Forts.)
Beispiele\11_Sound\
11_01_Sound.xls\clsWave

```
Private Declare Function waveOutOpen _
   Lib "winmm.dll" ( _
   lphWaveOut As Long, _
   ByVal uDeviceID As Long, _
   lpFormat As waveformat_tag, _
   ByVal dwCallback As Long, _
   ByVal dwInstance As Long, _
   ByVal dwFlags As Long _
   ) As Long

Private Declare Function waveOutClose _
   Lib "winmm.dll" ( _
   ByVal hWaveOut As Long _
   ) As Long

Private Declare Function waveOutPrepareHeader _
   Lib "winmm.dll" ( _
   ByVal hWaveOut As Long, _
   ByVal lpWaveOutHdr As Long, _
   ByVal uSize As Long _
   ) As Long

Private Declare Function waveOutUnprepareHeader _
   Lib "winmm.dll" ( _
   ByVal hWaveOut As Long, _
   ByVal lpWaveOutHdr As Long, _
   ByVal uSize As Long _
   ) As Long

Private Declare Function waveOutWrite _
   Lib "winmm.dll" ( _
   ByVal hWaveOut As Long, _
   ByVal lpWaveOutHdr As Long, _
   ByVal uSize As Long _
   ) As Long

Private Declare Sub CopyMemory _
   Lib "kernel32" Alias "RtlMoveMemory" ( _
   hpvDest As Any, _
   hpvSource As Any, _
   ByVal cbCopy As Long _
   )

'Global Memory
Private Declare Function GlobalAlloc _
   Lib "kernel32" ( _
   ByVal wFlags As Long, _
   ByVal dwBytes As Long _
   ) As Long

Private Declare Function GlobalFree _
   Lib "kernel32" ( _
   ByVal hMem As Long _
   ) As Long
```

Listing 11.5 (Forts.)
Beispiele\11_Sound\
11_01_Sound.xls\clsWave

```
Private Declare Function GlobalLock _
   Lib "kernel32" ( _
   ByVal hMem As Long _
   ) As Long

Private Declare Function GlobalUnlock _
   Lib "kernel32" ( _
   ByVal hMem As Long _
   ) As Long

Private Const GMEM_MOVEABLE = &H2
Private Const GMEM_SHARE = &H2000
Private Const WHDR_BEGINLOOP = &H4
Private Const WHDR_ENDLOOP = &H8
Private Const WAVE_ALLOWSYNC = 2
Private Const MMSYSERR_BASE = 0
Private Const WAVERR_BASE = 32
Private Const WAVE_FORMAT_QUERY = 1
Private Const WAVERR_STILLPLAYING = (WAVERR_BASE + 1)
Private Const MMSYSERR_INVALHANDLE = (MMSYSERR_BASE + 5)
Private Const MMSYSERR_NOMEM = (MMSYSERR_BASE + 7)
Private Const WAVERR_UNPREPARED = (WAVERR_BASE + 2)
Private Const WAVE_FORMAT_PCM = 1

'Typen
Private Type waveformat_tag
   wFormatTag As Integer
   nChannels As Integer
   nSamplesPerSec As Long
   nAvgBytesPerSec As Long
   nBlockAlign As Integer
   BitsPerSample As Integer
End Type

Private Type WaveHdr
   lpData As Long
   dwBufferLength As Long
   dwBytesRecorded As Long
   dwUser As Long
   dwFlags As Long
   dwLoops As Long
   lpNext As Long
   reserved As Long
End Type

Private mlngPercentLeft       As Long
Private mlngPercentRight      As Long
Private mlngFrequencyLeft     As Long
Private mlngDurationLeft      As Long
Private mlngFrequencyRight    As Long
Private mlngDurationRight     As Long
Private mlngGaugeLeft         As Long
Private mlngGaugeRight        As Long
Private mlngWaveLeft          As Long
Private mlngWaveRight         As Long
Private mlngLoops             As Long
Private mlngWave              As Long
Private mlngMemData           As Long
```

Listing 11.5 (Forts.)
Beispiele\11_Sound\
11_01_Sound.xls\clsWave

```
Private mlngMemHeader          As Long
Private mlngPointerMemHeader   As Long
Private mlngHeaderLength       As Long
Private mlngPointerMemData     As Long
Private mudtHeader             As WaveHdr

Public Sub Tonausgabe()
   If (mlngWaveLeft > 2) Or (mlngWaveLeft < 0) Then _
      mlngWaveLeft = 0
   If (mlngWaveRight > 2) Or (mlngWaveRight < 0) Then _
      mlngWaveRight = 0
   If (mlngFrequencyLeft > 5000) And (mlngWaveLeft = 0) Then _
      mlngWaveLeft = 1
   If (mlngFrequencyRight > 5000) And (mlngWaveRight = 0) Then _
      mlngWaveRight = 1
   Piepsen
End Sub

Private Function Piepsen()
   Dim udtWaveForm      As waveformat_tag
   Dim lngDataLen       As Long
   Dim i                As Long
   Dim varLeftData      As Variant
   Dim varRightData     As Variant
   Dim lngMaxL          As Long
   Dim lngMaxR          As Long
   Dim lngLeft          As Long
   Dim lngRight         As Long
   Dim aintStereoData() As Integer

   mlngHeaderLength = Len(mudtHeader)

   ' Daten für linken Kanal erzeugen
   Select Case mlngWaveLeft
       Case 0
         varLeftData = mySinus(mlngFrequencyLeft, _
            mlngDurationLeft, mlngGaugeLeft)
      Case 1
         varLeftData = myDreieck(mlngFrequencyLeft, _
            mlngDurationLeft, mlngGaugeLeft)
      Case 2
         varLeftData = myRechteck(mlngFrequencyLeft, _
            mlngDurationLeft, mlngGaugeLeft)
   End Select

   ' Daten für rechten Kanal erzeugen
   Select Case mlngWaveRight
      Case 0
         varRightData = mySinus(mlngFrequencyRight, _
            mlngDurationRight, mlngGaugeRight)
      Case 1
         varRightData = myDreieck(mlngFrequencyRight, _
            mlngDurationRight, mlngGaugeRight)
      Case 2
         varRightData = myRechteck(mlngFrequencyRight, _
            mlngDurationRight, mlngGaugeRight)
   End Select
```

Listing 11.5 (Forts.)
Beispiele\11_Sound\
11_01_Sound.xls\clsWave

```
' Die maximale Anzahl der Abtastungen wird ermittelt
lngMaxL = UBound(varLeftData)
lngMaxR = UBound(varRightData)
If lngMaxL >= lngMaxR Then
   lngDataLen = lngMaxL
Else
   lngDataLen = lngMaxR
End If

' Ein Array in der Zielgröße wird initialisiert
ReDim aintStereoData(1 To lngDataLen * 2)

' Aus den zwei Arrays für jeden Kanal ein Array
' machen. Alle ungeraden Elemente entsprechen Kanal
' links, alle geraden Elemente Kanal rechts.
For i = 1 To lngDataLen
   lngLeft = 0: lngRight = 0
   If i <= lngMaxL Then lngLeft = varLeftData(i)
   If i <= lngMaxR Then lngRight = varRightData(i)
   aintStereoData(((i - 1) * 2) + 1) = lngLeft
   aintStereoData(((i - 1) * 2) + 2) = lngRight
Next

' Die Einstellungen setzen
With udtWaveForm
   .wFormatTag = WAVE_FORMAT_PCM
   .nChannels = 2 'Stereo
   .BitsPerSample = 16 '16 Bit (Integer)
   .nBlockAlign = 4 '2 Integer pro Block, also 4
   .nSamplesPerSec = 44100 'Abtastrate in Hz
   .nAvgBytesPerSec = 44100 'Abtastrate in Hz
End With

Beenden True

' Open Wave, als myWave wird ein Wavehandle zurückgegeben
waveOutOpen mlngWave, 0&, udtWaveForm, 0&, 0&, WAVE_ALLOWSYNC

' Speicher für Header anlegen
mlngMemHeader = GlobalAlloc((GMEM_MOVEABLE Or GMEM_SHARE), _
   mlngHeaderLength)
mlngPointerMemHeader = GlobalLock(mlngMemHeader)

' Speicher für Daten anlegen
mlngMemData = GlobalAlloc((GMEM_MOVEABLE Or GMEM_SHARE), _
   lngDataLen * 4)
mlngPointerMemData = GlobalLock(mlngMemData)

' Daten in den Datenbuffer kopieren
CopyMemory ByVal mlngPointerMemData, aintStereoData(1), _
   lngDataLen * 4

' Header initialisieren
With mudtHeader
   .dwBufferLength = lngDataLen * 2
   .dwFlags = (WHDR_BEGINLOOP Or WHDR_ENDLOOP)
   .dwLoops = mlngLoops
```

Listing 11.5 (Forts.)
Beispiele\11_Sound\
11_01_Sound.xls\clsWave

```
      .lpData = mlngPointerMemData 'Pointer auf den Datenblock
   End With

   ' Den Header in den Headerblock schreiben
   CopyMemory ByVal mlngPointerMemHeader, mudtHeader.lpData, _
      mlngHeaderLength

   ' Die Tonausgabe beginnen
   waveOutPrepareHeader mlngWave, ByVal mlngPointerMemHeader, _
      mlngHeaderLength
   waveOutWrite mlngWave, ByVal mlngPointerMemHeader, _
      mlngHeaderLength
End Function

Public Property Let AnzahlLoops(ByVal vNewValue As Long)
   mlngLoops = vNewValue
End Property

Public Property Let TondauerLinks(ByVal vNewValue As Long)
   mlngDurationLeft = vNewValue
End Property
Public Property Let TonfrequenzLinks(ByVal vNewValue As Long)
   mlngFrequencyLeft = vNewValue
End Property
Public Property Let PegelLinks(ByVal vNewValue As Long)
   mlngGaugeLeft = vNewValue
   If mlngGaugeLeft > 100 Then mlngGaugeLeft = 100
   If mlngGaugeLeft < 0 Then mlngGaugeLeft = 0
End Property
Public Property Let KurvenartLinks(ByVal vNewValue As Long)
   mlngWaveLeft = vNewValue
End Property

Public Property Let TondauerRechts(ByVal vNewValue As Long)
   mlngDurationRight = vNewValue
End Property
Public Property Let TonfrequenzRechts(ByVal vNewValue As Long)
   mlngFrequencyRight = vNewValue
End Property
Public Property Let PegelRechts(ByVal vNewValue As Long)
   mlngGaugeRight = vNewValue
   If mlngGaugeRight > 100 Then mlngGaugeRight = 100
   If mlngGaugeRight < 0 Then mlngGaugeRight = 0
End Property
Public Property Let KurvenartRechts(ByVal vNewValue As Long)
   mlngWaveRight = vNewValue
End Property

Private Function mySinus(lngFrequence As Long, _
   lngDuration As Long, lngVolume As Long)
   Dim aintData()       As Integer
   Dim lngLoops         As Long
   Dim lngPosition      As Long
   Dim lngBytesPeriod   As Long
   Dim dblNullPos       As Double
   Dim dblAngle         As Double
   Dim i                As Long
   Dim k                As Long
```

Listing 11.5 (Forts.)
Beispiele\11_Sound\
11_01_Sound.xls\clsWave

```
    If (lngFrequence = 0) Or (lngDuration = 0) Or (lngVolume = 0) Then
        ReDim aintData(1 To 1)
        mySinus = aintData
        Exit Function
    End If

    ' Um für die eingestellte Zeit einen Ton zu erzeugen,
    ' muss die Periode mehrmals in den Speicher geschrieben
    ' werden. Die Anzahl der Perioden ist die Variable
    ' lngLoops
    lngLoops = lngFrequence * lngDuration / 1000

    ' Für jede Periode bleiben so viel Werte
    lngBytesPeriod = (44100 / lngFrequence)

    ' Das Array wird so initialisiert, dass alle Werte hineinpassen
    ReDim Preserve aintData(1 To lngLoops * lngBytesPeriod)

    ' Der Nullpunkt (halber maximaler Pegel) ist abhängig vom
    ' eingestellten Pegel dieses Kanals
    dblNullPos = Int((&HFFFF& * lngVolume / 100) / 2)

    dblAngle = 360 / lngBytesPeriod
    For k = 1 To lngLoops

        For i = 1 To lngBytesPeriod

            ' Die Amplituden werden berechnet und ins Array
            ' geschrieben
            lngPosition = (i + (k - 1) * lngBytesPeriod)
            aintData(lngPosition) = "&H" & Hex((Sin(CDbl(i) * _
             dblAngle * 3.1416 / 180) + 1) * dblNullPos)

        Next

    Next

    mySinus = aintData
End Function

Private Function myDreieck(lngFrequence As Long, _
    lngDuration As Long, lngVolume As Long)
    Dim aintData()       As Integer
    Dim lngLoops         As Long
    Dim lngPosition      As Long
    Dim lngBytesPeriod   As Long
    Dim dblIncrement     As Double
    Dim dblHeight        As Double
    Dim i                As Long
    Dim k                As Long

    If (lngFrequence = 0) Or (lngDuration = 0) Or (lngVolume = 0) Then
        ReDim aintData(1 To 1)
        myDreieck = aintData
        Exit Function
    End If
```

Listing 11.5 (Forts.)
Beispiele\11_Sound\
11_01_Sound.xls\clsWave

```
    ' Um für die eingestellte Zeit einen Ton zu erzeugen,
    ' muss die Periode mehrmals in den Speicher geschrieben
    ' werden. Die Anzahl der Perioden ist die Variable
    ' lngLoops
    lngLoops = lngFrequence * lngDuration / 1000

    ' Für jede Periode bleiben so viel Werte
    lngBytesPeriod = (44100 / lngFrequence)
    ' Das Array wird so initialisiert, dass alle Werte hineinpassen
    ReDim Preserve aintData(1 To lngLoops * lngBytesPeriod)

    ' Der Pegelunterschied für jede Abtastung wird berechnet.
    ' Der maximale Pegel ist abhängig vom
    ' eingestellten Pegel dieses Kanals
    dblIncrement = (&HFFFF& * lngVolume / 100) / _
       (lngBytesPeriod / 2)
    For k = 1 To lngLoops
       dblHeight = 0

       For i = 1 To lngBytesPeriod

          ' Die Amplituden werden berechnet und ins Array geschrieben
          lngPosition = (i + (k - 1) * lngBytesPeriod)
          If i <= lngBytesPeriod / 2 Then
              dblHeight = dblHeight + dblIncrement
          Else
              dblHeight = dblHeight - dblIncrement
          End If
          If dblHeight > &HFFFF& Then dblHeight = &HFFFF&
          If dblHeight < 0 Then dblHeight = 0
          aintData(lngPosition) = "&H" & Hex(dblHeight)

       Next

    Next

    myDreieck = aintData
End Function

Private Function myRechteck(lngFrequence As Long, _
    lngDuration As Long, lngVolume As Long)

    Dim aintData()       As Integer
    Dim lngLoops         As Long
    Dim lngPosition      As Long
    Dim lngBytesPeriod   As Long
    Dim dblIncrement     As Double
    Dim dblHeight        As Double
    Dim i                As Long
    Dim k                As Long

    If (lngFrequence = 0) Or (lngDuration = 0) Or _
       (lngVolume = 0) Then
       ReDim aintData(1 To 1)
       myRechteck = aintData
       Exit Function
    End If
```

Listing 11.5 (Forts.)
Beispiele\11_Sound\
11_01_Sound.xls\clsWave

```
    ' Der maximale Pegel wird berechnet. Ist abhängig vom
    ' eingestellten Pegel dieses Kanals
    dblIncrement = (&HFFFF& * lngVolume) / 100

    ' Um für die eingestellte Zeit einen Ton zu erzeugen,
    ' muss die Periode mehrmals in den Speicher geschrieben
    ' werden. Die Anzahl der Perioden ist die Variable
    ' lngLoops
    lngLoops = lngFrequence * lngDuration / 1000

    ' Für jede Periode bleiben so viel Werte
    lngBytesPeriod = (44100 / lngFrequence)

    ' Das Array wird so initialisiert, dass alle Werte hineinpassen
    ReDim Preserve aintData(1 To lngLoops * lngBytesPeriod)

    For k = 1 To lngLoops

       For i = 1 To lngBytesPeriod
          ' Die Amplituden werden berechnet und ins Array geschrieben
          dblHeight = 0
          lngPosition = (i + (k - 1) * lngBytesPeriod)
          If i <= lngBytesPeriod / 2 Then dblHeight = dblIncrement
          If dblHeight > &HFFFF& Then dblHeight = &HFFFF&
          If dblHeight < 0 Then dblHeight = 0
          aintData(lngPosition) = "&H" & Hex(dblHeight)

       Next

    Next
myRechteck = aintData
End Function

Public Sub Beenden(Optional blnNow As Boolean)
    Dim lngRet As Long

    If blnNow Then
       ' blnNow Beenden
       waveOutReset mlngWave
    Else
       ' Auf das Beenden warten
       Do
          lngRet = waveOutUnprepareHeader(mlngWave, _
             ByVal mlngPointerMemHeader, mlngHeaderLength)
       Loop While ((lngRet = WAVERR_STILLPLAYING) And _
          (lngRet <> MMSYSERR_INVALHANDLE))
    End If

    waveOutClose mlngWave

    ' Belegten Speicher freigeben
    GlobalUnlock mlngPointerMemData
    GlobalFree mlngMemData

    GlobalUnlock mlngPointerMemHeader
    GlobalFree mlngMemHeader

End Sub
```

Listing 11.5 (Forts.)
Beispiele\11_Sound\
11_01_Sound.xls\clsWave

```
Public Sub LautstärkeEinstellen()
   Dim lngLeft    As Long
   Dim lngRight   As Long
   Dim lngRet     As Long
   Dim lngVolume  As Long

   If mlngPercentLeft > 100 Then mlngPercentLeft = 100
   If mlngPercentRight > 100 Then mlngPercentRight = 100
   If mlngPercentLeft < 0 Then mlngPercentLeft = 0
   If mlngPercentRight < 0 Then mlngPercentRight = 0

   ' Die Lautstärke des Audio-Geräts wird geändert
   lngVolume = TwoWordToLong(mlngPercentRight, mlngPercentLeft)
   lngRet = waveOutSetVolume(mlngWave, lngVolume)
End Sub

Private Function TwoWordToLong(ByVal lngRight As Long, _
   ByVal lngLeft As Long) As Long
   ' Aus zwei Werten für links und rechts wird ein Long
   ' gemacht, das im niederwertigen Wort die Lautstärke
   ' des linken Kanals und im höherwertigen Wort die des
   ' rechten Kanals aufnimmt.

   If lngLeft > 100 Then lngLeft = 100
   If lngRight > 100 Then lngRight = 100
   If lngLeft < 0 Then lngLeft = 0
   If lngRight < 0 Then lngRight = 0

   lngLeft = CLng((CDbl(&HFFFF&) / 100) * lngLeft)
   lngRight = CLng((CDbl(&HFFFF&) / 100) * lngRight)

   TwoWordToLong = CLng("&H" & Hex$(lngRight) & _
       String(4 - Len(Hex$(lngLeft)), _
       Asc("0")) & Hex$(lngLeft))

End Function

Public Property Let LautstärkeLinks(ByVal vNewValue As Long)
   mlngPercentLeft = vNewValue
End Property
Public Property Let LautstärkeRechts(ByVal vNewValue As Long)
   mlngPercentRight = vNewValue
End Property

Private Sub Class_Initialize()
   mlngGaugeLeft = 100
   mlngGaugeRight = 100
End Sub
Private Sub Class_Terminate()
   Beenden
End Sub
```

Tonausgabe

Der eingestellte Ton wird ausgegeben.

AnzahlLoops

Hier wird angegeben, wie oft hintereinander der erzeugte Ton abgespielt wird. Meistens ist es besser, einen Ton nur in der Länge von einer Sekunde zu erzeugen und anschließend den Ton so oft abzuspielen, bis die gewünschte Gesamtdauer erreicht ist. Immerhin wird pro Sekunde erzeugten Tons ein Speicher von 176 KB benötigt. Außerdem erfordert das Berechnen in meiner Klasse relativ viel Rechenzeit. Dies kann sicher noch optimiert werden, indem beispielsweise die Daten einer Vollwelle zwischengespeichert werden, anstatt sie jedes Mal neu zu berechnen.

Beenden

Die Ausgabe des Tons wird sofort beendet, wenn der optionale Parameter den Wahrheitswert `True` hat, andernfalls wird das Ende einer Ausgabe abgewartet. Letzteres ist wichtig, wenn Sie einen Ton über mehrere Loops ausgeben wollen und nicht abrupt abbrechen wollen. Eventuell reservierter Speicher für Daten und Header wird freigegeben.

LautstärkeLinks

Die Lautstärke des linken Kanals des Audio-Geräts wird eingestellt. Die Änderungen werden aber erst durch das Ausführen der Methode `LautstärkeEinstellen` wirksam.

LautstärkeRechts

Die Lautstärke des rechten Kanals des Audio-Geräts wird eingestellt. Die Änderungen werden aber erst durch das Ausführen der Methode `LautstärkeEinstellen` wirksam.

LautstärkeEinstellen

Die eingestellte Lautstärke wird gesetzt. Dazu werden die Variablen `mlngPercentLeft` und `mlngPercentRight` an die Funktion `TwoWordToLong` übergeben. In dieser werden sie in je ein Wort (16 Bit) umgewandelt, und zwar so, dass 100% dem Wert `&HFFFF` entspricht. Daraus wird ein Longwert erzeugt, bei dem das höherwertige Wort die Lautstärke des rechten Kanals und das niederwertige Wort die Lautstärke des linken Kanals wiedergibt. Die API-Funktion `waveOutSetVolume` bekommt diesen Longwert anschließend übergeben und setzt entsprechend die Lautstärke.

TondauerRechts

Die Tondauer des rechten Kanals wird gesetzt.

TondauerLinks

Die Tondauer des linken Kanals wird gesetzt.

TonfrequenzRechts

Die Tonfrequenz des rechten Kanals wird gesetzt.

TonfrequenzLinks

Die Tonfrequenz des linken Kanals wird gesetzt.

PegelRechts

Die Lautstärke des rechten Kanals in Prozent wird gesetzt.

PegelLinks

Die Lautstärke des linken Kanals in Prozent wird gesetzt.

KurvenartRechts

Die Kurvenart des rechten Kanals wird gesetzt. 0 ist Sinus, 1 ist Dreieck und 2 ist Rechteck.

KurvenartLinks

Die Kurvenart des linken Kanals wird gesetzt. 0 ist Sinus, 1 ist Dreieck und 2 ist Rechteck.

Piepsen

Für die Initialisierung wird eine Variable des Typs `waveformat` mit den Einstellungen gefüllt.

```
Private Type waveformat_tag
   wFormatTag As Integer
   nChannels As Integer
   nSamplesPerSec As Long
   nAvgBytesPerSec As Long
   nBlockAlign As Integer
   BitsPerSample As Integer
End Type
```

- `wFormatTag`

  Art der Modulation, hier Pulscodemodulation, also WAVE_FORMAT_PCM
- `nChannels`

  Anzahl der Kanäle, hier Stereo, also 2
- `nSamplesPerSec`

  Abtastrate, hier die eingestellte Frequenz, also 44100
- `BitsPerSample`

  Auflösung pro Sample, Anzahl der Bits für jeden Wert, hier 16
- `nBlockAlign`

  Blockgröße, hier 16-Bit-Stereo, somit der Wert 4 (Bytes)
- `nAvgBytesPerSec`

  Abtastrate, hier die eingestellte Frequenz, also 44100

Mit der Funktion `waveOutOpen` wird ein Audio-Ausgabegerät geöffnet. Daraufhin wird ein Speicherbereich für den `Waveheader` und den Datenbereich angelegt und gesperrt, anschließend werden die eigentlichen Daten erzeugt.

Die Kurvenform ist eine Folge von Werten im Speicher, die jeweils den Momentanwert der Kurve an dieser Stelle beschreiben. Jedes Wort (`Integer`) stellt bei einer Auflösung von 16 Bit einen Punkt der Kurve dar. Wichtig ist dabei auch die Abtastrate; das sind die Samples per Sekunde, die ich auf 44.100 Hz fest eingestellt habe.

Wollen Sie bei diesem Beispiel also eine Sinuskurve der Frequenz von 1 Hz erzeugen und haben eine Abtastrate von 44 KHz eingestellt, so haben Sie 44.100 einzelne Punkte zur Verfügung, um den Sinus im Speicher abzubilden. Bei 1.000 Hz Sinus sind es nur noch 44 Punkte für eine Vollwelle, bei 10 KHz nur noch vier. Sie sehen also, dass ab einer bestimmten Frequenz gar keine Möglichkeit mehr besteht, einen Sinus sauber abzubilden, deshalb habe ich einen Sinus nur bis 5 KHz zugelassen, aber selbst bis zu diesem Bereich wird der Klirrfaktor so richtig mies sein. Danach wird Dreieck benutzt, auch wenn Sinus eingestellt ist.

Um die Daten zu erzeugen, werden je nach gewünschter Kurvenform die Funktionen `myRechteck`, `myDreieck` und `mySinus` benutzt.

Wenn Sie für jeden Kanal ein Array mit den Werten angelegt haben, muss daraus noch ein gemeinsames Array gemacht werden, wobei für jeden Schritt erst ein `Integer` für den linken und danach ein Integer für den rechten Kanal kommt. Zum Schluss ergibt sich also ein Array mit der doppelten Länge des größten Ausgangsarrays; das sind pro Sekunde 88.200 Integerwerte, also 176.400 Bytes. Die Daten werden daraufhin mit `CopyMemory` in den angelegten Datenbereich kopiert.

Für die Ausgabe wird noch eine Struktur des Typs `WaveHdr` benötigt.

```
Private Type WaveHdr
   lpData As Long
   dwBufferLength As Long
   dwBytesRecorded As Long
   dwUser As Long
   dwFlags As Long
   dwLoops As Long
   lpNext As Long
   reserved As Long
End Type
```

Diese Struktur ist der Header und wird mit den Informationen über Datenlänge, Anzahl der Durchläufe und einem Zeiger auf den Datenblock gefüllt. Außerdem werden noch ein paar Flags gesetzt, und zwar `WHDR_BEGINLOOP Or WHDR_ENDLOOP`. Dieser Header wird dann mit `CopyMemory` in den angelegten Headerbereich kopiert.

Daraufhin beginnt die Tonausgabe mit den Funktionen `waveOutPrepareHeader` und `waveOutWrite`. Als Übergabeparameter brauchen beide Funktionen noch das `Wavehandle`; das ist der Zeiger auf den Speicherbereich, ab dem der Header steht, und schließlich die Länge des Headers selbst.

mySinus

Der Vollkreis hat 360 Grad. Diese 360 Grad werden durch die zur Verfügung stehenden Samples pro Vollwelle geteilt und man erhält für jeden Sample einen Winkel. Mithilfe der Sinusfunktion wird für jeden dieser Winkel ein Wert für die Amplitude berechnet.

Der maximal zur Verfügung stehende Wert ist bei einem nicht vorzeichenbehafteten Integer 65.535 das sind die sechzehn Bit (`&HFFFF`), die Sinusfunktion liefert aber Werte von –1 bis +1. Um den aktuellen Pegel im Wertebereich von 0 bis 65.535 abzubilden, addiere ich zum Sinuswert die Zahl 1, habe somit Werte von null bis zwei, die mit 32.767 (`&H7FFF`) multipliziert werden.

Der vorherige Nullpunkt liegt jetzt somit bei der Hälfte des Maximalpegels, der bei 100% Pegel also einen Wert von 32.767 hat. Vorher wird aber der Nullpunkt an den eingestellten Pegel (in Prozent) für diesen Kanal angepasst. Der Prozentwert wird vom Parameter `Lautstärke` geliefert. Leider können an einen VBA-`Integer` keine Werte übergeben werden, die größer als 32.767 sind, ohne einen Überlauf zu bekommen, deshalb wird der Wert in einen Hexstring umgewandelt und dann zugewiesen.

myDreieck

Für die Dreieckkurve wird der maximale Pegel durch die Hälfte der zur Verfügung stehenden Samples pro Vollwelle geteilt. In der ersten Hälfte der Vollwelle steigt der Pegel um diesen Betrag zum Maximum, um bis zum Ende auf null abzufallen. Vorher wird aber der maximale Pegel an den eingestellten Pegel (in Prozent) für diesen Kanal angepasst. Der Prozentwert wird vom Parameter `Lautstärke` geliefert.

myRechteck

In der ersten Hälfte der Vollwelle ist der Pegel auf dem Maximalwert und dann bis zum Ende auf dem Wert null. Vorher wird aber der maximale Pegel an den eingestellten Pegel (in Prozent) für diesen Kanal angepasst. Der Prozentwert wird vom Parameter `Lautstärke` geliefert.

11.7 Töne mit MIDI

Im vorherigen Beispiel mussten die Töne auf die harte Tour erzeugt werden. Dafür kann dort aber im Prinzip jede beliebige Wellenform hergestellt und abgespielt werden. Wer aber »nur« die Töne der Tonleiter in verschiedenen Oktaven abspielen will, kann es auch einfacher haben. Mit den Midifunktionen der *winmm.dll* haben Sie sogar die Möglichkeit, verschiedene Instrumente nachzuahmen.

Da ich in Sachen Musik ziemlich unbedarft bin, ist es mir nicht allzu leicht gefallen, mich mit Halbtönen, Oktaven und dem ganzen Kram auseinander zu setzen. Frequenzangaben sind mir da als Techniker schon lieber. Also habe ich erst einmal versucht, die Tonleiter mitsamt ihren Oktaven in Frequenzangaben umzuwandeln, um mir eine Übersicht zu verschaffen.

Ich denke, die Grundzüge habe ich jetzt einigermaßen verstanden und will Ihnen mein hinzugewonnenes Wissen auch nicht vorenthalten. Ich hoffe, dass ich in den folgenden Ausführungen nicht zu weit danebenliege. Aber selbst wenn meine Theorien doch nicht so stimmen sollten, ändert das glücklicher-

weise nicht viel an dem Benutzen der API-Funktionen. Zum Piepsen langt es allemal. Hier also meine geistigen Ergüsse über Halbtöne, Oktaven und die zugehörigen Frequenzen:

Der Grundton ist der Kammerton A in der vierten Oktave. Die Frequenz dieses Tones liegt bei 440 Hertz. Der gleiche Halbton der nächsthöheren Oktave ist doppelt so hoch, der einer Oktave niedriger ist nur halb so groß. In Zahlen ausgedrückt bedeutet dies, dass die Frequenz des Tons A in der Oktave drei bei 220 Hertz und in der Oktave fünf bei 880 Hertz liegt.

In jeder Oktave gibt es zwölf Halbtöne, der Ton A ist jeweils der zehnte. Die einzelnen Töne haben ein gleiches Frequenzverhältnis zueinander, deshalb kann der Abstand der Oktaven nicht einfach durch zwölf geteilt werden. Vielmehr muss ein Faktor herausbekommen werden, mit dem eine Tonfrequenz multipliziert wird, um zum nächsthöheren Ton zu kommen.

Um zur Lösung zu gelangen, muss man die Frage beantworten, welche Zahl, die zwölfmal mit sich selbst multipliziert, zwei ergibt. Also $x^{12} = 2$, das ergibt für x die zwölfte Wurzel aus zwei, oder $2^{(1/12)}$. Das Ergebnis liegt bei etwa 1,05946309435... und noch viele Stellen mehr. Jetzt kann die Frequenz des ersten Tons der ersten Oktave als Grundfrequenz ausgerechnet werden. Man kommt dabei auf 32,7032 Hertz. Die anderen Frequenzen können nun nach folgender Formel berechnet werden.

$$\text{Grundfrequenz} * 2^{(\text{Oktave} - 1)} * (2^{(1/12)})^{\text{Ton}-1}$$

Nachfolgend eine Tabelle (Tabelle 11.1) mit den errechneten Frequenzen. Dabei sind die deutschen Bezeichnungen für die einzelnen Oktaven benutzt worden.

	Kontra (1)	Große (2)	Kleine (3)	Oktave 1 (4)	Oktave 2 (5)	Oktave 3 (6)	Oktave 4 (7)	Oktave 5 (8)	Oktave 6 (9)
C	32,70	65,41	130,81	261,63	523,25	1046,50	2093,00	4186,01	8372,02
Cis	34,65	69,30	138,59	277,18	554,37	1108,73	2217,46	4434,92	8869,85
D	36,71	73,42	146,83	293,66	587,33	1174,66	2349,32	4698,64	9397,27
Dis	38,89	77,78	155,56	311,13	622,25	1244,51	2489,02	4978,03	9956,06
E	41,20	82,41	164,81	329,63	659,26	1318,51	2637,02	5274,04	10548,08
F	43,65	87,31	174,61	349,23	698,46	1396,91	2793,83	5587,65	11175,30
Fis	46,25	92,50	185,00	369,99	739,99	1479,98	2959,96	5919,91	11839,82
G	49,00	98,00	196,00	392,00	783,99	1567,98	3135,96	6271,93	12543,86
Gis	51,91	103,83	207,65	415,30	830,61	1661,22	3322,44	6644,88	13289,75
A	55,00	110,00	220,00	440,00	880,00	1760,00	3520,00	7040,00	14080,00
Ais	58,27	116,54	233,08	466,16	932,33	1864,66	3729,31	7458,62	14917,24
H	61,74	123,47	246,94	493,88	987,77	1975,53	3951,07	7902,13	15804,27

Tabelle 11.1 Frequenzliste (deutsche Bezeichnungen für Oktave)

Listing 11.6
Beispiele\11_Sound\
11_02_Midi.xls\mdlMidi

```
Private Declare Function midiOutOpen _
   Lib "winmm.dll" ( _
   lplngMidiOut As Long, _
   ByVal uDeviceID As Long, _
   ByVal dwCallback As Long, _
   ByVal dwInstance As Long, _
   ByVal dwFlags As Long _
   ) As Long

Private Declare Function midiOutShortMsg _
   Lib "winmm.dll" ( _
   ByVal lngMidiOut As Long, _
   ByVal dwMsg As Long _
   ) As Long

Private Declare Function midiOutClose _
   Lib "winmm.dll" ( _
   ByVal lngMidiOut As Long _
   ) As Long

Declare Sub Sleep _
   Lib "kernel32" ( _
   ByVal dwMilliseconds As Long)

Private Const MIDI_MAPPER = -1
Private Const NOTE_OFF = &H80
Private Const NOTE_ON = &H90
Private Const PROGRAM_CHANGE = &HC0
Private Const CALLBACK_NULL = &H0

Private mastrInstruments(1 To 128) As String

Sub test()
Dim lRet As Long, i As Long

    ' Ton 1=C, Ton 2=Cis, Ton 3=D, Ton 4=Dis
    ' Ton 5=E, Ton 6=F, Ton 7=Fis, Ton 8=G
    ' Ton 9=Gis, Ton 10=A, Ton 11=B, Ton 12=H
    ' Kammerton A, 4. Oktave (440 Hz), Glockenspiel
    ' Lautstärke 50, Dauer 1000 mSekunden, Kanal 2
    ' MsgBox PlayMidi(10, 4, 10, 100, 1000, 2)
     CreateInstruments

     For i = 1 To UBound(mastrInstruments)

       If mastrInstruments(i) = "" Then _
          mastrInstruments(i) = "?"

       lRet = MsgBox(mastrInstruments(i), vbOKCancel, _
          "Instrument " & i)
       If lRet = vbCancel Then Exit For

       PlayMidi 10, 4, i, 100, 500, 2

     Next

End Sub
```

Listing 11.6 (Forts.)
Beispiele\11_Sound\
11_02_Midi.xls\mdlMidi

```
Public Function PlayMidi( _
   lngNote As Long, _
   Optional lngOktave As Long = 4, _
   Optional lngInstrument As Long, _
   Optional lngVolume As Long = 100, _
   Optional lngDuration As Long = 1000, _
   Optional lngChannel As Long _
   ) As Double

   Dim lngMidi    As Long
   Dim lngMessage As Long
   Dim lngMyNote  As Long

   ' Wenn außerhalb des Bereichs, Kammerton A benutzen
   If (lngNote = 0) Or (lngNote > 12) Then lngNote = 10
   If (lngOktave = 0) Or (lngOktave > 9) Then lngOktave = 4

   ' Frequenz berechnen
   PlayMidi = 32.7032 * 2 ^ (lngOktave - 1) * _
      (2 ^ (1 / 12)) ^ (lngNote - 1)

   ' Position des Tons bestimmen
   lngMyNote = (lngOktave - 1) * 12 + lngNote

   ' Midiausgabe initialisieren und Handle holen
   midiOutOpen lngMidi, MIDI_MAPPER, 0, 0, CALLBACK_NULL

   If lngInstrument > 0 Then

      ' Message zum Instrumentwechsel erzeugen
      lngMessage = lngChannel + PROGRAM_CHANGE + _
         (lngInstrument - 1) * &H100

      ' Neues Instrument
      midiOutShortMsg lngMidi, lngMessage

      DoEvents

   End If

   ' Message zum Starten erzeugen
   lngMessage = (lngChannel + NOTE_ON) _
      + (lngMyNote * &H100) + (lngVolume * &H10000)

   ' Ausgabe starten
   midiOutShortMsg lngMidi, lngMessage

   ' Warten, bis Soundausgabe beendet ist
   Sleep lngDuration

   ' Schließen
   midiOutClose lngMidi

End Function
```

Listing 11.6 (Forts.) Beispiele\11_Sound\11_02_Midi.xls\mdlMidi

```
Private Sub CreateInstruments()
   mastrInstruments(1) = "Flügel, Konzert"
   mastrInstruments(2) = "Klavier"
   mastrInstruments(3) = "Flügel, Elektrisch"
   mastrInstruments(4) = "Piano, Honkeytonk"
   mastrInstruments(5) = "Piano, Rhodes"
   mastrInstruments(6) = "Piano, Chorus"
   mastrInstruments(7) = "Cembalo"
   mastrInstruments(8) = "Clavinet"
   mastrInstruments(9) = "Celesta"
   mastrInstruments(10) = "Glockenspiel"
   mastrInstruments(11) = "Musikbox"
   mastrInstruments(12) = "Vibraphon"
   mastrInstruments(13) = "Marimba"
   mastrInstruments(14) = "Xylophon"
   mastrInstruments(15) = "Röhrenglocken"
   mastrInstruments(16) = "Dulcimer"
   mastrInstruments(20) = "Kirchenorgel"
   mastrInstruments(25) = "Gitarre"
   mastrInstruments(32) = "E-Gitarre"
   mastrInstruments(48) = "Trommel"
   mastrInstruments(116) = "Drum"
   mastrInstruments(117) = "Pauke1"
   mastrInstruments(118) = "Pauke2"
   mastrInstruments(119) = "Drum"
   mastrInstruments(123) = "Wellen"
   mastrInstruments(124) = "Flöte"
   mastrInstruments(125) = "Klingel"
   mastrInstruments(126) = "Wind"
   mastrInstruments(127) = "Rauschen"
   mastrInstruments(128) = "Schuss"
End Sub
```

Benutzen der Funktion PlayMidi

In der Prozedur `Test` wird die Prozedur `CreateInstruments` aufgerufen. Dort wird ein Datenfeld erstellt, welches die Namen der mir bekannten Instrumente aufnimmt. Vielleicht gibt es irgendwo in den Tiefen des Internets oder der MSDN eine Liste mit den offiziellen Namen, ich habe aber keine gefunden und somit zum Teil eigene Namen kreiert.

Sie können, wenn Sie Lust haben, die Liste beliebig abändern und neue Namen vergeben. Ich habe 128 verschiedene Instrumente ausgemacht. In einer Schleife wird das Datenfeld nacheinander durchlaufen und die Instrumentennummer mitsamt den eventuell vorhandenen Namen in einer `Msgbox` ausgegeben. Wenn Sie dort auf OK klicken, wird dieser Ton abgespielt.

Funktion PlayMidi

In der Funktion `PlayMidi` wird sichergestellt, dass der übergebene Ton im Bereich 1 bis 12 liegt. Dazu wird der Parameter auf zehn gesetzt, wenn er außerhalb des Wertebereichs liegt. Der Parameter Oktave wird auf den Wertebereich eins bis neun begrenzt. Er wird auf vier gesetzt, wenn er außerhalb liegen sollte. Der Defaultwert ist also der Kammerton A mit 440 Hz.

Der Rückgabewert der Funktion ist die Frequenz des Tons. Dieser wird aus den Parametern `Ton` und `Oktave` berechnet.

```
PlayMidi = 32.7032 * 2 ^ (lngOktave - 1) * _
   (2 ^ (1 / 12)) ^ (lngNote - 1)
```

Als Nächstes benötigen Sie den Index des Tons. Das ist im Prinzip die Position in der aufsteigend sortierten Frequenzliste.

Index	Frequenz	Index	Frequenz	Index	Frequenz	Index	Frequenz	Index	Frequenz
1	32,70	25	130,81	49	523,25	73	2093,00	97	8372,02
2	34,65	26	138,59	50	554,37	74	2217,46	98	8869,85
3	36,71	27	146,83	51	587,33	75	2349,32	99	9397,27
4	38,89	28	155,56	52	622,25	76	2489,02	100	9956,06
5	41,20	29	164,81	53	659,26	77	2637,02	101	10548,08
6	43,65	30	174,61	54	698,46	78	2793,83	102	11175,30
7	46,25	31	185,00	55	739,99	79	2959,96	103	11839,82
8	49,00	32	196,00	56	783,99	80	3135,96	104	12543,86
9	51,91	33	207,65	57	830,61	81	3322,44	105	13289,75
10	55,00	34	220,00	58	880,00	82	3520,00	106	14080,00
11	58,27	35	233,08	59	932,33	83	3729,31	107	14917,24
12	61,74	36	246,94	60	987,77	84	3951,07	108	15804,27
13	65,41	37	261,63	61	1046,5	85	4186,01		
14	69,30	38	277,18	62	1108,73	86	4434,92		
15	73,42	39	293,66	63	1174,66	87	4698,64		
16	77,78	40	311,13	64	1244,51	88	4978,03		
17	82,41	41	329,63	65	1318,51	89	5274,04		
18	87,31	42	349,23	66	1396,91	90	5587,65		
19	92,50	43	369,99	67	1479,98	91	5919,91		
20	98,00	44	392,00	68	1567,98	92	6271,93		
21	103,83	45	415,30	69	1661,22	93	6644,88		
22	110,00	46	440,00	70	1760,00	94	7040,00		
23	116,54	47	466,16	71	1864,66	95	7458,62		
24	123,47	48	493,88	72	1975,53	96	7902,13		

Tabelle 11.2 Der Index und die zugehörige Frequenz

Die Variable `lngMyNote` nimmt diesen Index auf.

```
lngMyNote = (lngOktave - 1) * 12 + lngNote
```

Danach wird mit `midiOutOpen` die Midiausgabe initialisiert und das Handle `lngMidi` geholt. Die Ausgabe erfolgt mit der Message `PROGRAM_CHANGE` an die Funktion `midiOutShortMsg`. Dann wird mit `Sleep` so lange gewartet, bis die Soundausgabe beendet ist, in dieser Zeit können Sie aber mit Excel nicht weiterarbeiten.

Zum Schluss wird das Midihandle mit `midiOutClose` geschlossen.

12

UserForms

12.1 Was Sie in diesem Kapitel erwartet

Manchmal wünscht man sich UserForms, die ein etwas anderes Verhalten als die standardmäßigen Formulare der Bibliothek `MSForms` an den Tag legen. Mit dem in diesem Kapitel vorgestellten Code können Sie eine UserForm so anpassen, dass diese sich wie ein normales Fenster verhält. D.h., die UserForm lässt sich anschließend durch Ziehen am Rahmen oder auch durch das Systemmenü in der Größe anpassen. Außerdem kann die Titelleiste komplett ausgeblendet werden.

Durch das Manipulieren der Fensterregionen lassen sich UserForms in beliebigen Formen erzeugen, sogar Löcher sind ohne Probleme möglich. In einem Beispiel wird gezeigt, wie Sie eine runde UserForm mit einem Loch in der Mitte anlegen.

12.2 Min, Max, Resize

UserForms, die sich wie andere Fenster minimieren, maximieren und durch Ziehen am Rahmen in der Größe ändern lassen, sind durch die Manipulation der Fensterstile kein großes Problem. Außerdem ist es damit möglich, das Kreuz zum Schließen der Form, das sich in der Titelleiste rechts oben befindet, gar nicht erst anzuzeigen.

Fenster besitzen Stile, die ihr Verhalten und das Aussehen mitbestimmen. Man kann mit ein paar API-Funktionen bestimmte Stile eines Fensters setzen oder löschen. Eine UserForm ist im Gegensatz zu vielen rein grafischen Elementen von Excel ein echtes Fenster, besitzt Fensterstile und ist somit hervorragend zum Manipulieren geeignet.

Für das nachstehende Beispiel fügen Sie in der Entwicklungsumgebung VBE von Excel eine UserForm ein. Dort hinein gehören fünf CommandButtons mit den Namen `cmdMaximize`, `cmdMinimize`, `cmdSysmenü`, `cmdResize` und `cmdTitle`. In das Klassenmodul dieser Form schreiben Sie den nachstehenden Code.

Listing 12.1
Beispiele\12_UserForms\
12_01_MinMaxResizeRound.xls\
ufMinMax

```
Private Declare Function GetWindowLong _
   Lib "user32" Alias "GetWindowLongA" ( _
   ByVal hwnd As Long, _
   ByVal nIndex As Long _
   ) As Long

Private Declare Function SetWindowLong _
   Lib "user32" Alias "SetWindowLongA" ( _
   ByVal hwnd As Long, _
   ByVal nIndex As Long, _
   ByVal dwNewLong As Long _
   ) As Long

Private Declare Function FindWindowA _
   Lib "user32" ( _
   ByVal lpClassName As String, _
   ByVal lpWindowName As String _
   ) As Long

Private Declare Function DrawMenuBar _
   Lib "user32" ( _
   ByVal hwnd As Long _
   ) As Long

Private Const WS_MAXIMIZEBOX = &H10000
Private Const WS_MINIMIZEBOX = &H20000
Private Const WS_SYSMENU = &H80000
Private Const WS_THICKFRAME = &H40000
Private Const WS_DLGFRAME = &H400000
Private Const GWL_STYLE = (-16)

Private mlngStyle As Long
Private mlngFormHandle As Long

Private Sub ChangeStyle()

      ' Den geänderten Stil setzen
      SetWindowLong mlngFormHandle, GWL_STYLE, mlngStyle

      ' Menübar neu zeichnen
      DrawMenuBar mlngFormHandle

End Sub

Private Sub InitMe()
   Dim strTitle As String

   If mlngFormHandle = 0 Then

      ' Alten Titel speichern
      strTitle = Me.Caption

      ' Eindeutigen Titel (Caption) vergeben
      Me.Caption = "lhdsgterfsdt"

      ' Das Fenster mit diesem Titel suchen
      mlngFormHandle = FindWindowA(vbNullString, "lhdsgterfsdt")
```

Listing 12.1 (Forts.)
Beispiele\12_UserForms\
12_01_MinMaxResizeRound.xls\
ufMinMax

```
        ' Alten Titel setzen
        Me.Caption = strTitle

        ' Die Fensterstile ermitteln
        mlngStyle = GetWindowLong(mlngFormHandle, GWL_STYLE)

    End If

End Sub

Private Sub cmdMaximize_Click()

    If mlngStyle And WS_MAXIMIZEBOX Then

        ' Stilbit WS_MAXIMIZEBOX löschen
        mlngStyle = mlngStyle And Not WS_MAXIMIZEBOX
        cmdMaximize.Caption = "Maximizebox EIN"

    Else

        ' Stilbit WS_MAXIMIZEBOX setzen
        mlngStyle = mlngStyle Or WS_MAXIMIZEBOX
        cmdMaximize.Caption = "Maximizebox AUS"

    End If

    ChangeStyle

End Sub

Private Sub cmdMinimize_Click()

    If mlngStyle And WS_MINIMIZEBOX Then

        ' Stilbit WS_MINIMIZEBOX löschen
        mlngStyle = mlngStyle And Not WS_MINIMIZEBOX
        cmdMinimize.Caption = "Minimizebox EIN"

    Else

        ' Stilbit WS_MINIMIZEBOX setzen
        mlngStyle = mlngStyle Or WS_MINIMIZEBOX
        cmdMinimize.Caption = "Minimizebox AUS"

    End If

    ChangeStyle

End Sub

Private Sub cmdSysmenü_Click()

    If mlngStyle And WS_SYSMENU Then

        ' Stilbit WS_SYSMENU löschen
        mlngStyle = mlngStyle And Not WS_SYSMENU
        cmdSysmenü.Caption = "Systemmenü EIN"
```

Listing 12.1 (Forts.)
Beispiele\12_UserForms\
12_01_MinMaxResizeRound.xls\
ufMinMax

```
    Else

        ' Stilbit WS_SYSMENU setzen
        mlngStyle = mlngStyle Or WS_SYSMENU
        cmdSysmenü.Caption = "Systemmenü AUS"

    End If

    ChangeStyle

End Sub

Private Sub cmdResize_Click()

    If mlngStyle And WS_THICKFRAME Then

        ' Stilbit WS_THICKFRAME löschen
        mlngStyle = mlngStyle And Not WS_THICKFRAME
        cmdResize.Caption = "Resize EIN"

    Else

        ' Stilbit WS_THICKFRAME setzen
        mlngStyle = mlngStyle Or WS_THICKFRAME
        cmdResize.Caption = "Resize AUS"

    End If

    ChangeStyle

End Sub

Private Sub cmdTitle_Click()

    If mlngStyle And WS_DLGFRAME Then

        ' Stilbit WS_DLGFRAME löschen
        mlngStyle = mlngStyle And Not WS_DLGFRAME
        cmdTitle.Caption = "Titelleiste EIN"

    Else

        ' Stilbit WS_DLGFRAME setzen
        mlngStyle = mlngStyle Or WS_DLGFRAME
        cmdTitle.Caption = "Titelleiste AUS"

    End If

    ChangeStyle

End Sub
```

Listing 12.1 (Forts.)
Beispiele\12_UserForms\
12_01_MinMaxResizeRound.xls\
ufMinMax

```
Private Sub UserForm_Activate()

    ' Fensterhandle ermitteln und Stile auslesen
    InitMe

End Sub

Private Sub UserForm_QueryClose(Cancel As Integer, _
    CloseMode As Integer)

    ' Trotz ausgeblendetem Systemmenü kann mit Alt/F4 die
    ' Form geschlossen werden. Das wird hiermit verhindert
    If (mlngStyle And WS_SYSMENU) = 0 Then Cancel = True

End Sub
```

So sieht das Ergebnis aus:

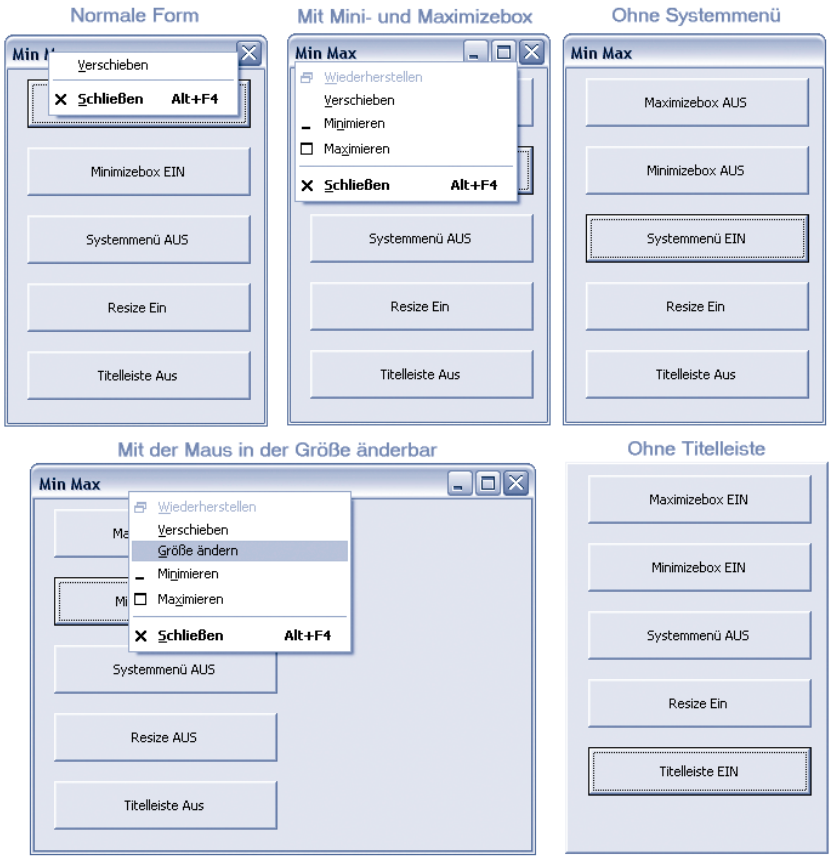


Abbildung 12.1
Verschiedene Stile der gleichen UserForm

UserForm_Activate

Beim Aktivieren der UserForm wird das Ereignis `UserForm_Activate` abgearbeitet. Dort wird die Prozedur `InitMe` aufgerufen.

InitMe

In dieser Prozedur wird beim ersten Aufruf das Fensterhandle der UserForm ermittelt. Das Handle wird hier nicht über den Klassennamen gesucht, weil diese sich schon einmal in den verschiedenen Officeversionen geändert haben und somit Schwierigkeiten auftreten könnten.

Allein der Fenstertext der UserForm wird zur Suche mit der API `FindWindowA` herangezogen, der Parameter für den Klassennamen wird dabei auf eine Nullzeichenfolge (`vbNullString`) gesetzt. Damit auch das richtige Fenster gefunden wird, verändere ich die Eigenschaft `Caption` der UserForm, indem ich eine eindeutige Zeichenfolge vergebe und den vorherigen Text zwischenspeichere. In diesem Beispiel benutze ich die Zeichenfolge »lhdsgterfsdt« zur Suche des Fensters.

Wenn das Handle gefunden wurde, wird die Caption auf den ursprünglichen Text zurückgesetzt und mit der Funktion `GetWindowLong` der aktuelle Fensterstil der UserForm abgefragt. Dazu wird als erster Parameter das Fensterhandle und als zweiter Parameter der Index `GWL_STYLE` an `GetWindowLong` übergeben. Als Funktionsergebnis erhalten Sie bei diesem Index einen Longwert zurück, der die Stile als Kombination von gesetzten Bits (Flags) enthält. Zugewiesen wird dieser zurückgelieferte Wert der Longvariablen `mlngStyle`, die auf Klassenebene deklariert und somit im gesamten Klassenmodul der UserForm verfügbar ist.

cmdSysmenü_Click

Bei einem Klick auf den Button CMDSYSMENÜ wird im Klickereignis die Variable `mlngStyle` daraufhin untersucht, ob das Flag `WS_SYSMENU` gesetzt oder gelöscht ist. Ist es gesetzt, wird es mit binärem `AND NOT` gelöscht, ist es nicht gesetzt, wird es mit einem binären `Or` gesetzt. Gleichzeitig wird auch die Caption des Buttons angepasst, damit diese immer anzeigt, welche Aktion durch einen Klick darauf ausgelöst werden kann. Danach wird die interne Prozedur `ChangeStyle` aufgerufen, die den Stil der UserForm ändert.

Der Stil `WS_SYSMENU` ist außer für das Aktivieren des Systemmenüs auch für das Anzeigen der drei Symbole in der rechten Ecke der Titelleiste zuständig. Ist er gesetzt, ist das Kreuzsymbol zum Schließen sichtbar. Wird es gelöscht, werden weder das Kreuz, noch eventuell eingeschaltete Minimierungs- oder Maximierungssymbole angezeigt.

cmdMaximize_Click

Bei einem Klick auf den Button CMDMAXIMIZE wird im Klickereignis die Variable `mlngStyle` daraufhin untersucht, ob das Flag `WS_MAXIMIZEBOX` gesetzt oder gelöscht ist. Ist es gesetzt, wird es mit binärem `AND NOT` gelöscht, ist es nicht gesetzt, wird es mit einem binären `Or` gesetzt. Gleichzeitig wird auch die

Caption des Buttons angepasst, damit diese immer anzeigt, welche Aktion durch einen Klick darauf ausgelöst werden kann. Danach wird die interne Prozedur `ChangeStyle` aufgerufen, die den Stil der UserForm ändert.

Der Stil `WS_MAXIMIZEBOX` ist zuständig für das Aktivieren des Maximieren-Symbols, beziehungsweise für das Minimieren-Symbol nach dem Maximieren. Auch bei einem nicht gesetzten Flag `WS_MINIMIZEBOX` erscheint das Symbol zum Minimieren, es wird aber deaktiviert dargestellt.

cmdMinimize_Click

Bei einem Klick auf den Button CMDMINIMIZE wird im Klickereignis die Variable `mlngStyle` daraufhin untersucht, ob das Flag `WS_MINIMIZEBOX` gesetzt oder gelöscht ist. Ist es gesetzt, wird es mit binärem `AND NOT` gelöscht, ist es nicht gesetzt, wird es mit einem binären `Or` gesetzt. Gleichzeitig wird auch die Caption des Buttons angepasst, damit diese immer anzeigt, welche Aktion durch einen Klick darauf ausgelöst werden kann. Danach wird die interne Prozedur `ChangeStyle` aufgerufen, die den Stil der UserForm ändert.

Der Stil `WS_MINIMIZEBOX` ist zuständig für das Aktivieren des Minimieren-Symbols. Wenn das Maximieren-Symbol eingeschaltet ist, erscheint auch bei dem nicht gesetzten Flag `WS_MINIMIZEBOX` das Symbol zum Minimieren, es wird aber deaktiviert dargestellt.

cmdResize_Click

Bei einem Klick auf den Button CMDRESIZE wird im Klickereignis die Variable `mlngStyle` daraufhin untersucht, ob das Flag `WS_THICKFRAME` gesetzt oder gelöscht ist. Ist es gesetzt, wird es mit binärem `AND NOT` gelöscht, ist es nicht gesetzt, wird es mit einem binären `Or` gesetzt. Gleichzeitig wird auch die Caption des Buttons angepasst, damit diese immer anzeigt, welche Aktion durch einen Klick darauf ausgelöst werden kann. Danach wird die interne Prozedur `ChangeStyle` aufgerufen, die den Stil der UserForm ändert.

Der Stil `WS_THICKFRAME` ist zuständig für den Rahmen der UserForm und bestimmt auch ihr Verhalten mit, unter anderem auch das der Größenänderung durch die Maus.

cmdTitle_Click

Bei einem Klick auf den Button CMDTITLE wird im Klickereignis die Variable `mlngStyle` daraufhin untersucht, ob das Flag `WS_DLGFRAME` gesetzt oder gelöscht ist. Ist es gesetzt, wird es mit binärem `AND NOT` gelöscht, ist es nicht gesetzt, wird es mit einem binären `Or` gesetzt. Gleichzeitig wird auch die Caption des Buttons angepasst, damit diese immer anzeigt, welche Aktion durch einen Klick darauf ausgelöst werden kann. Danach wird die interne Prozedur `ChangeStyle` aufgerufen, die den Stil der UserForm ändert.

Der Stil `WS_DLGFRAME` ist unter anderem zuständig für den Rahmen der UserForm. Wird dieses Flag gesetzt, besitzt das Fenster anschließend keine Titelleiste mehr.

ChangeStyle

Die Prozedur `ChangeStyle` steht für das Ändern des Fensterstils zur Verfügung. Mittels der Funktion `SetWindowLong` wird der Fensterstil angepasst.

Dazu wird als erster Parameter das Fensterhandle, als zweiter Parameter die Konstante `GWL_STYLE` und als dritter Parameter der Longwert übergeben, der die Fensterstile enthält. Damit die Änderung schließlich auch angezeigt wird, rufen Sie die Funktion `DrawMenuBar` auf. Diese bewirkt ein Neuzeichnen der Titelleiste.

UserForm_QueryClose

Das Schließen einer UserForm selbst lässt sich ja auch ohne die API recht einfach in der Ereignisprozedur `QueryClose` unterdrücken, indem dort `Cancel` auf `TRUE` gesetzt wird.

```
Private Sub UserForm_QueryClose( _
   Cancel As Integer, CloseMode As Integer)
   Cancel = True
End Sub
```

In dem vorliegenden Beispiel wird das Schließen des Fensters verhindert, wenn das Systemmenü ausgeblendet ist. Dazu wird nachgeschaut, ob das entsprechende Stil-Bit gesetzt ist. Das wird deshalb gemacht, weil die Tastenkombination Alt+F4 des Systemmenüs immer noch das Schließen des Fensters anstößt, obwohl das Kreuz schon nicht mehr sichtbar ist.

12.3 Runde UserForms mit Loch

Manchmal wünscht man sich Userformen, die eine etwas andere Gestalt als die standardmäßigen Fenster besitzen. Sei es, um aus der Masse herauszuragen, oder aber, weil man etwas ganz besonderes darstellen will oder muss. In vielen Anwendungen wird ja demonstriert, dass so etwas grundsätzlich möglich ist. Genau das sollte auch mit den Userformen der Officeanwendungen funktionieren, die ja schließlich auch nur Fenster sind. Und mit dem nötigen Know-How klappt das auch ganz vorzüglich.

Wenn man dabei die richtigen API-Funktionen einsetzt, kann man sich Userformen in jeder beliebigen Form basteln. Dabei macht man sich zunutze, dass der sichtbare Bereich eines Fensters eine Region ist.

Mit Hilfe verschiedener API-Funktionen wie CreateEllipticRgn, CreatePolygonRgn, CreatePolyPolygonRgn, CreateRectRgn oder CreateRoundRectRgn kann man runde, elliptische, polygone oder rechteckige Regionen erzeugen, die miteinander kombinieren werden können. Dabei kann sogar der überlappende Bereich zweier Regionen ausgestanzt, oder auch nur der gemeinsame Teilbereich benutzt werden.

Wird die ursprüngliche Region eines Fensters mit der erzeugten überschrieben, nimmt der sichtbare Bereich eines Fensters die Form der zugewiesenen Region an.

In dem nachfolgenden Beispiel wird zur Demonstration eine elliptische Region erzeugt und mit einer rechteckigen derart kombiniert, dass nur die nicht gemeinsamen Teile übrig bleiben. Damit kann man eine runde Userform mit einem rechteckigen Loch erzeugen.

Abbildung 12.2
UserForm mit normaler und veränderter Region

Fügen Sie in der Entwicklungsumgebung von Excel eine UserForm ein und schreiben Sie den nachfolgenden Code (Listing 12.2) in das Klassenmodul dieser Form.

Starten Sie anschließend die UserForm. Nach einem Klick mit der linken Maustaste müsste die Form rund werden und mit einem rechteckigem Loch in der Mitte versehen sein. Das Loch ist nicht nur durchsichtig, es verhält sich auch wie ein Loch, d.h., ein Mausklick geht an das darunter liegende Fenster, wenn die UserForm nicht modal aufgerufen wurde.

Wenn Sie Versionen > Excel 97 verwenden, können Sie die Form modeless aufrufen, sodass Sie ohne zusätzlichen Aufwand mit Excel weiterarbeiten können. UserForms werden aber unter Excel 97 grundsätzlich modal angezeigt, eine andere Option gibt es dabei nicht. Das modale Verhalten unter Excel 97 ist oft etwas hinderlich, dem kann aber durch ein paar API-Aufrufe abgeholfen werden.

Listing 12.2
Beispiele\12_UserForms\12_01_MinMaxResizeRound.xls\ufRund

```
Private Declare Function FindWindow _
   Lib "user32" Alias "FindWindowA" ( _
   ByVal lpClassName As String, _
   ByVal lpWindowName As String _
   ) As Long

Private Declare Function EnableWindow _
   Lib "user32" ( _
   ByVal hwnd As Long, _
   ByVal bEnable As Long _
   ) As Long

Private Declare Function GetWindowRect _
   Lib "user32" ( _
   ByVal hwnd As Long, _
   lpRect As RECT _
   ) As Long
```

Listing 12.2 (Forts.) Beispiele\12_UserForms\12_01_MinMaxResizeRound.xls\ufRund

```
Private Declare Function SetWindowRgn _
   Lib "user32" ( _
   ByVal hwnd As Long, _
   ByVal hRgn As Long, _
   ByVal bRedraw As Boolean _
   ) As Long

Private Declare Function CreateRoundRectRgn _
   Lib "gdi32" ( _
   ByVal X1 As Long, _
   ByVal Y1 As Long, _
   ByVal X2 As Long, _
   ByVal Y2 As Long, _
   ByVal X3 As Long, _
   ByVal Y3 As Long _
   ) As Long

Private Declare Function CreateRectRgn _
   Lib "gdi32" ( _
   ByVal X1 As Long, _
   ByVal Y1 As Long, _
   ByVal X2 As Long, _
   ByVal Y2 As Long _
   ) As Long

Private Declare Function CombineRgn _
   Lib "gdi32" ( _
   ByVal hDestRgn As Long, _
   ByVal hSrcRgn1 As Long, _
   ByVal hSrcRgn2 As Long, _
   ByVal nCombineMode As Long _
   ) As Long

Private Declare Function SendMessage _
   Lib "user32" Alias "SendMessageA" ( _
   ByVal hwnd As Long, _
   ByVal wMsg As Long, _
   ByVal wParam As Long, _
   lParam As Any _
   ) As Long

Private Declare Function DeleteObject _
   Lib "gdi32" ( _
   ByVal hObject As Long _
   ) As Long

Private Declare Function ReleaseCapture _
   Lib "user32" () As Long

Private Type RECT
   Left As Long
   Top As Long
   Right As Long
   Bottom As Long
End Type
```

Listing 12.2 (Forts.)
Beispiele\12_UserForms\
12_01_MinMaxResizeRound.xls\
ufRund

```
Private Const WM_NCLBUTTONDOWN = &HA1
Private Const HTCAPTION = 2
Private Const RGN_DIFF = 4

Private mlngFormRegion  As Long
Private mlngRegion1     As Long
Private mlngRegion2     As Long
Private mlngHwndForm    As Long
Private mlngHwndClient  As Long
Private mlngDummy       As Long

Private Sub UserForm_Click()
   Call Round
End Sub

Private Sub Round()
   Dim udtDimension     As RECT
   Dim udtDimension1    As RECT
   Dim lngPos1x         As Long
   Dim lngPos1y         As Long
   Dim lngPos2x         As Long
   Dim lngPos2y         As Long
   Dim lngPos3x         As Long
   Dim lngPos3y         As Long

   If mlngFormRegion <> 0 Then Exit Sub

   ' Handle auf die UserForm
   mlngHwndForm = GetMyHwnd()

   ' Größe und Rahmen festlegen
   Me.BorderStyle = fmBorderStyleNone
   Me.Width = 100
   Me.Height = 125

   ' Größe in Pixel ermitteln
   GetWindowRect mlngHwndForm, udtDimension

   ' Rect-Struktur für Region 1 ausfüllen
   With udtDimension
      lngPos1x = 3
      lngPos1y = 30
      lngPos2x = (.Right - .Left) - 3
      lngPos2y = (.Bottom - .Top) - 3
      lngPos3x = lngPos2x
      lngPos3y = lngPos2y
   End With

   ' Runde Region 1 erstellen
   mlngRegion1 = CreateRoundRectRgn( _
      lngPos1x, lngPos1y, _
      lngPos2x, lngPos2y, _
      lngPos3x, lngPos3y)

   ' Eckpunkte für Region 2 festlegen
   lngPos1x = lngPos2x \ 2 - 20
   lngPos1y = lngPos2y \ 2 + 15 - 20
```

Listing 12.2 (Forts.)
Beispiele\12_UserForms\12_01_MinMaxResizeRound.xls\ufRund

```
    lngPos2x = lngPos2x \ 2 + 20
    lngPos2y = lngPos2y \ 2 + 15 + 20

    ' Rechteckige Region 2 erstellen
    mlngRegion2 = CreateRectRgn( _
       lngPos1x, lngPos1y, _
       lngPos2x, lngPos2y)

    ' Beide zu Region 1 kombinieren, gemeinsames
    ' wird entfernt
    CombineRgn mlngRegion1, mlngRegion1, mlngRegion2, RGN_DIFF

    ' Fensterregion ändern
    mlngFormRegion = SetWindowRgn(mlngHwndForm, mlngRegion1, True)

    ' Größe ermitteln
    GetWindowRect mlngHwndForm, udtDimension

End Sub

Private Function GetMyHwnd()
    Dim strCaption          As String
    Dim strSearchCaption    As String

    ' Suchstring festlegen
    strSearchCaption = "Ich werd dich schon kriegen"

    ' Alte Caption speichern
    strCaption = Me.Caption

    ' Suchstring als Caption
    Me.Caption = strSearchCaption

    ' Handle ermitteln
    GetMyHwnd = FindWindow(vbNullString, strSearchCaption)

    ' Alte Caption zurück
    Me.Caption = strCaption

End Function

Private Sub UserForm_MouseDown( _
    ByVal Button As Integer, _
    ByVal Shift As Integer, _
    ByVal X As Single, _
    ByVal Y As Single)

    If Button = 1 Then
        ' Linke Maustaste
        If mlngHwndForm <> 0 Then
            ' Form wurde schon geändert

            ' Verschieben
            ReleaseCapture
            ' Klick auf Titelleiste simulieren
            SendMessage mlngHwndForm, WM_NCLBUTTONDOWN, HTCAPTION, 0
        End If
```

Listing 12.2 (Forts.)
Beispiele\12_UserForms\
12_01_MinMaxResizeRound.xls\
ufRund

```
    Else

        'Zum Beenden
        Unload Me

    End If

End Sub


Private Sub UserForm_Terminate()

    ' Aufräumen und Regionen zerstören
    DeleteObject mlngRegion1
    DeleteObject mlngRegion2
    DeleteObject mlngFormRegion

End Sub

Private Sub UserForm_Activate()

    #If VBA6 = 0 Then
        Dim hwndXL As Long
        ' Zugriffsnummer Excel ermitteln
        hwndXL = FindWindowA("XLMAIN", Application.Caption)
        ' Form modeless machen (nur Excel 97)
        ' Danach können Sie wie gewohnt mit dem Tabellenblatt
        ' arbeiten. Ein Problem gibt es: Um mit den Menüs zu
        ' arbeiten, muss einmal ein Doppelklick auf irgendein
        ' Menü oder CommandButton ausgeführt werden. Dann geht
        ' der Dialog Anpassen auf. Wenn er weggeklickt wird,
        ' ist alles in Ordnung.
        EnableWindow hwndXL, True
    #End If

End Sub
```

UserForm_Click

Wenn Sie mit der linken Maustaste auf die UserForm klicken, wird über das Klickereignis die Prozedur `Round` aufgerufen.

Round

In dieser Prozedur wird überprüft, ob die auf Klassenebene deklarierte Variable `mlngFormRegion` schon einen Wert enthält. Wenn ja, wurde die Form bereits verändert und die Prozedur wird verlassen.

Ist die Form noch jungfräulich – also die Variable `mlngFormRegion` gleich null –, wird die Größe der Form so angepasst, dass nachher eine schöne runde Form entsteht. Bei den Abmessungen sind der Fantasie aber prinzipiell keine Grenzen gesetzt und die Werte sollten auch ruhig mal geändert werden. Nur durch Probieren können Sie ein Gefühl dafür bekommen, wie die Funktionen zum Erzeugen von Regionen arbeiten.

Die Hauptfensternummer der UserForm wird mit der Unterfunktion `GetMyHwnd` ermittelt. Wenn Sie die Fensternummer schließlich ermittelt haben, holen Sie sich mithilfe der Funktion `GetWindowRect` die Größe und Position der Form in Pixel. Dazu wird die Struktur `Rect` mit der Position der linken oberen und rechten unteren Ecke gefüllt.

Anschließend werden die Parameter für die runde Region gesetzt, diese an die API `CreateRoundRectRgn` übergeben und es wird damit die erste Region erzeugt.

Danach werden die Parameter für die rechteckige Region gesetzt, an die API `CreateRectRgn` übergeben und die zweite Region erzeugt. Diese stellt nachher das rechteckige Loch dar.

Beide Regionen werden daraufhin mit `CombineRgn` und dem auf `RGN_DIFF` gesetzten Parameter `nCombineMode` so kombiniert, dass nur die Differenz beider Regionen übrig bleibt. Differenz bedeutet in dem Fall, dass die gemeinsamen Teile quasi weggestanzt werden. Die erste erzeugte Region enthält anschließend die kombinierte Region aus Loch und runder Region als Ergebnis der API `CombineRegion`.

Wenn Sie nun über die gewünschte Zielregion verfügen, wird mit `SetWindowRgn` die normalerweise rechteckige Region des Fensters überschrieben. Die Zielregion ist bewusst so dimensioniert, dass die UserForm anschließend keine Titelleiste mehr besitzt, die aber zum Verschieben der Form wichtig ist. Es sieht einfach nicht sehr schön aus, wenn nur ein Teil der Titelleiste zu sehen ist.

Es ist aber nicht so, dass diese Leiste nach dem Überschreiben der Region generell weg ist. Ich habe die runde Region aber so geformt, dass sie die Titelleiste nicht mit einschließt. Das Verschieben der Form klappt mit einem Trick aber dennoch:

GetMyHwnd

In dieser Funktion wird das Fensterhandle über den Fenstertext gesucht; das ist der Text, der in der Titelleiste des Fensters steht. Mit VBA kann dieser Text über die Eigenschaft `Caption` der UserForm gesetzt oder ausgelesen werden. Zum Ermitteln der Fensternummer lese und speichere ich erst einmal den ursprünglichen Fenstertext in einer Variablen. Dann setze ich als Caption wahllos ein paar Worte ein, die aber sonst nicht als Fenstertext vorkommen sollten, und suche über diesen String das Fensterhandle. Danach wird die Caption auf den ursprünglichen Fenstertext zurückgesetzt.

Bei der Suche selbst kommt die API-Funktion `FindWindow` zum Einsatz. Der Parameter, der normalerweise den Klassennamen aufnimmt, wird auf `vbNullString` gesetzt. Wenn ein leerer String dafür eingesetzt wird, sucht die Funktion ein Fenster mit einem leeren String als Klassennamen, nur bei der Übergabe von `vbNullString` wird dieser Parameter ignoriert.

UserForm_MouseDown

Bei einem Klick mit der linken Maustaste wird mittels `ReleaseCapture` die Maus freigegeben und es wird mit `SendMessage` ein Drücken der linken Maustaste auf die (immer noch unsichtbar vorhandene) Titelleiste simuliert. Mit diesem Kniff ist es möglich, die UserForm trotz ausgeblendeter Titelleiste zu verschieben.

Bei einem Rechtsklick wird die Form entladen.

UserForm_Terminate

In dieser Prozedur, die abgearbeitet wird, wenn die UserForm entladen wird, werden die Objekte umweltgerecht entsorgt und die Bits und Bytes kostengünstig recycelt.

UserForm_Activate

In dieser Prozedur wird die UserForm `modeless` gemacht, wenn es sich bei der aktuellen Officeversion um Office 97 handelt. Dazu wird die bedingte Kompilierung eingesetzt, wobei der Code in dieser Ereignisprozedur nur kompiliert und somit ausgeführt wird, wenn die Konstante VBA6 nicht wahr ist.

Handelt es sich um Office 97, wird zuerst einmal die Fensterzugriffsnummer des Excel-Hauptfensters ermittelt. Ist das Fensterhandle von Excel gefunden, was unter Excel 97 ein Fenster mit dem Klassennamen `XLMAIN` ist, werden mit der Funktion `EnableWindow` die Maus- und Keyboardeingaben darauf gelenkt. Dazu wird beim Aufruf der Funktion `EnableWindow` im Parameter `hwnd` die ermittelte Fensterzugriffsnummer übergeben und der Parameter `bEnable` dieser Funktion wird auf `True` gesetzt. Damit verliert die UserForm sofort ihren modalen Charakter.

Das ist zwar nur eine Notlösung und hat möglicherweise ein paar unerwünschte Seiteneffekte, ich selber setze aber den Code seit Jahren ohne Probleme ein und es ist mir diesbezüglich noch nichts Negatives aufgefallen. Wird die UserForm erst einmal modeless angezeigt, werden auch Ereignisprozeduren des Tabellenblattes ganz normal ausgelöst und wie gewohnt abgearbeitet. Die aufrufende Prozedur wird ungeachtet dessen erst dann fortgesetzt, wenn die Form entladen oder versteckt wird.

Eine kleine Unannehmlichkeit im Zusammenhang mit den Menüs darf dabei aber nicht verschwiegen werden:

Um mit den Menüs auch weiterhin arbeiten zu können, müssen Sie, nachdem die Form modeless angezeigt wird, einmal einen Doppelklick auf irgendein Menü oder einen Menübutton machen. Dadurch geht der Dialog ANPASSEN auf und wird dieser ohne Änderung weggeklickt, können Sie auch die Menüs wie gewohnt weiterbenutzen.

13

Fremde Anwendungen

13.1 Was Sie in diesem Kapitel erwartet

Das Zusammenspiel verschiedener Anwendungen ist heutzutage ein sehr wichtiges Thema. Dieses Kapitel zeigt in ein paar Beispielen, wie Sie mit OLE fremde, automatisierungsfähige Programme quasi fernsteuern können. Dabei befasst sich ein Beispiel mit dem Versenden einer E-Mail mittels Outlook und ein anderes mit dem Starten einer PowerPoint-Präsentation.

Aber auch Programme ohne eine solche Schnittstelle können ausgeführt und dabei die von diesen Programmen unterstützten Startparameter mit übergeben werden. Ein Beispiel zeigt, wie Sie ein fremdes Programm mit der `Shell`-Funktion starten und mit der weiteren Programmausführung erst dann fortfahren, wenn das fremde Programm beendet wurde.

Wenn das Programm, dass mit einer Datei verknüpft ist, vorher nicht bekannt ist, kann die API-Funktion `ShellExecute` eingesetzt werden. Das kann beispielsweise der Fall sein, wenn ein HTML-Dokument geöffnet werden soll und auf den verschiedenen Rechnern jeweils andere Browser zur Darstellung dieser Hypertextseiten installiert sind. `ShellExecute` respektiert die Einstellungen des Benutzers und öffnet die Dateien mit dem in der Registry für diese Dateierweiterung eingetragenen Programm.

Beim Einsatz der `Shell`-Funktion wird auch manchmal das mit der Dateierweiterung verknüpfte Programm benötigt. In dem Beispiel zum Starten einer Präsentation mit der `Shell`-Funktion wird Code vorgestellt, mit dem Sie das zugehörige Programm ermitteln können.

Ein weiteres Beispiel zeigt, wie Sie fremde Prozesse, die möglicherweise durch Fehler bei der Automatisierung als Leichen übrig geblieben sind, rigoros abschießen und aus dem Speicher entfernen können.

13.2 Allgemeines

13.2.1 OLE

OLE-Automatisierung (Object Linking and Embedding) ist eine der Möglichkeiten, Zugriff auf fremde Anwendungen zu erlangen. Dazu macht man sich eine Technik von Microsoft zunutze, die entwickelt wurde, um DDE abzulösen.

DDE (Dynamic Data Exchange), der dynamische Datenaustausch, wurde und wird zum Teil noch für Übertragung und Aktualisierung von Informationen zwischen verschiedenen Anwendungen benutzt. Dazu stellt ein Serverprogramm Daten bereit, die in eine Clientanwendung als Verknüpfung eingefügt werden. Der Server ist dabei das Programm, das Dienste zur Verfügung stellt, der Client nimmt diese Dienste dann in Anspruch.

Die DDE-Verknüpfung bietet den Vorteil, dass Daten, die in der Server-Anwendung verändert wurden, auch in der Clientanwendung aktualisiert dargestellt werden. Das kann sofort geschehen, wenn beide Anwendungen gleichzeitig laufen, beim erneuten Öffnen der Clientdatei oder wenn in der Client-Anwendung explizit der Befehl zum Aktualisieren gegeben wird.

Leider gibt es bei DDE ein paar Schwächen. Um beispielsweise Änderungen an den Daten vorzunehmen, muss die Server-Anwendung erst gestartet oder zu ihr gewechselt werden. Auch werden nicht alle Formatierungen der Originaldaten mit übernommen.

Um diese und noch ein paar andere Schwächen zu beseitigen, wurde OLE eingeführt. Eingefügte Daten im Client-Dokument werden als eigenständige Objekte betrachtet, wobei diese OLE-Objekte den Namen des OLE-Servers und alle anderen notwendigen Informationen intern speichern. Die Objekte werden in der Client-Anwendung geändert, obwohl im Hintergrund der Server aktiv wird.

Die Objekte können verknüpft und eingebettet werden. Bei einer Verknüpfung befinden sich die Daten in einer anderen Datei unter der Kontrolle der Server-Anwendung. Bei einer Änderung der Daten nimmt die Client-Anwendung eine automatische Aktualisierung vor.

Bei einem eingebetteten Objekt wird keine Verknüpfung zwischen Server und Client eingerichtet. Das Client-Objekt enthält die Daten und alle Informationen über die Server-Anwendung, wie den Namen des Servers oder die Dateistruktur. Zur Änderung wird dann die Server-Anwendung herangezogen.

Das Wichtigste für einen VBA-Benutzer ist aber die Automatisierung. Durch die Automatisierung legt der OLE-Server Schnittstellen offen, durch die andere Anwendungen auf Objekte des Servers zugreifen können. Diese offen gelegten Objekte können zum Beispiel bei Excel Arbeitsmappen und bei Word Dokumente sein. Damit ist es möglich, dass eine VBA-Prozedur Objekte anderer OLE-Anwendungen verwenden kann.

Um solche Objekte zu benutzen, gibt es zwei Möglichkeiten. Sie setzen in der VBE einen Verweis auf die andere OLE-Anwendung und legen sich mit

```
Dim XYZ As New Anwendung.Object
```

oder

```
Dim XYZ As Anwendung.Object
Set XYZ = New Anwendung.Object
```

das gewünschte Objekt an. Dabei hat man es mit früher Bindung (early binding) zu tun.

Das hat den großen Vorteil, dass die Client-Anwendung die Typen, Eigenschaften und Methoden des Objektes kennt und der Programmablauf schneller ist. Außerdem können Sie den Objektkatalog benutzen, um Informationen über das Objekt zu erlangen.

Mit der `CreateObject`- oder `GetObject`-Funktion können Sie das Gleiche erreichen, aber die Typenbibliothek steht in diesem Fall nicht zur Verfügung und die Client-Anwendung kennt keine Typen und Konstanten des OLE-Servers. Das nennt man späte Bindung (late binding).

Mit `CreateObject` wird eine neue Instanz der Anwendung angelegt und einer Objektvariablen zugewiesen, mit `GetObject` wird die Objektvariable mit einer schon laufenden Instanz angelegt.

Natürlich müssen Sie die Eigenschaften und Methoden des Objektes kennen, die Sie benutzen wollen. Am besten probieren Sie alles in der Originalanwendung aus und passen dann den funktionierenden Code für die Verwendung in der Client-Anwendung an.

Um erzeugte Prozesse aus dem Speicher zu entfernen, schließen Sie die fremde Anwendung mit der `Quit`-Methode des `Application`-Objektes. Die erzeugten Dokumente sollten vorher mit `Close` geschlossen werden.

13.2.2 Shell

Um einmal schnell eine fremde Anwendung zu starten, die keine Automatisierungsschnittstelle bietet, wie zum Beispiel Notepad, kommt die Funktion `Shell` zum Einsatz. Wie diese Funktion benutzt wird, können Sie der Hilfe entnehmen.

Das besondere an dieser Funktion ist aber, dass sie die `TaskID` des gestarteten Programms zurückgibt. Mit dieser kann der gewöhnliche VBA-Programmierer nicht viel anfangen, aber damit kann beispielsweise eine Abfrage gestartet werden, ob das gestartete Programm noch läuft. Dazu erfahren Sie später mehr.

13.2.3 ShellExecute

Diese API-Funktion öffnet oder druckt eine angegebene Datei. Dabei wird das Programm benutzt, das für die entsprechende Dateierweiterung auf dem System registriert wird. Es können die gleichen Parameter mit übergeben werden, die auch an der Befehlszeile möglich sind.

Nachfolgend die Deklarationsanweisung:

```
Private Declare Function ShellExecute _
   Lib "shell32.dll" Alias "ShellExecuteA" ( _
   ByVal Fensterzugriffsnummer As Long, _
   ByVal lpOperation_wie_Open_oder_Print_oder_explore As String, _
   ByVal lpDateiname_incl_Pfad As String, _
   ByVal lpZusätzliche_Startparameter As String, _
   ByVal lpArbeitsverzeichnis As String, _
   ByVal nGewünschte_Fenstergröße_der_Anwendung As Long _
   ) As Long
```

Als Ergebnis liefert diese Funktion einen Longwert zurück, der größer 32 ist, wenn die Funktion erfolgreich war.

Ansonsten wird einer der folgenden Werte zurückgeliefert:

- `Private Const ERROR_FILE_NOT_FOUND = 2&`

  Die angegebene Datei wurde nicht gefunden.

- `Private Const ERROR_PATH_NOT_FOUND = 3&`

  Der angegebene Pfad wurde nicht gefunden.

- `Private Const ERROR_BAD_FORMAT = 11&`

  Die *.exe*-Datei hat das falsche Format oder lässt sich unter diesem Betriebssystem nicht starten.

- `Private Const SE_ERR_ACCESSDENIED = 5`

  Der Zugriff auf die Datei wurde verweigert.

- `Private Const SE_ERR_ASSOCINCOMPLETE = 27`
- `Private Const SE_ERR_DDEBUSY = 30`
- `Private Const SE_ERR_DDEFAIL = 29`

  Die DDE-Transaktion ist fehlgeschlagen.

- `Private Const SE_ERR_DDETIMEOUT = 28`

  Die DDE-Transaktion konnte nicht abgeschlossen werden, da ein Timeout aufgetreten ist.

- `Private Const SE_ERR_DLLNOTFOUND = 32`

  Die *.dll* wurde nicht gefunden.

- `Private Const SE_ERR_NOASSOC = 31`

  Mit der Dateinamenserweiterung ist keine Anwendung verknüpft.

- `Private Const SE_ERR_OOM = 8`

  Es gibt zu wenig Speicher, um die Operation auszuführen.

- `Private Const SE_ERR_SHARE = 26`

Fensterzugriffsnummer

Sie ist das Handle zum Besitzerfenster und kann im Allgemeinen auch Null sein.

lpOperation_wie_Open_oder_Print_oder_explore

Dieser Parameter gibt die Operation an, die ausgeführt werden soll. Ist der Parameter Null, wird `Open` angenommen. Was für ein String möglich ist, ergibt sich aus den Einträgen in der Registry für den jeweiligen Datentyp.

Folgende Strings werden häufig benutzt:

- `Open` öffnet die Datei mit der verknüpften Anwendung.
- `Print` druckt ein Dokument mit der verknüpften Anwendung. Bei einer ausführbaren Datei wird diese gestartet.
- `Explore` startet den Explorer und markiert den übergebenen Pfad.
- `Show` startet beispielsweise eine PowerPoint-Präsentation.

lpDateiname_incl_Pfad

Der Pfad inklusive Dateiname. Bei Übergabe von `Explore` im zweiten Parameter wird ein Pfad übergeben.

lpZusätzliche_Startparameter

Bei einer ausführbaren Datei können hier zusätzliche Startparameter als String übergeben werden. Wenn Sie mit einem Dokument arbeiten, wird Null übergeben.

lpArbeitsverzeichnis

Bei einer ausführbaren Datei wird das Arbeitsverzeichnis als String übergeben.

nGewünschte_Fenstergröße_der_Anwendung

Bei einer ausführbaren Datei wird mit diesem Parameter gesteuert, wie die Anwendung angezeigt wird.

- `Private Const SW_HIDE = 0`

  Versteckt das Fenster der Anwendung , ein anderes wird aktiviert.
- `Private Const SW_MAXIMIZE = 3`

  Maximiert das Fenster der Anwendung.
- `Private Const SW_MINIMIZE = 6`

  Minimiert das Fenster der Anwendung.
- `Private Const SW_RESTORE = 9`

  Aktiviert und zeigt das Fenster der Anwendung an. Es werden dabei die Originalwerte des Fensters benutzt.
- `Private Const SW_SHOW = 5`

  Aktiviert das Fenster der Anwendung und zeigt es in der aktuellen Größe und Position an.
- `Private Const SW_SHOWDEFAULT = 10`

  Stellt die Werte des Fensters wieder her, mit der die Applikation gestartet wurde.

- `Private Const SW_SHOWMAXIMIZED = 3`

  Aktiviert das Fenster und zeigt es maximiert an.

- `Private Const SW_SHOWMINIMIZED = 2`

  Aktiviert das Fenster und zeigt es minimiert an.

- `Private Const SW_SHOWMINNOACTIVE = 7`

  Zeigt das Fenster minimiert an, aber aktiviert es nicht.

- `Private Const SW_SHOWNORMAL = 1`

  Aktiviert und zeigt das Fenster an. Wenn es maximiert oder minimiert ist, wird die normale Größe wiederhergestellt.

- `Private Const SW_SHOWNA = 8`

  Zeigt das Fenster in der aktuellen Position und Größe an. Das aktive Fenster behält die Aktivität.

- `Private Const SW_SHOWNOACTIVATE = 4`

  Das Fenster wird angezeigt, aber nicht aktiviert.

13.3 E-Mail mit VBA, Shell und ShellExecute versenden

Um Arbeitsmappen zu versenden, haben Sie die Möglichkeit, die Methode `SendMail` zu benutzen. Mehr zu dieser Methode können Sie in der Hilfe zu Excel-VBA nachlesen.

```
ActiveWorkbook.SendMail recipients:="Empfaenger@Host.de", _
   Subject:="Test"
```

Diese Methode reicht in den meisten Fällen aus. Wenn aber nur eine einfache Textnachricht verschickt werden soll, ist das aber schon zu viel, denn mit der Methode `SendMail` wird ja eine komplette Datei verschickt.

Eine weitere Möglichkeit, eine Mail zu verschicken, ist die, mit dem `Shell`-Befehl das Mailprogramm zu starten und dabei die entsprechenden Startparameter mit zu übergeben. Leider sind die Übergabeparameter nicht einheitlich und abhängig vom verwendeten E-Mail-Client. Außerdem muss das verwendete Mailprogramm und der Pfad dorthin bekannt sein.

Etwas besser ist da schon der Einsatz der API-Funktion `ShellExecute`. Diese benutzt zum Ausführen von Dateien das damit verknüpfte Programm. Was bei Dateien hervorragend funktioniert, bereitet beim Senden von E-Mails ähnliche Schwierigkeiten wie beim Benutzen der `Shell`-Funktion. Sie müssen dabei zwar nicht mehr den Pfad zum E-Mail-Client kennen, aber die Übergabeparameter müssen bekannt sein. Auch das Senden von Dateianhängen funktioniert meines Wissens damit nicht.

Folgender Beispielcode (Listing 13.1) funktioniert zumindest mit Outlook und Outlook Express:

Listing 13.1
Beispiele\13_UserForms\
13_01_FremdeAnwendungen.xls\
mdlShellExecute

```
Private Declare Function ShellExecute _
   Lib "shell32.dll" Alias "ShellExecuteA" ( _
   ByVal Fensterzugriffsnummer As Long, _
   ByVal lpOperation_wie_Open_oder_Print As String, _
   ByVal lpDateiname_incl_Pfad As String, _
   ByVal lpZusätzliche_Startparameter As String, _
   ByVal lpArbeitsverzeichnis As String, _
   ByVal nGewünschte_Fenstergröße_der_Anwendung As Long _
   ) As Long

Private Const SW_HIDE = 0
Private Const SW_MAX = 10
Private Const SW_MAXIMIZE = 3
Private Const SW_MINIMIZE = 6
Private Const SW_NORMAL = 1
Private Const SW_SHOW = 5
Private Const SW_SHOWDEFAULT = 10
Private Const SW_SHOWMAXIMIZED = 3
Private Const SW_SHOWMINIMIZED = 2
Private Const SW_SHOWMINNOACTIVE = 7
Private Const SW_SHOWNORMAL = 1
Private Const SW_SHOWNOACTIVATE = 4

Sub test()

   Dim strDestAddress  As String
   Dim strSubject      As String
   Dim strBody         As String

   strDestAddress = "mailto:Empfänger@Host.de?"
   strSubject = "Subject= Subject"
   strBody = "&body= Das ist der Body"

   strDestAddress = strDestAddress & strSubject & strBody

   ShellExecute 0, "open", strDestAddress, _
      vbNullString, vbNullString, SW_SHOWNORMAL

End Sub
```

13.4 E-Mail mit Outlook versenden

Outlook ist ein fester Bestandteil des Office-Paketes und dürfte somit fast jedem Benutzer von Office zur Verfügung stehen. Der große Vorteil dieses Programms gegenüber dem Windows-Standardmailprogramm Outlook Express besteht in der Möglichkeit, es über die OLE-Automation fernzusteuern.

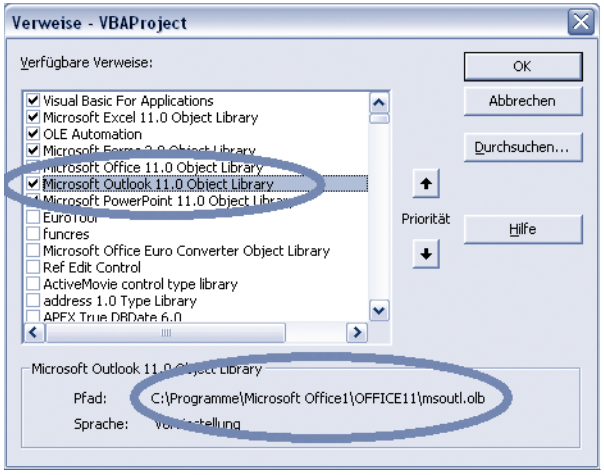
Und dies ist zugleich auch ein Nachteil. So ist es sogar aus einem VBScript möglich, dieses Programm zu benutzen, also Mails zu versenden und auch sonst auf alle gespeicherten Informationen wie zum Beispiel das Adressbuch

zuzugreifen. Das haben sich viele Virenprogrammierer zunutze gemacht und dieses an sich positive Feature ordentlich missbraucht.

Microsoft hat aber darauf reagiert und lässt das automatische Versenden aus anderen Anwendungen nicht mehr ohne weiteres zu. Man muss nun immer wieder auf einen Dialog reagieren und das Versenden explizit zulassen. Wenn Sie dies umgehen wollen, müssen Sie fremde Software einsetzen.

Im nachfolgenden Beispiel (Listing 13.2) finden Sie zwei Prozeduren, welche die Funktionalität von Outlook benutzen. Der Unterschied zwischen beiden ist, dass in der ersten (`SendMessage`) ein Verweis auf Outlook (Abbildung 13.1) benötigt wird, während in der zweiten lediglich Outlook auf dem Rechner korrekt installiert sein muss.

Abbildung 13.1
Verweis auf Outlook



Listing 13.2
Beispiele\13_UserForms\13_01_FremdeAnwendungen.xls\mdlMail

```
Private Sub testMail()
SendMessage "Empfänger@host.de", "Subject", "Body", _
   ActiveWorkbook.FullName, "CC@Host.de", "BCC@Host.de"
End Sub

Public Sub SendMessage( _
   strRecipient As String, _
   strSubject As String, _
   strBody As String, _
   Optional strAtt As String, _
   Optional strCC As String, _
   Optional strBCC As String)


   ' Mit einem Verweis
   Dim objOutlApp As Outlook.Application
   Dim objOutlNam As Outlook.Namespace
   Dim objOutlMsg As Outlook.MailItem
   Dim objOutlRec As Outlook.Recipient

   ' Outlook Session.
   Set objOutlApp = New Outlook.Application
```

Listing 13.2 (Forts.)
Beispiele\13_UserForms\
13_01_FremdeAnwendungen.xls\
mdlMail

```
    ' Namespace erzeugen
    Set objOutlNam = objOutlApp.GetNamespace("MAPI")

    ' Mail erzeugen
    Set objOutlMsg = objOutlApp.CreateItem(olMailItem)

    With objOutlMsg

        ' Subjekt hinzufügen
        .Subject = strSubject

        ' Body hinzufügen
        .Body = strBody

        ' Empfänger hinzufügen
        Set objOutlRec = .Recipients.Add(strRecipient)
        objOutlRec.Type = olTo

        If strBCC <> "" Then
            ' BCC hinzufügen
            Set objOutlRec = .Recipients.Add(strBCC)
            objOutlRec.Type = olBCC
        End If

        If strCC <> "" Then
            ' CC hinzufügen
            Set objOutlRec = .Recipients.Add(strCC)
            objOutlRec.Type = olCC
        End If

        If Dir$(strAtt) <> "" Then
            ' Dateianlage hinzufügen
            .Attachments.Add strAtt
        End If

        .Send ' Versenden

    End With

    Set objOutlMsg = Nothing

End Sub

Public Sub SendMessage1( _
    strRecipient As String, _
    strSubject As String, _
    strBody As String, _
    Optional strAtt As String, _
    Optional strCC As String, _
    Optional strBCC As String)

    ' Ohne Verweis
    Dim objOutlApp As Object
    Dim objOutlNam As Object
    Dim objOutlMsg As Object
    Dim objOutlRec As Object
```

Listing 13.2 (Forts.)
Beispiele\13_UserForms\
13_01_FremdeAnwendungen.xls\
mdlMail

```
    Const olMailItem = 0
    Const olTo = 1
    Const olCC = 2
    Const olBCC = 3

    ' Outlook session.
    Set objOutlApp = CreateObject("Outlook.Application")

    ' Namespace erzeugen
    Set objOutlNam = objOutlApp.GetNamespace("MAPI")

    ' Mail erzeugen
    Set objOutlMsg = objOutlApp.CreateItem(olMailItem)

    With objOutlMsg

        ' Subjekt hinzufügen
        .Subject = strSubject

        ' Body hinzufügen
        .Body = strBody

        ' Empfänger hinzufügen
        Set objOutlRec = .Recipients.Add(strRecipient)
        objOutlRec.Type = olTo

        If strBCC <> "" Then
            ' BCC hinzufügen
            Set objOutlRec = .Recipients.Add(strBCC)
            objOutlRec.Type = olBCC
        End If

        If strCC <> "" Then
            ' CC hinzufügen
            Set objOutlRec = .Recipients.Add(strCC)
            objOutlRec.Type = olCC
        End If

        If Dir$(strAtt) <> "" Then
            ' Dateianlage hinzufügen
            .Attachments.Add strAtt
        End If

        .Send ' Versenden

    End With

    Set objOutlMsg = Nothing

End Sub
```

13.5 Anwendung starten und warten

Es gibt sicherlich Situationen, in denen man programmgesteuert eine andere Anwendung starten und erst nach deren Ende mit der Programmausführung fortfahren möchte. Außerdem wäre auch ein Timeout nicht schlecht, der die gestartete Anwendung nach einer gewissen Zeit beendet.

Leider kümmern sich `Shell` und `ShellExecute` nicht mehr um die gestartete Anwendung und die Programmausführung wird auch nicht unterbrochen. Wie schon erwähnt, liefert aber die `Shell`-Funktion die `TaskID` des gestarteten Programms zurück. Damit wird es möglich, nachzuschauen, ob die Anwendung noch läuft sowie den gestarteten Prozess zu einem beliebigen Zeitpunkt einfach abzumurksen.

Listing 13.3
Beispiele\13_UserForms\13_01_FremdeAnwendungen.xls\mdlStartAndWait

```
Private Const PROCESS_QUERY_INFORMATION = &H400
Private Const STILL_ACTIVE = &H103
Private Const PROCESS_ALL_ACCESS = &H1F0FFF

Private Declare Function GetExitCodeProcess _
   Lib "kernel32" ( _
   ByVal hProcess As Long, _
   lpExitCode As Long _
   ) As Long

Declare Sub Sleep _
   Lib "kernel32" ( _
   ByVal dwMilliseconds As Long)

Private Declare Function TerminateProcess _
   Lib "kernel32" ( _
   ByVal hProcess As Long, _
   ByVal uExitCode As Long _
   ) As Long

Private Declare Function OpenProcess _
   Lib "kernel32" ( _
   ByVal dwDesiredAccess As Long, _
   ByVal bInheritHandle As Long, _
   ByVal dwProcessId As Long _
   ) As Long

Private Declare Function CloseHandle _
   Lib "kernel32" ( _
   ByVal hObject As Long _
   ) As Long

Private Sub test()
   If StartAndWait("Notepad.exe", 10) Then
      MsgBox "Notepad normal beendet"
   Else
      MsgBox "Notepad abgeschossen"
   End If
End Sub
```

Listing 13.3 (Forts.)
Beispiele\13_UserForms\13_01_FremdeAnwendungen.xls\mdlStartAndWait

```
Public Function StartAndWait( _
   strProgram As String, _
   Optional lngdteTimeoutSec As Long = 300 _
   ) As Boolean

   Dim lngTaskID  As Long
   Dim lngProcess As Long
   Dim lngRuns    As Long
   Dim dteTimeout As Date

   ' Programm starten, TaskID zurückbekommen
   lngTaskID = Shell(strProgram, 1)

   ' Handle zum Prozess holen
   lngProcess = OpenProcess(PROCESS_ALL_ACCESS, _
                0&, lngTaskID)

   ' Timeoutzeit ermitteln
   dteTimeout = Now + TimeSerial(0, 0, lngdteTimeoutSec)

   StartAndWait = True

   Do
   ' So lange durchlaufen, wie der Prozess existiert

      ' die Variable lngRuns mit Prozessinfos füllen
      GetExitCodeProcess lngProcess, lngRuns

      If Now > dteTimeout Then
         ' Timeoutzeit erreicht

         If lngRuns = STILL_ACTIVE Then

            ' Prozess killen
            TerminateProcess lngProcess, 0&

            ' Timeout signalisieren
            StartAndWait = False

            ' Schleife verlassen
            Exit Do

         End If

      End If

      ' Schlafen legen, um die Prozessorauslastung
      ' zu verringern
      Sleep 100

   Loop While lngRuns = STILL_ACTIVE

' Handle schließen
CloseHandle lngProcess

End Function
```

Test

Die Prozedur `Test` ist, wie nicht anders zu erwarten, zum Testen der Prozedur `StartAndWait` gedacht.

StartAndWait

Das fremde Programm wird mit der `Shell`-Funktion gestartet, die eine `TaskID` zurückgibt. Mit der API-Funktion `OpenProcess` besorgen Sie sich dann ein Prozesshandle auf das gestartete Programm. In einer Schleife wird nun mit der API-Funktion `GetExitCodeProcess` der aktuelle Status des Prozesses abgefragt. Dieser wird über den Parameter `lpExitCode` geliefert.

Wenn der Prozess läuft, wird `STILL_ACTIVE` zurückgegeben und die Schleife noch einmal durchlaufen. Damit die Prozessorauslastung während des Schleifendurchlaufes reduziert wird, schicken Sie mit der API-Prozedur `Sleep` die eigene Anwendung für jeweils 100 Millisekunden in den Tiefschlaf.

Wird die Timeoutzeit überschritten, beenden Sie mit der API-Funktion `TerminateProcess` den gestarteten Prozess und die Schleife wird verlassen.

Zum Schluss wird das Prozesshandle geschlossen.

13.6 Fremde Programme abschießen

Wenn Sie häufig mit Automatisierung arbeiten, kann es immer mal wieder vorkommen, dass die gestartete Anwendung nicht sauber terminiert wird. In den meisten Fällen liegt das an der eigenen Schlampigkeit, weil noch irgendeine Referenz darauf existiert. Es soll aber auch vereinzelt vorkommen, dass man daran gänzlich unschuldig ist.

Mit der Zeit gibt es dann jedenfalls viele Prozessleichen, die Ressourcen fressen und den Rechner langsam machen. Über den Taskmanager können die Prozesse abgeschossen werden, es lässt sich aber auch programmgesteuert erledigen, wenn der Programmnamen bekannt ist.

Listing 13.4
Beispiele\13_UserForms\13_01_FremdeAnwendungen.xls\mdlKill

```
Private Const MAX_PATH = 260
Private Const TH32CS_SNAPPROCESS As Long = 2
Private Const PROCESS_ALL_ACCESS = &H1F0FFF

Private Type PROCESSENTRY32
    dwSize As Long
    cntUsage As Long
    th32ProcessID As Long
    th32DefaultHeapID As Long
    th32ModuleID As Long
    cntThreads As Long
    th32ParentProcessID  As Long
    pcPriClassBase As Long
    dwFlags As Long
    szExeFile As String * MAX_PATH
End Type
```

Listing 13.4 (Forts.)
Beispiele\13_UserForms\13_01_FremdeAnwendungen.xls\mdlKill

```
Private Declare Function CreateToolhelp32lngSnapshot _
    Lib "kernel32" ( _
    ByVal dwFlags As Long, _
    ByVal th32ProcessID As Long _
    ) As Long

Private Declare Function CloseHandle Lib "kernel32" ( _
    ByVal hObject As Long _
    ) As Long

Private Declare Function Process32First Lib "kernel32" ( _
    ByVal hlngSnapshot As Long, _
    ByRef lppe As PROCESSENTRY32 _
    ) As Long

Private Declare Function Process32Next Lib "kernel32" ( _
    ByVal hlngSnapshot As Long, _
    ByRef lppe As PROCESSENTRY32 _
    ) As Long

Private Declare Function OpenProcess Lib "kernel32" ( _
    ByVal dwDesiredAccess As Long, _
    ByVal bInheritHandle As Long, _
    ByVal dwProcessId As Long _
    ) As Long

Private Declare Function TerminateProcess Lib "kernel32" ( _
    ByVal lngProcess As Long, _
    ByVal uExitCode As Long _
    ) As Long

Sub test()
    ' Teilname der .exe
    MsgBox KillProcess("notepad") & vbCrLf & _
        "Prozesse abgeschossen", vbOKOnly, "notepad"
End Sub

Public Function KillProcess( _
    strExeName As String _
    ) As Long

   Dim lngSnapshot      As Long
   Dim udtProzessinfo   As PROCESSENTRY32
   Dim lngRet           As Long
   Dim lngProcess       As Long
   Dim strName          As String

   'Schnappschuss aller momentan laufenden Prozesse erzeugen
   lngSnapshot = CreateToolhelp32lngSnapshot(TH32CS_SNAPPROCESS, 0&)

   With udtProzessinfo

      .dwSize = Len(udtProzessinfo)

      'Infos über ersten Prozess
      lngRet = Process32First(lngSnapshot, udtProzessinfo)
```

Listing 13.4 (Forts.)
Beispiele\13_UserForms\13_01_FremdeAnwendungen.xls\mdlKill

```
      Do While lngRet
         ' Durchlaufen, solange lngRet <> null ist

           ' Beim ersten Auftreten von chr(0) kürzen
           strName = StringFromASCIIZ(.szExeFile)

           If InStr(1, LCase(strName), LCase(strExeName)) Then
               'Richtige ausführbare Datei gefunden

            'Prozesshandle holen
            lngProcess = OpenProcess(PROCESS_ALL_ACCESS, _
                0, .th32ProcessID)

            If lngProcess Then

               'Prozess schonungslos abschießen
               TerminateProcess lngProcess, 0&

               ' Anzahl abgeschossener Prozesse zurückgeben
               KillProcess = KillProcess + 1

               'Handle schließen
               CloseHandle lngProcess

            End If

         End If

         'Infos über nächsten Prozess
         lngRet = Process32Next(lngSnapshot, udtProzessinfo)

      Loop

   End With

'Handle schließen
CloseHandle lngSnapshot

End Function

Private Function StringFromASCIIZ(ASCIIZ As String) As String
   'ASCIIZ-String kürzen
   If InStr(1, ASCIIZ, Chr(0)) > 0 Then
      StringFromASCIIZ = Left$(ASCIIZ, InStr(1, ASCIIZ, Chr(0)) - 1)
   Else
      StringFromASCIIZ = ASCIIZ
   End If
End Function
```

Test

Die Prozedur `Test` ist zum Testen der Funktion `KillProcess` gedacht.

KillProcess

Mit der Funktion `CreateToolhelp32Snapshot` wird eine Momentaufnahme aller zu diesem Zeitpunkt laufenden Prozesse mitsamt den zugehörigen `Heaps`, `Modulen` und `Threads` erstellt und ein Handle darauf zurückgeliefert.

Die API-Funktion `Process32First` liefert den ersten Prozess dieser durch das Handle `lngSnapshot` repräsentierten Prozessliste. Mit der API-Funktion `Process32Next` kommen Sie dann an den nächsten Prozess der Liste. Sie durchlaufen nun in einer Schleife die gesamte Liste.

Das Element `szExeFile` der Struktur `PROCESSENTRY32` enthält einen nullterminierten String mit dem Namen der ausführbaren Datei. Enthält dieser Name den übergebenen String `strExeName`, holen Sie sich mit der Funktion `OpenProcess` ein Handle mit den entsprechenden Rechten auf den Prozess. Wie bei vielen Handles, die von API-Funktionen geliefert werden müssen Sie sicherstellen, dass das Handle auch wieder mit `CloseHandle` geschlossen wird.

Mit diesem Handle können Sie den Prozess abschießen, dafür wird `TerminateProcess` benutzt. Das Funktionsergebnis von `KillProcess` soll die Anzahl der terminierten Prozesse anzeigen, also wird jedes Mal nach dem Aufruf `TerminateProcess` eins zum Funktionsergebnis hinzugezählt.

StringFromASCIIZ

Diese Funktion dient lediglich dazu, den nullterminierten String beim ersten Auftreten von `chr(0)` zu kürzen.

13.7 Präsentation starten

Eine PowerPoint-Präsentation lässt sich auf verschiedene Weisen starten. Einmal mit `Shell`, einfacher mit `ShellExecute` und unter voller Kontrolle mit der OLE-Automation.

13.7.1 Shell

Wenn Sie warten wollen, bis eine gestartete Anwendung beendet ist, benötigen Sie die `TaskID`, mit der Sie sich ein Handle auf den Prozess holen können. Mit diesem Prozesshandle können Sie sich schließlich die Information darüber holen, ob der Prozess noch läuft. Die benötigte `TaskID` wird leider nur von der `Shell`-Funktion geliefert.

Ein Problem mit der `Shell`-Funktion ist, dass Sie die Anwendung kennen müssen, mit der die Datei geöffnet werden kann. Außerdem benötigen Sie den zugehörigen Programmpfad, wenn sich die ausführbare Datei nicht gerade in einem der Hauptsuchpfade befindet. Das kann aber auf jedem Rechner anders sein, weshalb es in vielen Fällen zu Fehlern kommt.

Um an die benötigten Informationen zu kommen, können Sie die API-Funktion `FindExecutable` einsetzen, um das Programm und den Speicherort zu finden, mit der eine Datei geöffnet werden kann.

Nachfolgend ein Beispiel (Listing 13.5):

Listing 13.5
Beispiele\13_UserForms\
13_01_FremdeAnwendungen.xls\
mdlPPTShell

```
Private Declare Function FindExecutable _
   Lib "shell32.dll" Alias "FindExecutableA" ( _
   ByVal lpFile As String, _
   ByVal lpDirectory As String, _
   ByVal lpResult As String _
   ) As Long

Private Sub TestXLShell()
   Dim strPath As String

   strPath = Application.GetOpenFilename( _
      "Power Point(*.pot),*.pot", , _
      "Präsentation Neu")

   If Dir(strPath) <> "" Then

      ' Leere Präsentation
      XLShell strPath, "/n"

   End If

   strPath = Application.GetOpenFilename( _
      "Power Point(*.ppt),*.ppt", , _
      "Präsentation Starten")

   If Dir(strPath) <> "" Then

      ' Starten
      XLShell strPath, "/s"

   End If

   strPath = Application.GetOpenFilename( _
      "Power Point(*.ppt),*.ppt", , _
      "Präsentation Öffnen")

   If Dir(strPath) <> "" Then

      ' Öffnen
      XLShell strPath

   End If

   strPath = Application.GetOpenFilename( _
      "Power Point(*.ppt),*.ppt", , _
      "Präsentation Drucken")

   If Dir(strPath) <> "" Then

      ' Drucken
      XLShell strPath, "/p"

   End If

End Sub
```

Listing 13.5 (Forts.)
Beispiele\13_UserForms\13_01_FremdeAnwendungen.xls\mdlPPTShell

```
Public Function XLShell( _
   ByVal strFile As String, _
   Optional ByVal strParam As String _
   ) As Long

   Dim strProgram As String
   Dim strResult As String

   ' Puffer anlegen
   strProgram = String(255, 0)

   ' Verknüpftes Programm für Datei finden
   FindExecutable strFile, vbNullString, strProgram

   ' Beim ersten chr(0) abschneiden
   strProgram = Left(strProgram, InStr(1, strProgram, Chr(0)) - 1)

   ' Wenn kein Programm verknüpft ist, abbrechen
   If strProgram = "" Then Exit Function

   ' Den Befehlsstring zusammensetzen
   strFile = """" & strProgram & """  " & strParam & _
      " """ & strFile & """

   ' Ausführen und TaskId zurückgeben
   XLShell = Shell(strFile, vbNormalFocus)

End Function
```

TestXLShell

Zuerst wird ein Dialog zur Dateiauswahl gestartet, wobei der Filter auf Dateien mit der Endung *.pot*, also auf Vorlagendateien, gesetzt wird. Wird eine Datei gewählt, wird die Funktion `XLShell` aufgerufen. Dabei werden der Dateipfad und der Parameter `/n` für eine neue Datei übergeben.

Anschließend holen Sie sich den Namen und den Pfad zu einer PowerPoint-Datei mit der Endung *.ppt*. Dann wird die Funktion `XLShell` aufgerufen, der Dateipfad und der Parameter `/s` zum Zeigen der Präsentation übergeben.

Wird die Funktion `XLShell` nur mit dem Pfad gestartet, wird die übergebene Präsentation im Entwurfsmodus geöffnet.

Wird an `XLShell` der Parameter `/p` mit übergeben, wird die Präsentation gedruckt.

XLShell

Diese Hüllfunktion dient dazu, die `Shell`-Funktion etwas komfortabler zu machen.

An die API-Funktion `FindExecutable` wird der Dateipfad und ein Stringpuffer mit dem Namen `strProgram` übergeben. Dieser Stringpuffer enthält nach der Rückkehr der Funktion den Speicherort und den Programmnamen, mit der eine Datei geöffnet werden kann.

An den Pfad zur Programmdatei wird anschließend der Befehlszeilenparameter und der Pfad zu der zu öffnenden Datei gehängt. Dieser String wird an die `Shell`-Funktion übergeben, die `TaskID`, die Shell zurückliefert, wird als Funktionsergebnis von `XLShell` zurückgegeben.

Diese Funktion ist übrigens nicht auf PowerPoint beschränkt. Wenn Sie die nötigen Befehlszeilenparameter der Anwendungen kennen und mit übergeben, können Sie beliebige Dateien beispielsweise öffnen oder drucken, ohne den Anwendungspfad zu kennen.

13.7.2 ShellExecute

Mit dieser API-Funktion muss das mit der Datei verknüpfte Programm nicht mehr bekannt sein. Wie das Beispiel (Listing 13.6) zeigt, wird das Programm benutzt, das für die entsprechende Dateierweiterung auf dem System registriert wird.

Dabei wird lediglich der Dateipfad, der Befehl in Form eines Strings wie beispielsweise `open` und die gewünschte Fenstergröße übergeben.

Listing 13.6
Beispiele\13_UserForms\13_01_FremdeAnwendungen.xls\mdlPPTSHExecute

```
Private Declare Function ShellExecute Lib "shell32.dll" _
   Alias "ShellExecuteA" (ByVal Fensterzugriffsnummer As Long, _
   ByVal lpOperation_wie_Open_oder_Print As String, _
   ByVal lpDateiname_incl_Pfad As String, _
   ByVal lpZusätzliche_Startparameter As String, _
   ByVal lpArbeitsverzeichnis As String, _
   ByVal nGewünschte_Fenstergröße_der_Anwendung As Long) _
   As Long

Private Const SW_HIDE = 0
Private Const SW_MAX = 10
Private Const SW_MAXIMIZE = 3
Private Const SW_MINIMIZE = 6
Private Const SW_NORMAL = 1
Private Const SW_SHOW = 5
Private Const SW_SHOWDEFAULT = 10
Private Const SW_SHOWMAXIMIZED = 3
Private Const SW_SHOWMINIMIZED = 2
Private Const SW_SHOWMINNOACTIVE = 7
Private Const SW_SHOWNORMAL = 1
Private Const SW_SHOWNOACTIVATE = 4


Private Sub PPTPrintOut()
   Dim strPath As String

   strPath = Application.GetOpenFilename( _
      "Power Point(*.ppt),*.ppt", , "Drucken")
   If Dir(strPath) = "" Then Exit Sub

   ShellExecute 0&, "Print", , strPath, _
      vbNullString, vbNullString, SW_SHOWMINNOACTIVE

End Sub
```

Listing 13.6 (Forts.)
Beispiele\13_UserForms\
13_01_FremdeAnwendungen.xls\
mdlPPTSHExecute

```
Private Sub PPTStart()
   Dim strPath As String

   strPath = Application.GetOpenFilename( _
      "Power Point(*.ppt),*.ppt", , "Starten")
   If Dir(strPath) = "" Then Exit Sub

   ShellExecute 0&, "Show", , strPath, _
      vbNullString, vbNullString, SW_SHOWNORMAL

End Sub

Private Sub PPTOpen()
   Dim strPath As String

   strPath = Application.GetOpenFilename( _
      "Power Point(*.ppt),*.ppt")
   If Dir(strPath) = "" Then Exit Sub

   ShellExecute 0&, "Open", strPath, _
      vbNullString, vbNullString, SW_SHOWNORMAL

End Sub

Private Sub PPTNew()
   Dim strPath As String

   strPath = Application.GetOpenFilename( _
      "Power Point( *.pot), *.pot", , "Neu")
   If Dir(strPath) = "" Then Exit Sub

   ShellExecute 0&, "Open", strPath, _
      vbNullString, vbNullString, SW_SHOWNORMAL

End Sub
```

13.7.3 OLE

Mit der OLE-Automation haben Sie die volle Kontrolle über das fernzusteuernde Programm. Dabei müssen Sie aber die Konzepte, Eigenschaften, Methoden und Konstanten des zu steuernden Programms gut kennen.

In dem vorliegenden Beispiel bin ich so vorgegangen, dass ich die Aktionen, die durchgeführt werden sollen, in der Zielanwendung als Makro aufgezeichnet, entsprechend gekürzt und auf meine Bedürfnisse angepasst habe. Übrigens sollte sich niemand schämen, wenn er den Makrorekorder benutzt. Das Auffinden der benötigten Eigenschaften und Methoden ist nicht immer einfach und auch die Hilfe ist manchmal erst eine Hilfe, wenn man weiß, wonach man suchen muss.

In dem folgenden Beispiel (Listing 13.7) muss ein Verweis (Abbildung 13.2) auf die PowerPoint-Bibliothek gesetzt werden.

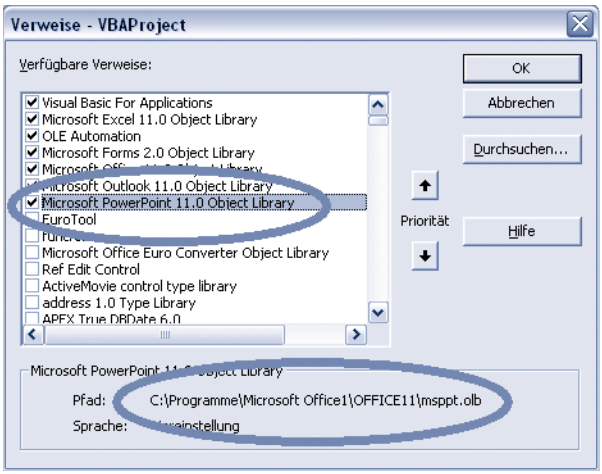


Abbildung 13.2
Verweis auf PowerPoint

Listing 13.7
Beispiele\13_UserForms\
13_01_FremdeAnwendungen.xls\
mdlPPTOle

```
Private Sub testOle()
   Dim strPath As String

   strPath = Application.GetOpenFilename( _
      "Power Point(*.ppt),*.ppt", , _
      "Präsentation Starten")

   If Dir(strPath) <> "" Then

      ' Starten
      PptOLENewShowPrint strPath

   End If

   strPath = Application.GetOpenFilename( _
      "Power Point(*.ppt),*.ppt", , _
      "Präsentation öffnen")

   If Dir(strPath) <> "" Then

      ' Öffnen
      PptOLENewShowPrint strPath, , True

   End If

   strPath = Application.GetOpenFilename( _
      "Power Point(*.ppt),*.ppt", , _
      "Präsentation Drucken")

   If Dir(strPath) <> "" Then

      ' Drucken
       PptOLENewShowPrint strPath, True

   End If
```

Listing 13.7 (Forts.)
Beispiele\13_UserForms\13_01_FremdeAnwendungen.xls\mdlPPTOle

```
    ' Präsentation Neu
    PptOLENewShowPrint
End Sub

Public Sub PptOLENewShowPrint( _
   Optional strPPTPath As String, _
   Optional blnPrint As Boolean, _
   Optional blnOpen As Boolean)

   Dim objPptApp     As PowerPoint.Application
   Dim objPptPresent As PowerPoint.Presentation

   ' Objekterstellung
   Set objPptApp = New PowerPoint.Application

   ' Aktivieren
   objPptApp.Activate

   If strPPTPath = "" Then
      ' Neu anlegen

      ' Sichtbar machen
      objPptApp.Visible = msoTrue

      ' Präsentation hinzufügen
      Set objPptPresent = objPptApp.Presentations.Add(True)

      With objPptPresent

         ' Folie hinzufügen
         .Slides.Add Index:=1, Layout:=ppLayoutTitle

      End With

   Else

      ' Präsentation öffnen
       Set objPptPresent = objPptApp.Presentations.Open( _
          strPPTPath)

      With objPptPresent

         If blnPrint Then

            ' Drucken
            .PrintOut

            Application.Wait Now + TimeSerial(0, 0, 5)

            ' Präsentation schließen
            .Close

            objPptApp.Quit
```

Listing 13.7 (Forts.)
Beispiele\13_UserForms\
13_01_FremdeAnwendungen.xls\
mdlPPTOle

```
        Else
            If blnOpen Then

                ' Sichtbar machen
                objPptApp.Visible = msoTrue

            Else

                ' Show starten
                .SlideShowSettings.Run

            End If

        End If

    End With
End If

End Sub
```

testOle

In dieser Prozedur wird für fast jede Aktion, die getestet werden soll, ein Dialog zur Dateiauswahl gestartet. Dieser Dateipfad wird dann an die Prozedur `PptOLENewShowPrint` übergeben. Ohne Übergabe eines weiteren Parameters, wird die Präsentation gestartet.

Wenn als zweiter Parameter der Wahrheitswert `True` übergeben wird, wird die Präsentation gedruckt, wird der dritte Parameter auf `True` gesetzt, öffnen Sie sie zum Bearbeiten. Wird überhaupt kein Parameter übergeben, wird eine neue Präsentation angelegt.

PptOLENewShowPrint

Nachdem Sie mit

```
Set objPptApp = New PowerPoint.Application
```

ein `Application`-Objekt von PowerPoint angelegt hat, wird es aktiviert.

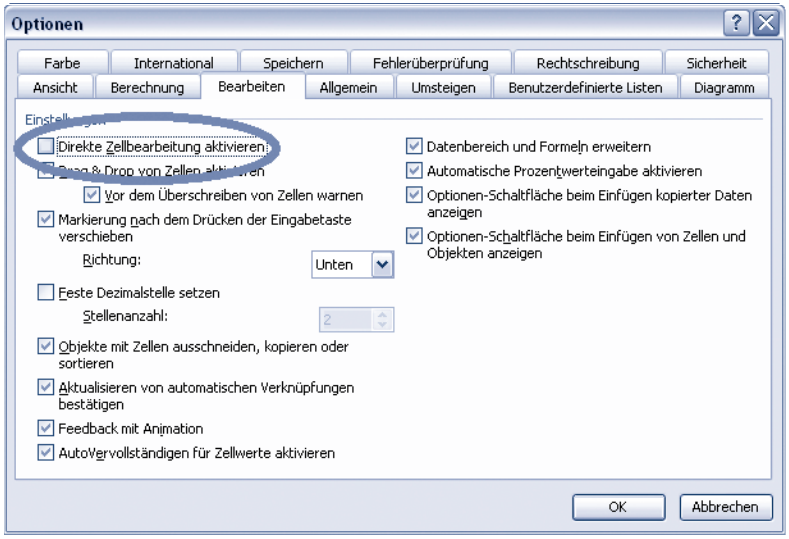
Wurde der Dateipfad nicht übergeben, wird mit der `Add`-Methode der `Presentations`-Auflistung eine neue Präsentation angelegt und eine Folie hinzugefügt.

Ist ein Dateipfad übergeben worden, wird die Präsentation gestartet oder geöffnet, je nachdem, ob der Parameter `blnOpen` auf `True` gesetzt worden ist. Ist der Parameter `blnPrint` auf `True` gesetzt worden, wird die Präsentation gedruckt und Excel mit der `Wait`-Methode ein paar Sekunden außer Gefecht gesetzt, um PowerPoint etwas Zeit zum Anstoßen des Druckes zu geben.

13.8 Wahlhilfe benutzen

Um aus einem Tabellenblatt heraus Telefonnummern zu wählen, bedienen Sie sich im folgenden Beispiel des Ereignisses `Worksheet_BeforeDoubleClick`. Dazu entfernen Sie am besten unter EXTRAS | OPTIONEN | BEARBEITEN den Haken bei DIREKTE ZELLBEARBEITUNG AKTIVIEREN (Abbildung 13.3), um nicht immer in den Bearbeitungsmodus zu gelangen. Ich persönlich empfinde diese Option ohnehin als ziemlich überflüssig.

Abbildung 13.3
Direkte Zellbearbeitung deaktivieren



Ist das der Fall, kann die folgende Ereignisprozedur (Listing 13.8) in das Klassenmodul eines Tabellenblattes eingefügt werden:

Listing 13.8
Beispiele\13_UserForms\13_01_FremdeAnwendungen.xls\Tabelle4

```
Private Sub Worksheet_BeforeDoubleClick( _
   ByVal Target As Range, _
   Cancel As Boolean)

   Dim blnRet As Boolean

   With Target
       If .Column = 1 Then blnRet = myDial(.Value)
       If .Column = 2 Then blnRet = myDial(.Value, True)
       If .Column = 3 Then blnRet = myDial(.Value, True, True)
       If blnRet = False Then _
               MsgBox "Verbindung nach :" & _
               .Value & vbCrLf & "fehlgeschlagen!"
   End With
End Sub
```

Bei einem Doppelklick in Spalte A wird die Telefonnummer in der Zelle gewählt, die durch `Target` referenziert ist, in Spalte B wird als Präfix für die Amtsholung in Nebenstellenanlagen eine Null hinzugefügt und in Spalte C ein Komma für eine Wartezeit und die Null zur Amtsholung.

Die folgende Funktion (Listing 13.9) stellt die eigentliche Funktionalität zur Verfügung. Selbstverständlich muss die Wahlhilfe korrekt funktionieren, da die API-Funktion `tapiRequestMakeCall` diese benutzt.

Listing 13.9
Beispiele\13_UserForms\
13_01_FremdeAnwendungen.xls\
mdlDial

```
Private Declare Function tapiRequestMakeCall _
   Lib "Tapi32.dll" ( _
   ByVal Nummer As String, _
   ByVal AppName As String, _
   ByVal AnruferName As String, _
   ByVal Kommentar As String _
   ) As Long

Public Function myDial(ByVal strTNumber As String, _
   Optional blnWithNull As Boolean, _
   Optional blnIdleTime As Boolean)

   Dim i          As Long
   Dim strDummy   As String


   strTNumber = Trim(strTNumber)

   ' Nur die Ziffern behalten
   For i = 1 To Len(strTNumber)
       If IsNumeric(Mid(strTNumber, i, 1)) Then
           strDummy = strDummy & Mid(strTNumber, i, 1)
       End If
   Next

   If blnIdleTime Then _
      strDummy = "," & strDummy 'Mit Wartezeit (bei Modem)

   If blnWithNull Then _
      strDummy = "0" & strDummy 'Mit Amtsholung

   strTNumber = strDummy

   If IsNumeric(strTNumber) Then

      If MsgBox("Wollen Sie die Nummer : " & _
         strTNumber & "   wählen?", vbYesNo, "myDial") _
         = vbYes Then

         ' Bei Erfolg den Wahrheitswert True zurückgeben
         If tapiRequestMakeCall(strTNumber, "", "Michael", "") = 0 _
            Then myDial = True

      End If

   Else

       MsgBox strTNumber, vbInformation, _
       "Keine gültige Telefonnummer"

   End If
End Function
```

Zu Beginn wird der String mit der Telefonnummer erzeugt. In einer Schleife werden aus dem übergebenen String nur die Ziffern extrahiert, alle anderen Zeichen werden entfernt. Ist der Parameter `blnIdleTime` gesetzt, kommt als Präfix ein Komma als zusätzliche Wartezeit hinzu, ist `blnWithNull` gesetzt, wird an die erste Stelle eine Null zur Amtsholung in Nebenstellenanlagen gesetzt.

14

Netzwerke und Internet

14.1 Was Sie in diesem Kapitel erwartet

Netzwerke und das Internet gehören heutzutage fast zu jedem normalen Arbeitsplatz. In diesem Kapitel werden einige Beispiele vorgestellt, die sich mit diesen Themen beschäftigen.

Im ersten Beispiel wird gezeigt, wie Sie an die Informationen über Drucker, freigegebene Verzeichnisse oder Laufwerke der verfügbaren Netzwerkressourcen kommen.

In einem anderem Beispiel wird eine Klasse vorgestellt, mit der Sie den Nameserver anzapfen können, um aus einer IP-Adresse den Hostnamen geliefert zu bekommen und umgekehrt. Weiterhin haben Sie die Möglichkeit, einen Ping abzusetzen und die Laufzeit auszuwerten. Mithilfe des TTL-Zählers können Sie sogar das Tool *Tracert* nachbauen, das die Laufzeit bis zu jeden einzelnen Hop auf dem Weg zum Ziel liefert.

Im dritten Beispiel wird eine Klasse benutzt, die es ermöglicht, Internetseiten im Quelltext auszulesen. Mittels `InStr`, `Mid$` und Co. können Sie sich dann daraus beispielsweise Börsenkurse extrahieren oder die heruntergeladene Seite als Text bzw. Hypertextdokument speichern.

Des Weiteren können Sie sich mit einem FTP-Server über einen Usernamen und dem entsprechenden Passwort verbinden, Verzeichnisse anlegen, löschen, auslesen, zwischen den Verzeichnissen wechseln, Dateien löschen, umbenennen sowie Dateien hoch- und herunterladen.

Mit ein paar Zeilen Code können Sie auch kurze Nachrichten an andere Netzwerkbenutzer senden.

14.2 Netzwerkressourcen

Mithilfe einiger API-Funktionen ist es möglich, die vorhandenen Ressourcen eines Netzwerkes wie Drucker, freigegebene Verzeichnisse oder Laufwerke in einer Liste auszugeben. Die Netzwerkressourcen sind dabei aus Sicht der API in

einer Baumstruktur aufgebaut. Begonnen wird bei der Wurzel (`Root`). Diese Wurzel dient als Container für alle anderen Ressourcen.

Die einzelnen Ressourcen können auch als Container dienen und wiederum Ressourcen aufnehmen. An solch einem Knoten verzweigt sich der Baum und bildet separate Äste. In VB würde für diese Darstellung das `Treeview` Steuerelement benutzt. Der Einfachheit halber wird bei diesem Beispiel darauf verzichtet und stattdessen eine einfache Liste generiert.

Es gilt also, die Ressourcen eines Containers nacheinander auszulesen. Wenn Sie dabei auf einen Container stoßen, müssen auch erst alle Elemente dieses Containers ausgelesen werden. Das schreit geradezu nach rekursiven Funktionsaufrufen. Rekursionen sind übrigens gar nicht so kompliziert, wie immer behauptet wird. Wenn Sie das Konzept einmal verstanden haben, können Sie sich damit viel Schreibarbeit und noch mehr Hirnschmalz sparen. Es ist in den meisten Fällen nämlich weitaus komplizierter, das Gleiche iterativ zu realisieren.

14.2.1 Anstoßen der Suche

Mit folgendem Klickereignis (Listing 14.1) eines Buttons wird die Suche gestartet:

Listing 14.1
Beispiele\14_Netzwerk-Internet\14_01_Netressource.xls\Tabelle1

```
Private Sub cmdNetzwerkressourcen_Click()
   Dim udtMyNetwork()   As MyNetwork
   Dim i                As Long

   ' Infos holen
   ReadNetwork udtMyNetwork

   Cells.ClearContents
   Cells(7, 1) = "Remote-Name"
   Cells(7, 2) = "Parent"
   Cells(7, 3) = "Ress./Container"
   Cells(7, 4) = "Display-Typ"
   Cells(7, 5) = "Provider"
   Cells(7, 6) = "LocalName"
   Cells(7, 7) = "Kommentar"

   For i = 1 To UBound(udtMyNetwork)

       With udtMyNetwork(i)
           Cells(i + 7, 1) = .strRemoteName
           Cells(i + 7, 2) = .strParent
           Cells(i + 7, 3) = .strUsage
           Cells(i + 7, 4) = .strDisplaytype
           Cells(i + 7, 5) = .strProvider
           Cells(i + 7, 6) = .strLocalName
           Cells(i + 7, 7) = .strComment
       End With

   Next
End Sub
```

cmdNetzwerkressourcen_Click

Die Funktion `ReadNetwork` (Listing 14.2), die im Klickereignis `cmdNetzwerkressourcen_Click` (Listing 14.1) aufgerufen wird, benötigt als Parameter ein dynamisches Datenfeld vom Typ `MyNetwork`. Dieses bekommt den Namen `udtMyNetwork` und liefert die Informationen über die gefundenen Ressourcen zurück.

Nachdem die Funktion bei ihrer Rückkehr im Parameter `udtMyNetwork` die Infos geliefert hat, werden alle Zellen des Tabellenblatts geleert und die Spaltenüberschriften eingetragen. In einer Schleife werden danach alle Informationen aus dem Datenfeld `udtMyNetwork` ins Tabellenblatt eingetragen.

14.2.2 Die Suche nach Ressourcen

Im nachfolgenden Code (Listing 14.2), der in ein Standardmodul gehört, erfolgt die eigentliche Suche nach Ressourcen. Wenn sehr große Netzwerke durchsucht werden, kann die Ausführung aber doch recht lange dauern. In diesem Fall müssten Sie die Suche so anpassen, dass immer nur einzelne Abschnitte durchsucht werden.

Listing 14.2
Beispiele\14_Netzwerk-Internet\14_01_Netressource.xls\mdlNetzwerkumgebung

```
Private Declare Function WNetCloseEnum _
   Lib "mpr.dll" ( _
   ByVal hEnum As Long _
   ) As Long

Private Declare Function WNetEnumResource _
   Lib "mpr.dll" Alias "WNetEnumResourceA" ( _
   ByVal hEnum As Long, _
   lpcCount As Long, _
   lpBuffer As Any, _
   lpBufferSize As Long _
   ) As Long

Private Declare Function WNetOpenEnum _
   Lib "mpr.dll" Alias "WNetOpenEnumA" ( _
   ByVal dwScope As Long, _
   ByVal dwType As Long, _
   ByVal dwUsage As Long, _
   lpNetResource As Any, _
   lphEnum As Long _
   ) As Long

Private Declare Function lstrlen _
   Lib "kernel32" ( _
   ByVal str As Long _
   ) As Long

Private Declare Function lstrcpy _
   Lib "kernel32" ( _
   ByVal dest As String, _
   ByVal src As Long _
   ) As Long
```

Listing 14.2 (Forts.)
Beispiele\14_Netzwerk-Internet\14_01_Netressource.xls\mdlNetzwerkumgebung

```
Private Declare Sub CopyMemory _
   Lib "kernel32" Alias "RtlMoveMemory" ( _
   Destination As Any, _
   Source As Any, _
   ByVal Length As Long)

Public Type MyNetwork
   strParent As String
   strDisplaytype As String
   strUsage As String
   strLocalName As String
   strRemoteName As String
   strComment As String
   strProvider As String
End Type

Private Type NETRESOURCE
   dwScope As Long
   dwType As Long
   dwDisplayType As Long
   dwUsage As Long
   lpLocalName As Long
   lpRemoteName As Long
   lpComment As Long
   lpProvider As Long
End Type
'
Private Const RESOURCEUSAGE_ALL = &H0&
Private Const RESOURCEUSAGE_CONNECTABLE = &H1&
Private Const RESOURCEUSAGE_lngPtrNetres = &H2&
'
Private Const RESOURCEDISPLAYTYPE_DIRECTORY = &H9&
Private Const RESOURCEDISPLAYTYPE_DOMAIN = &H1&
Private Const RESOURCEDISPLAYTYPE_FILE = &H4&
Private Const RESOURCEDISPLAYTYPE_GENERIC = &H0&
Private Const RESOURCEDISPLAYTYPE_GROUP = &H5&
Private Const RESOURCEDISPLAYTYPE_NETWORK = &H6&
Private Const RESOURCEDISPLAYTYPE_ROOT = &H7&
Private Const RESOURCEDISPLAYTYPE_SERVER = &H2&
Private Const RESOURCEDISPLAYTYPE_SHARE = &H3&
Private Const RESOURCEDISPLAYTYPE_SHAREADMIN = &H8&
'
Private Const RESOURCETYPE_ANY = &H0&
Private Const RESOURCETYPE_DISK = &H1&
Private Const RESOURCETYPE_PRINT = &H2&
'
Private Const RESOURCE_CONNECTED = &H1&
Private Const RESOURCE_GLOBALNET = &H2&
Private Const RESOURCE_REMEMBERED = &H3&

Public Function ReadNetwork( _
   udtNetEnum() As MyNetwork, _
   Optional strParent As String, _
   Optional lngResIndex As Long, _
   Optional lngPtrNetres As Long _
   ) As Long
```

Listing 14.2 (Forts.)
Beispiele\14_Netzwerk-Internet\
14_01_Netressource.xls\
mdlNetzwerkumgebung

```
Dim lngNetHandle           As Long
Dim lngSize                As Long
Dim lngCounter             As Long
Dim lngRet                 As Long
Dim lngCount               As Long
Dim lngPos                 As Long
Dim strDisplaytype         As String
Dim strLocalName           As String
Dim strRemoteName          As String
Dim strComment             As String
Dim strProvider            As String
Dim udtNetzressource(1024) As NETRESOURCE

If lngResIndex = 0 Then

   ' Root finden, wenn begonnen wird
   lngRet = WNetOpenEnum( _
      RESOURCE_GLOBALNET, _
      RESOURCETYPE_ANY, _
      RESOURCEUSAGE_ALL, _
      ByVal 0&, _
      lngNetHandle)

Else

   ' Die Struktur udtNetzressource(0) mit Daten
   ' füllen. Es wurde ein Pointer darauf übergeben,
   ' um die Parameter optional zu machen, damit
   ' am Anfang keine Struktur NETRESOURCE
   ' übergeben werden muss
   CopyMemory udtNetzressource(0), ByVal lngPtrNetres, 32

   ' Ressourcen im Container udtNetzressource(0)
   ' auflisten
   lngRet = WNetOpenEnum( _
      RESOURCE_GLOBALNET, _
      RESOURCETYPE_ANY, _
      RESOURCEUSAGE_ALL, _
      udtNetzressource(0), _
      lngNetHandle)

   ' Die lngPos im Array udtNetEnum() festlegen
   lngPos = lngResIndex

End If

' Wenn lngRet <> 0, dann Fehler
If lngRet = 0 Then

   ' Anzahl der Bytes bestimmen, die in
   ' die Struktur geschrieben werden können
   lngSize = UBound(udtNetzressource) * Len(udtNetzressource(0))
```

Listing 14.2 (Forts.)
Beispiele\14_Netzwerk-Internet\14_01_Netressource.xls\mdlNetzwerkumgebung

```
' Anzahl der vorhandenen Ressourcen ermitteln und
' gleichzeitig das Array udtNetzressource füllen
lngCount = &HFFFFFFFF
lngRet = WNetEnumResource( _
   lngNetHandle, _
   lngCount, _
   udtNetzressource(0), _
   lngSize)

If lngCount > 0 Then
' Ressourcen sind vorhanden

   For lngCounter = 0 To lngCount - 1
   'Alle Ressourcen durchlaufen

      With udtNetzressource(lngCounter)

         lngPos = lngPos + 1

         ' Das Array udtNetEnum an die Anzahl
         ' der gesamten Ressourcen anpassen
         ReDim Preserve udtNetEnum(0 To lngPos)

         ' Auslesen, was für ein Typ die Ressource ist
         Select Case .dwDisplayType
            Case RESOURCEDISPLAYTYPE_DIRECTORY
               strDisplaytype = "Directory"
            Case RESOURCEDISPLAYTYPE_DOMAIN
               strDisplaytype = "Domäne"
            Case RESOURCEDISPLAYTYPE_FILE
               strDisplaytype = "Datei"
            Case RESOURCEDISPLAYTYPE_GENERIC
               strDisplaytype = "Generic"
            Case RESOURCEDISPLAYTYPE_GROUP
               strDisplaytype = "Group"
            Case RESOURCEDISPLAYTYPE_NETWORK
               strDisplaytype = "Netzwerk"
            Case RESOURCEDISPLAYTYPE_ROOT
               strDisplaytype = "Root"
            Case RESOURCEDISPLAYTYPE_SERVER
               strDisplaytype = "Server"
            Case RESOURCEDISPLAYTYPE_SHARE
               strDisplaytype = "Share"
            Case RESOURCEDISPLAYTYPE_SHAREADMIN
               strDisplaytype = "ShareAdmin"
            Case Else
               strDisplaytype = ""
         End Select

         ' Das Array udtNetEnum mit ermittelten Daten füllen

         strLocalName = StringFromPointer(.lpLocalName)
         strRemoteName = StringFromPointer(.lpRemoteName)
         strComment = StringFromPointer(.lpComment)
         strProvider = StringFromPointer(.lpProvider)
```

Listing 14.2 (Forts.)
Beispiele\14_Netzwerk-Internet\
14_01_Netressource.xls\
mdlNetzwerkumgebung

```
            With udtNetEnum(lngPos)
               .strDisplaytype = strDisplaytype
               .strParent = strParent
               .strComment = strComment
               .strLocalName = strLocalName
               .strProvider = strProvider
               .strRemoteName = strRemoteName
               .strUsage = "Ressource"
               Application.StatusBar = "Nr.: " & lngPos & _
                  "   Remotename = " & strRemoteName
            End With

            If .dwUsage And RESOURCEUSAGE_lngPtrNetres Then
               ' Wenn das Element ein Container ist, die
               ' gleiche Funktion rekursiv aufrufen
               udtNetEnum(lngPos).strUsage = "Container"
               lngPos = ReadNetwork( _
                  udtNetEnum, _
                  strRemoteName, _
                  lngPos, _
                  VarPtr(udtNetzressource(lngCounter)))
            End If

         End With

      Next lngCounter

   End If

End If

' Rückgabewert ist die aktuelle Position im
' Array udtNetEnum
ReadNetwork = lngPos

'Enum beenden und Handle schließen
WNetCloseEnum lngNetHandle

Application.StatusBar = False

End Function
Private Function StringFromAsciiZ(ASCIIZ As String)
   'ASCIIZ-String kürzen

   If InStr(1, ASCIIZ, Chr(0)) > 0 Then
      StringFromAsciiZ = _
         Left$(ASCIIZ, InStr(1, ASCIIZ, Chr(0)) - 1)
   Else
      StringFromAsciiZ = _
         ASCIIZ
   End If
End Function
```

Listing 14.2 (Forts.)
Beispiele\14_Netzwerk-Internet\14_01_Netressource.xls\mdlNetzwerkumgebung

```
Private Function StringFromPointer(lngAscii As Long)
   ' Aus einem Pointer einen String machen

   Dim lngCount   As Long
   Dim strName    As String

   If lngAscii = 0 Then Exit Function

   ' Länge des Strings bestimmen
   lngCount = lstrlen(lngAscii)

   ' Puffer anlegen
   strName = String(lngCount, 0)

   ' Den String ab dem Pointer in den Puffer kopieren
   lstrcpy strName, lngAscii

   If InStr(1, strName, Chr(0)) <> 0 Then
      ' Beim ersten Auftreten von chr(0) kürzen
      strName = Left$(strName, InStr(1, strName, Chr(0)) - 1)
   End If

   ' String zurückgeben
   StringFromPointer = strName

End Function
```

ReadNetwork

Nun zu der Funktion `ReadNetwork`, dem eigentlichen Arbeitstier. Im Funktionskopf werden verschiedene Parameter übergeben:

- `udtNetEnum() As MyNetwork`

  Dieser Parameter nimmt die eigentlichen Daten der Ressourcen auf. Da er `ByRef` (standardmäßig) übergeben wurde, wird die beim ersten Aufruf übergebene Variable geändert, unabhängig davon, wie viele Rekursionen durchlaufen werden. Das hat den schönen Nebeneffekt, dass keine Variablen benötigt werden, die auf Modulebene deklariert sind.

  Hier eine Beschreibung des selbst angelegten Datentyps `MyNetwork`:

```
Public Type MyNetwork
   strParent As String
   strDisplayType As String
   strUsage As String
   strLocalName As String
   strRemoteName As String
   strComment As String
   strProvider As String
End Type
```

  - `strParent`

    Gibt den Container zurück, in dem sich diese Ressource befindet.

  - `strDisplayType`

    Gibt an, wie eine Ressource im Explorer angezeigt wird.

Folgende Werte sind möglich:

`Directory`, `Domäne`, `Datei`, `Generic`, `Group`, `Netzwerk`, `Root`, `Server`, `Share`, `ShareAdmin`.

- `strUsage`

  Gibt an, ob es sich um eine Ressource oder einen Container handelt.

- `strLocalName`

  Gibt den lokalen Namen einer Netzwerkressource an.

- `strRemoteName`

  Gibt den Remotenamen einer Netzwerkressource an.

  Beispiel: *\\PATRICK3\SharedDocs, Microsoft-Terminaldienste*

- `strComment`

  Gibt den Kommentar einer Netzwerkressource zurück.

  Beispiel: HP PSC 950, Microsoft-Terminaldienste

- `strProvider`

  Gibt den Provider einer Netzwerkressource zurück.

  Beispiel: Microsoft Windows-Netzwerk, Web Client Network

- `Optional strParent As String`

  Dieser optionale Parameter enthält den Namen des Containers, dessen Netzwerkressourcen durchsucht werden sollen. Dieser wird unverändert in der Struktur `MyNetwork` als Element `strParent` gespeichert.

- `Optional lngResIndex As Long`

  Dieser optionale Parameter enthält die aktuelle Position im Array vom Typ `MyNetwork`, das die Informationen aufnimmt.

- `Optional lngPtrNetres As Long`

  Dieser optionale Parameter zeigt auf eine Speicherstelle, ab der sich eine Struktur vom Typ `NETRESOURCE` befindet, die den Container beschreibt, dessen enthaltene Ressourcen ermittelt werden sollen. Sie könnten zwar anders deklarieren und an dieser Stelle einen Parameter vom Typ `NETRESOURCE` übergeben, aber diese Übergabe lässt sich nicht optional machen. Somit müssten Sie auch beim ersten Aufruf eine solche Struktur übergeben, obwohl diese erst bei den rekursiven Aufrufen benötigt wird.

  Wird nichts oder Null übergeben, wird die Wurzel (Root) durchsucht.

Wenn der Parameter `lResIndex` nicht übergeben wurde oder Null ist, wird die Funktion `WNetOpenEnum` mit dem auf Null gesetzten Parameter `lpNetResource` aufgerufen. Somit wird die Suche nach Ressourcen bei der Wurzel aufgenommen. Der Parameter `lphEnum` der API-Funktion `WNetOpenEnum` liefert nach dem Funktionsaufruf ein Enum-Handle.

Dieses Handle wird zusammen mit einem Puffer und dessen Größe an die API-Funktion `WnetEnumResource` übergeben. Der Puffer ist als Array vom Typ `NETRESOURCE` angelegt. Der Parameter `lpcCount` liefert nach der Rückkehr die Anzahl der von der API-Funktion in den Puffer geschriebenen Einträge zurück.

Danach werden die Informationen aus dem Puffer ausgelesen und damit das ursprünglich übergebene dynamische Datenfeld vom Typ `MyNetwork` gefüllt.

Handelt es sich bei einer gefundenen Netzwerkressource (Typ `NETRESOURCE`) um einen Container, ruft sich die Funktion `ReadNetwork` nochmals auf und Sie übergeben ihr im Parameter `lngPtrNetres` einen Zeiger auf diese Struktur im Speicher. Dazu wird die undokumentierte Funktion `VarPtr` benutzt, die solch einen Zeiger als Wert liefert. Sie erkennen einen Container daran, dass das Element `.dwUsage` den Wert `RESOURCEUSAGE_CONTAINER` angenommen hat.

StringFromAsciiZ

Oft wird von einer API-Funktion ein nullterminierter String in einen übergebenen Puffer geschrieben. Aber nicht immer wird dabei die Länge des Strings mitgeliefert. Diese Funktion sucht nach den ersten Auftreten eines `CHR(0)` Zeichens und liefert den String bis zu dieser Stelle zurück.

Übrigens kann ein vergessenes `CHR(0)`-Zeichen in einem String zu Fehlern führen, die einen zur Verzweiflung treiben können. Beispielsweise wird bei einer Ausgabe in eine Messagebox die Ausgabe beim ersten Auftreten dieses Zeichens abgebrochen, alles, was dahinter steht, wird ignoriert.

StringFromPointer

Manchmal wird von einer API-Funktion nur ein Pointer auf einen nullterminierten String geliefert. Die Funktion `StringFromPointer` holt sich den String von dieser Speicherstelle.

Dazu wird mit `lstrlen` die Länge des Strings im Speicher ermittelt. Anschließend wird ein Stringpuffer in dieser Länge erzeugt und mittels `lstrcpy` der String ab dem Pointer in den vorher angelegten Puffer kopiert.

14.3 Tracert

Die Bibliotheken `wsock32.dll` und `icmp.dll` bieten eine Menge Funktionen, die sich mit dem Internet beschäftigen. In diesem Beispiel stelle ich eine Klasse vor, die viele dieser Funktionen kapselt und folgende Aufgaben erledigt:

- Sie können an einen Host einen Ping absetzen und die Laufzeit des Paketes ermitteln.
- Es ist auch möglich, den Nameserver anzuzapfen, um aus einer IP den Hostnamen geliefert zu bekommen. Auch der umgekehrte, häufiger benutzte Weg kann gegangen werden und Sie können aus einen gegebenen Hostnamen eine IP machen.
- Die eigene IP-Adresse und der eigene Hostname kann über eine Eigenschaft der Klasse ausgelesen werden.
- Indem Sie den Time To Live-Zähler (TTL) manipulieren, können Sie eine Route mit allen Hops (Zwischenstationen) bis zum Ziel verfolgen. Als Ergebnis erhalten Sie bei dem vorliegenden Code die IP-Nummer, den Hostnamen und die Laufzeit bis zu dem gewünschten Hop.

14.3.1 Benutzen der Klasse clsTracert

Listing 14.3
Beispiele\14_Netzwerk-Internet\
14_02_Netzwerk.xls\Tabelle1

```
Private Sub cmdHostName_Click()
   Dim myTracert As New clsTracert
   MsgBox myTracert.Convert_IP_To_Host(Range("C6"))
End Sub

Private Sub cmdIP_Click()
   Dim myTracert As New clsTracert
   MsgBox myTracert.Convert_Host_To_IP(Range("C4"))
End Sub

Private Sub cmdMyHost_Click()
   Dim myTracert As New clsTracert
   MsgBox myTracert.MyHostName
End Sub

Private Sub cmdMyIP_Click()
   Dim myTracert As New clsTracert
   MsgBox myTracert.MyHostIP
End Sub

Private Sub cmdPing_Click()
   Dim myTracert  As New clsTracert
   Dim strMsg     As String

   With myTracert

      .DestHost = Me.Range("C8").Value
      .Hop = Me.Range("D8").Value

      .VBAPing

      MsgBox .Result

   End With

End Sub

Private Sub cmdTracert_Click()
   Dim myTracert  As New clsTracert
   Dim strMsg     As String
   Dim i          As Long
   Dim strHop     As String * 10
   Dim strIP      As String * 30
   Dim strHost    As String * 50
   Dim strTime    As String * 20

   Dim strDestIP  As String
   Dim strDestHost  As String

   With myTracert
      strDestHost = Me.Range("C10").Value
      .DestHost = strDestHost
      strDestIP = .DestIP
      Application.StatusBar = "Tracert"
```

Listing 14.3 (Forts.)
Beispiele\14_Netzwerk-Internet\14_02_Netzwerk.xls\Tabelle1

```
        Do
            i = i + 1

            .Hop = i

            .VBAPing

            strHop = "Hop : " & i
            strIP = "IP : " & .CurIP
            strHost = "Host : " & .CurHost
            strTime = "Time : " & .CurRetTime & " mSec"

            Application.StatusBar = strHop & vbTab & _
               strIP & vbTab & strTime & vbTab & strHost

            strMsg = strMsg & strHop & vbTab & _
               strIP & vbTab & _
               strTime & vbTab & _
               strHost & vbCrLf

            If i > 40 Then Exit Do

        Loop While .DestIP <> .CurIP

        MsgBox strMsg, , strDestHost & "     " & .DestIP

    End With

    Application.StatusBar = False

End Sub
```

cmdHostName_Click

Es wird eine Instanz der Klasse `clsTracert` angelegt und an die Funktion `Convert_IP_To_Host` wird eine IP-Adresse übergeben, die in Zelle C6 steht. Diese wird, wenn möglich, in einen Hostnamen umgewandelt und in einer Messagebox ausgegeben.

cmdIP_Click

Es wird eine Instanz der Klasse `clsTracert` angelegt und an die Funktion `Convert_Host_To_IP` wird ein Hostname übergeben, der in Zelle C4 steht. Dieser wird, wenn möglich, in eine IP-Adresse umgewandelt und in einer Messagebox ausgegeben.

cmdMyHost_Click

Es wird eine Instanz der Klasse `clsTracert` angelegt, die Eigenschaft `MyHostName`, die den eigenen Hostnamen liefert, wird ausgelesen und in einer Messagebox ausgegeben.

cmdMyIP_Click

Es wird eine Instanz der Klasse `clsTracert` angelegt, die Eigenschaft `MyHostIP`, die die eigene IP liefert, wird ausgelesen und in einer Messagebox ausgegeben.

cmdPing_Click

Es wird eine Instanz der Klasse `clsTracert` angelegt, die Eigenschaften `DestHost` und `Hop` werden gesetzt, die Methode VBAPing aufgerufen und das Ergebnis über die Eigenschaft `Result` ausgelesen und in einer Messagebox ausgegeben.

cmdTracert_Click

Es wird eine Instanz der Klasse `clsTracert` angelegt und die Eigenschaft `DestHost` gesetzt.

Zuerst setzen Sie den TTL-Zähler auf eins, indem Sie an die Eigenschaft `Hop` diesen Wert übergeben. Anschließend rufen Sie die Methode `VBAPing` auf.

Diese sendet eine `ICMP`-Nachricht an das Ziel. Das `ICMP` (Internet Control Message Protocol) ist ein Hilfsprotokoll, mit dem Fehler- und Informationsmeldungen zwischen Rechnern im Netz ausgetauscht werden. Im RFC (Request for Comments) 792 und 1256 ist das Protokoll genau beschrieben und wenn Sie sich dafür interessieren, bemühen Sie einfach einmal eine Suchmaschine. Sie werden überrascht sein, wie viel frei zugängliche Informationen Sie darüber finden werden.

Beim jeder Vermittlungsstelle im Netz wird der TTL-Zähler um 1 vermindert. Der Countdown ist am Hop 1 abgelaufen, weil dort der Zähler auf null steht. Das Paket wird deshalb verworfen und es wird eine ICMP-Nachricht zurückgesendet. Diese Rückmeldung kann ausgelesen und das Ergebnis ausgewertet werden.

Als nächsten Schritt erhöhen Sie den TTL um eins und wiederholen die Prozedur. Das machen Sie so lange, bis das Ziel erreicht ist. An jedem Hop, an dem der TTL Zähler den Wert null erreicht, wird eine ICMP-Nachricht abgesetzt. Voraussetzung bei der ganzen Sache ist natürlich, dass der Betreiber dieser Vermittlungsstelle das Absetzen solcher Nachrichten überhaupt erlaubt.

Die Eigenschaften `CurIP`, `CurHost` und `CurRetTime` der Klasse liefern die Informationen über einen Hop, werden zu einem String zusammengefasst und am Ende in einer Messagebox ausgegeben.

14.3.2 Die Klasse clsTracert

Listing 14.4
Beispiele\14_Netzwerk-Internet\14_02_Netzwerk.xls\clsTracert

```
Private Const MIN_SOCKETS_REQD As Long = 1
Private Const SOCKET_ERROR As Long = -1
Private Const MAX_WSADescription = 256
Private Const MAX_WSASYSStatus = 128
Private Const AF_INET As Long = 2

Private Type Hostent
   hName As Long
   hAliases As Long
   hAddrType As Integer
   hLen As Integer
   hAddrList As Long
End Type
```

Listing 14.4 (Forts.)
Beispiele\14_Netzwerk-Internet\14_02_Netzwerk.xls\clsTracert

```
Private Type WSAdata
   wVersion As Integer
   wHighVersion As Integer
   szDescription(0 To MAX_WSADescription) As Byte
   szSystemStatus(0 To MAX_WSASYSStatus) As Byte
   iMaxSockets As Integer
   iMaxUdpDg As Integer
   lpVendorInfo As Long
End Type

Private Type IP_OPTION_INFORMATION
   ttl As Byte
   Tos As Byte
   Flags As Byte
   OptionsSize As Long
   OptionsData As String * 128
End Type

Private Type IP_ECHO_REPLY
   Address(0 To 3) As Byte
   Status As Long
   RoundTripTime As Long
   DataSize As Integer
   Reserved As Integer
   data As Long
   Options As IP_OPTION_INFORMATION
   ReturnedData As String * 256
End Type

Private Declare Sub CopyMemory _
   Lib "kernel32" Alias "RtlMoveMemory" ( _
   hpvDest As Any, _
   ByVal hpvSource As Long, _
   ByVal cbCopy As Long)

Private Declare Function lstrcpy _
   Lib "kernel32" Alias "lstrcpyA" ( _
   ByVal lpString1 As String, _
   ByVal lpString2 As Long _
   ) As Long

Private Declare Function gethostbyaddr _
   Lib "wsock32" ( _
   szHost As Any, _
   ByVal dwHostLen As Integer, _
   dwSocketType As Integer _
   ) As Long

Private Declare Function GetHostByName _
   Lib "wsock32.dll" Alias "gethostbyname" ( _
   ByVal Hostname As String _
   ) As Long

Private Declare Function WSAStartup _
   Lib "wsock32" ( _
   ByVal wVersionRequired As Long, _
   lpWSAdata As WSAdata _
   ) As Long
```

Listing 14.4 (Forts.)
Beispiele\14_Netzwerk-Internet\
14_02_Netzwerk.xls\clsTracert

```
Private Declare Function WSACleanup _
   Lib "wsock32.dll" () As Long

Private Declare Function inet_addr _
   Lib "wsock32" ( _
   ByVal cp As String _
   ) As Long

Private Declare Function inet_ntoa _
   Lib "wsock32" ( _
   ByVal in_addr As Long _
   ) As Long

Private Declare Function IcmpCreateFile _
   Lib "icmp.dll" () As Long

Private Declare Function IcmpCloseHandle _
   Lib "icmp.dll" ( _
   ByVal HANDLE As Long _
   ) As Boolean

Private Declare Function IcmpSendEcho _
   Lib "ICMP" ( _
   ByVal IcmpHandle As Long, _
   ByVal DestAddress As Long, _
   ByVal RequestData As String, _
   ByVal RequestSize As Integer, _
   RequestOptns As IP_OPTION_INFORMATION, _
   ReplyBuffer As IP_ECHO_REPLY, _
   ByVal ReplySize As Long, _
   ByVal Timeout As Long _
   ) As Boolean

Private Declare Function gethostname _
   Lib "wsock32.dll" ( _
   ByVal szHost As String, _
   ByVal dwHostLen As Long _
   ) As Long

Private Type lngIP
   lngIP As Long
End Type

Private Type IP
   Byte4 As Byte
   Byte3 As Byte
   Byte2 As Byte
   Byte1 As Byte
End Type

Private mudtEcho        As IP_ECHO_REPLY
Private mstrDestHost    As String
Private mstrDestHostIP  As String
Private mbytHop         As Byte
Private mstrCurIP       As String
Private mstrCurHost     As String
Private mlngCurRetTime  As Long
```

Listing 14.4 (Forts.)
Beispiele\14_Netzwerk-Internet\14_02_Netzwerk.xls\clsTracert

```
Private mstrMessage     As String
Private mlngTimeout     As Long
Private mblnInitialize  As Long

Public Sub VBAPing()
   Dim i As Byte

   ' Hostname ermitteln
   mstrDestHost = Hostname_From_IP(mstrDestHostIP)

   ' Den Host einer modulweiten Variable zuweisen
   mstrDestHostIP = mstrDestHost

   ' Der übergebene Parameter liefert die IP vom Hostname
   Lng_IP_From_Hostname mstrDestHostIP

   With mudtEcho

      ' Ping anstoßen mit Zieladresse und begrenzten TTL
      If Ping(mstrDestHost, mudtEcho, mbytHop, mlngTimeout) Then

         ' IP zusammensetzen
         mstrCurIP = CStr(.Address(0)) & "." & _
            CStr(.Address(1)) & "." & _
            CStr(.Address(2)) & "." & _
            CStr(.Address(3))

         ' Den aktuellen Hostname ermitteln
         mstrCurHost = Hostname_From_IP(mstrCurIP)

         ' Zeit bis zum Hop zurückgeben
         mlngCurRetTime = .RoundTripTime

      Else

         mstrCurIP = "*.*.*.*"
         mstrCurHost = "?"
         mlngCurRetTime = mlngTimeout

      End If

      ' String zusammensetzen
      mstrMessage = "DestHost  : " & mstrDestHost
      mstrMessage = mstrMessage & vbCrLf & _
         "DestIP : " & mstrDestHostIP
      mstrMessage = mstrMessage & vbCrLf & _
         "CurIP : " & mstrCurIP
      mstrMessage = mstrMessage & vbCrLf & _
         "CurHost : " & mstrCurHost
      mstrMessage = mstrMessage & vbCrLf & _
         "CurTime : " & CStr(mlngCurRetTime)

   End With

End Sub
```

Listing 14.4 (Forts.)
Beispiele\14_Netzwerk-Internet\14_02_Netzwerk.xls\clsTracert

```
Private Function Ping( _
   ByVal Hostname As String, _
   udtEcho As IP_ECHO_REPLY, _
   Optional ttl As Byte, _
   Optional Timeout As Long _
   ) As Boolean

   Dim lngFile          As Long
   Dim udtOptInfo       As IP_OPTION_INFORMATION
   Dim lngHostIP        As Long
   Dim strRequestData   As String
   Dim lngTimeout       As Long

   ' Timeout festlegen
   If Timeout = 0 Then
       lngTimeout = 6000
   Else
       lngTimeout = Timeout
   End If

   ' Time to Life festsetzen
   udtOptInfo.ttl = 255
   If ttl Then udtOptInfo.ttl = ttl

   'Hostname nach Long umwandeln
   lngHostIP = Lng_IP_From_Hostname(Hostname)

   ' Wenn der Hostname eine IP ist, dann so
   If lngHostIP = 0 Then lngHostIP = lngIP_From_IP(Hostname)

   If mblnInitialize = False Then
      mblnInitialize = MyInit
      If mblnInitialize = False Then Exit Function
   End If

   ' Datenblock erzeugen, der gesendet wird
   strRequestData = String(32, "x")

   ' ICMP Filehandle besorgen
   lngFile = IcmpCreateFile()

   ' Ping absetzen
   If IcmpSendEcho(lngFile, lngHostIP, _
      strRequestData, Len(strRequestData), _
      udtOptInfo, udtEcho, Len(udtEcho) + 8, _
      lngTimeout) Then Ping = True

   ' Etwas Zeit zum Verschnaufen lassen
   DoEvents

   ' ICMP Filehandle schließen
   IcmpCloseHandle lngFile

End Function
```

Listing 14.4 (Forts.)
Beispiele\14_Netzwerk-Internet\14_02_Netzwerk.xls\clsTracert

```
Private Function MyInit() As Boolean
   Dim udtWSAData As WSAdata

   'Socket Initialisieren
   If WSAStartup(MIN_SOCKETS_REQD, udtWSAData) = SOCKET_ERROR Then
      MyInit = False
      Exit Function
   End If

   MyInit = True

End Function

Private Function IPString_From_IPLong(IP As Long) As String
   Dim ipPtr As Long
   ' Aus einer IP-Adresse Long eine gewohnte IP-Adresse machen
   ipPtr = inet_ntoa(IP)
   IPString_From_IPLong = String(16, 0)
   lstrcpy IPString_From_IPLong, ipPtr
   IPString_From_IPLong = Left$(IPString_From_IPLong, _
       InStr(1, IPString_From_IPLong, Chr(0)))

End Function

Private Function lngIP_From_IP(ByVal strIP As String) As Long
   Dim udtIP   As IP
   Dim udtLIP  As lngIP
   ' Aus einer gewohnten IP-Adresse ein Long machen

   On Error Resume Next
   udtIP.Byte4 = CLng(Left$(strIP, InStr(1, strIP, ".") - 1))
   strIP = Right$(strIP, Len(strIP) - InStr(1, strIP, "."))
   udtIP.Byte3 = CLng(Left$(strIP, InStr(1, strIP, ".") - 1))
   strIP = Right$(strIP, Len(strIP) - InStr(1, strIP, "."))
   udtIP.Byte2 = CLng(Left$(strIP, InStr(1, strIP, ".") - 1))
   strIP = Right$(strIP, Len(strIP) - InStr(1, strIP, "."))
   udtIP.Byte1 = CLng(strIP)

   LSet udtLIP = udtIP

   lngIP_From_IP = udtLIP.lngIP

End Function

Private Function Lng_IP_From_Hostname(Hoststring As String) As Long
   ' Wenn Hoststring als Referenz übergeben wurde, dann
   ' wird die IP als Variable Hoststring in gewohnter Notation
   ' zurückgegeben (192.168.100.2)

   Dim strHostName         As String * 256
   Dim lngPtrToHostent     As Long
   Dim strIPFromHostname   As String
   Dim udtHost             As Hostent
   Dim lngIP               As Long
   Dim buffer(1 To 4)      As Byte
   Dim i                   As Long

   On Error Resume Next
```

Listing 14.4 (Forts.)
Beispiele\14_Netzwerk-Internet\14_02_Netzwerk.xls\clsTracert

```
    If mblnInitialize = False Then
       mblnInitialize = MyInit
       If mblnInitialize = False Then Exit Function
    End If

    ' Nullchar anhängen
    strHostName = Hoststring & vbNullChar

    Hoststring = ""

    ' Pointer auf eine Hostent-Struktur ermitteln
    lngPtrToHostent = GetHostByName(strHostName)
    If lngPtrToHostent = 0 Then
        Exit Function
    End If

    With udtHost

        ' Aus dem Speicher in eine Hostent-Struktur
        ' kopieren
        CopyMemory udtHost, lngPtrToHostent, Len(udtHost)

        ' Pointer auf die Adresse ermitteln
        CopyMemory lngIP, .hAddrList, 4

        ' In ein Datenfeld kopieren
        CopyMemory buffer(1), lngIP, 4

        ' Gleichzeitig in ein Long kopieren
        CopyMemory Lng_IP_From_Hostname, lngIP, 4

        ' Aus dem Datenfeld in einen String
        For i = 1 To 4
            Hoststring = Hoststring _
                & buffer(i) & "."
        Next

        Hoststring = Left$(Hoststring, Len(Hoststring) - 1)

    End With

End Function

Private Function Hostname_From_IP( _
    ByVal IP_String As String _
    ) As String

    ' Aus einer IP in gewohnter Notation (192.168.100.5)
    ' wird der Hostname ermittelt

    Dim lngNetwByteOrder As Long
    Dim lp_to_Hostent    As Long
    Dim udtHost          As Hostent
    Dim buffer(1 To 4)   As Byte

    Hostname_From_IP = IP_String
```

Listing 14.4 (Forts.)
Beispiele\14_Netzwerk-Internet\14_02_Netzwerk.xls\clsTracert

```
    If mblnInitialize = False Then
       mblnInitialize = MyInit
       If mblnInitialize = False Then Exit Function
    End If

    ' IP in Long umwandeln
    lngNetwByteOrder = inet_addr(IP_String)
    If lngNetwByteOrder = True Then Exit Function

    ' In einen Buffer kopieren
    CopyMemory buffer(1), VarPtr(lngNetwByteOrder), 4

    ' Pointer auf eine Hostent-Struktur ermitteln
    lp_to_Hostent = gethostbyaddr(buffer(1), 4, AF_INET)
    If lp_to_Hostent = 0 Then Exit Function

    ' Aus dem Speicher in eine Hostent-Struktur kopieren
    CopyMemory udtHost, lp_to_Hostent, Len(udtHost)

    ' Buffer bereitstellen
    Hostname_From_IP = String(256, 0)

    ' Name in Buffer kopieren
    CopyMemory ByVal Hostname_From_IP, udtHost.hName, 255

    ' Nullchars abschneiden
    Hostname_From_IP = Left$(Hostname_From_IP, _
        InStr(1, Hostname_From_IP, vbNullChar) - 1)

End Function

Public Function Convert_IP_To_Host(IP As String) As String
    Convert_IP_To_Host = Hostname_From_IP(IP)
End Function

Public Function Convert_Host_To_IP(Host As String) As String
    Convert_Host_To_IP = Host
    Lng_IP_From_Hostname Convert_Host_To_IP
End Function

Public Function Convert_Long_To_IP(IP As String) As String
    Convert_Long_To_IP = IPString_From_IPLong(IP)
End Function

Public Function MyHostName() As String
    Dim strHost As String * 256

    If mblnInitialize = False Then
       mblnInitialize = MyInit
       If mblnInitialize = False Then Exit Function
    End If

    ' Eigenen Hostnamen ermitteln
    gethostname strHost, 256

    MyHostName = Left(strHost, InStr(1, strHost, Chr(0)) - 1)

End Function
```

Listing 14.4 (Forts.)
Beispiele\14_Netzwerk-Internet\
14_02_Netzwerk.xls\clsTracert

```
Public Function MyHostIP() As String
   Dim strHost As String * 256

   If mblnInitialize = False Then
      mblnInitialize = MyInit
      If mblnInitialize = False Then Exit Function
   End If

   ' Eigenen Hostnamen ermitteln
   gethostname strHost, 256

   MyHostIP = Left(strHost, InStr(1, strHost, Chr(0)) - 1)

   ' In IP umwandeln
   Lng_IP_From_Hostname MyHostIP

End Function

Public Property Get CurIP() As String
   CurIP = mstrCurIP
End Property

Public Property Get CurHost() As String
   CurHost = mstrCurHost
End Property

Public Property Get CurRetTime() As Long
   CurRetTime = mlngCurRetTime
End Property

Public Property Get Timeout() As Long
   Timeout = mlngTimeout
End Property

Public Property Let Timeout(ByVal vNewValue As Long)
   mlngTimeout = vNewValue
End Property

Public Property Get Hop() As Byte
   Hop = mbytHop
End Property

Public Property Let Hop(ByVal vNewValue As Byte)
   mbytHop = vNewValue
End Property

Public Property Get DestIP() As String
   DestIP = mstrDestHostIP
End Property

Public Property Let DestIP(ByVal vNewValue As String)
   mstrDestHost = Hostname_From_IP(vNewValue)
   mstrDestHostIP = mstrDestHost
   Lng_IP_From_Hostname mstrDestHostIP
End Property
```

Listing 14.4 (Forts.)
Beispiele\14_Netzwerk-Internet\14_02_Netzwerk.xls\clsTracert

```
Public Property Get DestHost() As String
   DestHost = mstrDestHost
End Property

Public Property Let DestHost(ByVal vNewValue As String)
   mstrDestHost = Hostname_From_IP(vNewValue)
   mstrDestHostIP = mstrDestHost
   Lng_IP_From_Hostname mstrDestHostIP
End Property

Public Property Get Result() As String
   Result = mstrMessage
End Property

Private Sub Class_Initialize()
   mblnInitialize = MyInit
End Sub

Private Sub Class_Terminate()
   WSACleanup
End Sub
```

Convert_Long_To_IP

Wandelt einen `Long`-Wert, der in der `Internet-Byte-Order` vorliegt, in eine IP Adresse um. Die Byteorder des `Long` ist anders als in der Windows-Welt.

Convert_Host_To_IP

Wandelt einen Hostnamen in eine IP-Adresse um.

Auch hier muss eine Internetverbindung bestehen, wenn die Funktion den Nameserver anzapfen soll.

Convert_IP_To_Host

Wandelt eine IP-Adresse in den Hostnamen um.

Selbstverständlich muss eine Internetverbindung bestehen, wenn die Funktion den Nameserver anzapfen soll.

CurHost

Aktueller Hostname.

Diese Eigenschaft liefert den Hostnamen, von dem der `EchoRequest` kommt. Bei einem gesetzten Hop ist das der Hostname des Rechners, an dem das Paket verworfen wurde.

CurIP

Aktuelle IP-Adresse.

Diese Eigenschaft liefert die IP Adresse, von dem der `EchoRequest` kommt. Beim gesetzten Hop ist das die IP-Adresse des Rechners, an dem das Paket verworfen wurde.

CurRetTime

Laufzeit des Pakets. Diese Eigenschaft liefert die Laufzeit des Pakets in Millisekunden bis zum gesetzten Hop und zurück.

DestHost

Der Hostname des Ziels. Eine der Eigenschaften `DestHost` oder `DestIP` muss gesetzt werden, die andere kann dann jeweils ausgelesen werden.

DestIP

Die IP-Adresse des Ziels. Eine der Eigenschaften `DestHost` oder `DestIP` muss gesetzt werden, die andere kann dann jeweils ausgelesen werden.

Hop

Hiermit wird der TTL-Zähler (`Time To Live`) gesetzt.

Um bei Fehlern ein unendliches Kreisen von IP-Paketen im Netz zu vermeiden, enthält jedes Paket einen Wert, der bei jeder Vermittlungsstelle (Hop) im Internet um eins vermindert wird. Ist der Zähler null, wird das Paket verworfen. Diesen Zähler nennt man den `TTL`-Zähler (Time To Live). Wird nichts gesetzt, wird der Wert `255` angenommen.

Ist dieser kleiner als die Anzahl der benötigten Hops bis zum Ziel, wird das Paket an diesem gesetzten Hop verworfen und es wird von dort eine `ICMP`-Nachricht abgesetzt.

Hostname_From_IP

Aus einer IP in der gewohnten Notation (192.168.100.5) wird der Hostname ermittelt.

Anfangs wird dabei überprüft, ob schon ein `Socket` initialisiert ist. Ist das nicht der Fall, rufen Sie die Funktion `MyInit` auf. Mithilfe der API `inet_addr` wird anschließend aus der IP ein Longwert gemacht. Dieser wird mit `CopyMemory` in ein Bytearray kopiert und dieses Array an die Funktion `GetHostbyAddr` übergeben. `GetHostbyAddr` liefert einen Zeiger auf die Struktur `Hostent` zurück. Um die gelieferten Informationen auszulesen, wird der Speicher, auf den der Zeiger verweist, mittels `CopyMemory` in eine Variable des Typs `Hostent` kopiert.

Mit `CopyMemory` wird aus dieser Struktur ein Zeiger auf das Element `hName` extrahiert. Ab dieser Speicherstelle werden mit `CopyMemory` `255` Bytes in einen Stringpuffer kopiert, dieser String beim ersten Auftreten des Zeichens `chr(0)` gekürzt und als Funktionsergebnis zurückgegeben.

IPString_From_IPLong

Aus einer IP-Adresse, die als ein Long vorliegt, wird mithilfe der API `inet_ntoa` eine gewohnte IP-Adresse gemacht. Dazu wird der String, der im Speicher ab dem zurückgelieferten Zeiger steht, mit der Funktion `lstrcpy` in einen 16 Byte großen Stringpuffer kopiert und als Funktionsergebnis zurückgegeben.

Lng_IP_From_Hostname

Diese Funktion liefert als Longwert die IP-Adresse eines als Namen übergebenen Hosts zurück. Über den Parameter `Hoststring` wird die IP-Adresse in gewohnter Notation zurückgegeben.

Anfangs wird dabei überprüft, ob schon ein `Socket` initialisiert ist. Ist das nicht der Fall, rufen Sie die Funktion `MyInit` auf. Damit ein Nameserver nach der IP abgefragt werden kann, gibt es die Funktion `GetHostByName`. Diese Funktion liefert einen Zeiger auf die Struktur `Hostent` zurück. Um die benötigte Informationen auszulesen, wird der Speicher, auf den der Zeiger verweist, mittels `CopyMemory` in eine Variable des Typs `Hostent` kopiert.

Mit `CopyMemory` wird aus dieser Struktur ein Zeiger auf das Element `hAddrList` extrahiert. Ab dieser Speicherstelle werden mit `CopyMemory` vier Bytes in ein Bytearray kopiert und daraus der IP-String gemacht. Mit `CopyMemory` werden vier Bytes in eine `Long`-Variable kopiert und als Funktionsergebnis zurückgegeben.

lngIP_From_IP

Aus einer IP-Adresse, die als String vorliegt, wird ein `Long`-Wert gemacht. Dazu wird die Struktur `udtIP` mit den einzelnen durch einen Punkt getrennten Werten gefüllt. Mit `LSet` werden die vier Byteelemente dieser Struktur in das Longelement der Struktur `udtLIP` kopiert und dieser `Long`-Wert als Funktionsergebnis zurückgegeben.

MyHostIP

Diese Eigenschaft gibt die eigene IP-Adresse zurück.

Dazu liefert die API-Funktion `gethostname` den eigenen Hostnamen, der wiederum an die Funktion `Lng_IP_From_Hostname` zum Umwandeln in eine IP übergeben wird.

MyHostName

Diese Eigenschaft gibt den eigenen Hostnamen zurück.

Dazu wird die API-Funktion `gethostname` benutzt.

MyInit

In der Funktion `MyInit` müssen Sie mittels der API-Funktion `WSAStartup` zuerst einen Kommunikationspunkt oder Socket mit dem Internet aufbauen. Sie erhalten, wenn die Aktion erfolgreich war, einen Wert zurück, der ungleich `SOCKET_ERROR` (–1) ist. Das ist die Grundlage der Internetkommunikation. Bei dieser Initialisierung erhalten Sie gleichzeitig auch Informationen über den Socket selbst, und zwar wird die Struktur `WSAData` mit Informationen gefüllt. Ich habe im vorliegenden Beispiel aber nur die Informationen Socketversion und Systemstatus verfügbar gemacht. Geschlossen wird der Socket mit der API-Funktion `WSACleanup`.

Ping

Es wird über den Hostnamen erst einmal die zugehörige IP-Adresse ermittelt. Dazu dient die Funktion `Lng_IP_From_Hostname`.

Um einen Ping abzuschicken, benötigen Sie die Funktion `IcmpSendEcho`. Diese braucht wiederum ein `Filehandle`, und zwar ein `Internet-Filehandle`. Das wird mittels der Funktion `IcmpCreateFile` erzeugt. An die Funktion `IcmpSendEcho` wird noch eine Struktur vom Typ `IP_ECHO_REPLY` übergeben, welche die zurückgelieferten Daten aufnimmt. Des Weiteren ist eine Struktur vom Typ `IP_OPTION_INFORMATION` erforderlich, die verschiedene Einstellungen entgegennimmt. Die Variable `strRequestData` enthält den Text, der auf die Reise geschickt wird und `lngTimeout` nimmt den Timeout in Millisekunden auf, nach dessen Ablauf die Funktion auf jeden Fall zurückkehrt.

Die Struktur `IP_ECHO_REPLY` liefert alle benötigten Informationen, auch den empfangenen Text. Das Funktionsergebnis von `IcmpSendEcho` gibt Auskunft über den Erfolg, der auch als Funktionsergebnis von `Ping` weitergegeben wird.

Zum Schluss wird das Filehandle und der Socket geschlossen.

Result

Liefert einen String mit den zusammengefassten Infos.

Diese Eigenschaft liefert einen String mit den Informationen, die jeweils durch einen Zeilenumbruch getrennt sind. Er kann direkt in einer `Msgbox` angezeigt werden.

Timeout

Timeout für die Funktion `IcmpSendEcho`.

Wenn nach dieser Zeit in Millisekunden kein `Echo-Request` kommt, bricht die Funktion `IcmpSendEcho` ab.

VBAPing

Hiermit wird das Pingen angestoßen.

Zu Beginn wird ein eventuell fehlender Hostname oder eine fehlende IP-Adresse ermittelt, dazu muss aber eine der beiden Klasseneigenschaften `DestHost` oder `DestIP` gesetzt sein. Danach wird die Funktion `Ping` aufgerufen.

Liefert die Funktion `Ping` den Wahrheitswert `True` für einen Erfolg zurück, wird die ausgefüllte Struktur `mudtEcho` ausgewertet, andernfalls wird die momentane IP `mstrCurIP` auf »*.*.*.*«, der aktuelle Hostname auf »?« und die Laufzeit `mlngCurRetTime` auf den Timeout gesetzt.

Schließlich wird mit diesen Informationen der String `mstrMessage` zusammengesetzt, der von außerhalb der Klasse mit der Eigenschaft `Result` ausgelesen werden kann.

14.4 FTP und HTTP

In der Zeit des Internets wird es zunehmend wichtiger, Dateien auch auf andere Rechner zu übertragen, die nicht im eigenen Netzwerk eingebunden sind. Das kann ein Firmenrechner sein, auf den über eine Internetverbindung Kundendaten mit dem FTP-Protokoll übertragen oder von dem neue Aufträge abgeholt werden müssen. Aber auch im privaten Bereich wird man immer öfter mit der Datenübertragung von und zu einem fremden Rechner konfrontiert, selbst wenn es nur darum geht, die eigene Homepage zu aktualisieren.

Um beispielsweise aus Excel heraus eine Kopie von sich selbst zu einem FTP-Server zu schicken, könnten Sie die Methode `ActiveWorkbook.SaveCopyAs` benutzen, um die aktuelle Mappe zu speichern und diese anschließend mithilfe der vorliegenden Klasse auf den Server zu kopieren. Auch die Speicherung als HTML und die anschließende Übertragung zum eigenen Webspace wäre damit ohne Probleme möglich.

Im Internet wird zur Dateiübertragung am häufigsten das *FTP*-Protokoll benutzt. Das auf *TCP/IP* basierende und somit verbindungsorientierte *File Transfer Protocol* wird in den RFCs 959 und 765 beschrieben. Die *RFC* (`Request for Comments`) sind Dokumente, die Internetstandards beschreiben. Glücklicherweise müssen Sie nicht unbedingt das Protokoll kennen, um es zu benutzen. Es kann aber auf der anderen Seite auch nicht schaden, wenn Sie sich trotzdem damit auseinander setzen.

Die Sache mit der Dateiübertragung via FTP funktioniert im Prinzip folgendermaßen:

Auf dem Remoterechner läuft ein FTP-Server und wartet (meistens) auf Port 21 auf Anfragen. Dieser Server ist nichts anderes als ein gestartetes Programm, dass auf dem Remoterechner die eigentliche Arbeit übernimmt. Mit einem Benutzernamen und dem zugehörigen Passwort können Sie sich an diesen Server anmelden. Dort haben Sie, je nach vergebener Berechtigung, die Möglichkeit, Verzeichnisse anzulegen, Verzeichnisse und Dateien zu löschen oder umzubenennen und Dateien von und zu dem Remoterechner zu übertragen.

Ein FTP-Client, mit dessen Hilfe man sich an einen FTP-Server anmelden und die Dateiübertragung steuern kann, sollte eigentlich auf jedem Rechner standardmäßig vorhanden sein. Geben Sie einfach mal in der Kommandozeile FTP ein.

Wollen Sie die Übertragung automatisieren, ist es bei den meisten FTP-Clients möglich, sich eine Textdatei mit den Kommandos anzulegen, die nacheinander ausgeführt werden sollen. Darunter fällt auch die Anmeldung an den Server. Den Pfad zu dieser Datei können Sie beim Aufrufen des Clients an der Kommandozeile mit übergeben.

Damit lässt es sich schon arbeiten. Sie können die Textdatei vor dem Aufruf programmgesteuert manipulieren und sind somit auch einigermaßen flexibel. Mit dem `Shell`-Befehl und dem gleichzeitig übergebenen Pfad zur Textdatei wird der FTP-Client dann gestartet. Die Kommandos sollten nun nacheinander

abgearbeitet werden. Sie geben aber damit die Kontrolle über den eigentlichen Vorgang vollkommen aus der Hand. Zurück bleibt immer ein mulmiges Gefühl.

Es gibt zum Glück aber ein paar API-Funktionen, die das Gleiche machen. Wenn Sie diese in eine Klasse packen und mit ein paar Hüllroutinen versehen, haben Sie alles, was Sie für die Datenübertragung brauchen. Das Schöne dabei ist, dass Sie die Kontrolle über die Übertragung nicht mehr ganz aus der Hand geben.

Die vorliegende Klasse bietet zudem noch die Möglichkeit, den Quelltext von Internetseiten zu liefern. Wer zum Beispiel Börsenkurse sucht, kann sich den Quelltext herunterladen und an der entsprechenden Textstelle die Kurse auslesen.

14.4.1 Die Klasse clsInternet

Listing 14.5
Beispiele\14_Netzwerk-Internet\14_03_Internet.xls\clsInternet

```
Option Explicit
Private Const FTP_TRANSFER_TYPE_UNKNOWN = &H0
Private Const FTP_TRANSFER_TYPE_ASCII = &H1
Private Const FTP_TRANSFER_TYPE_BINARY = &H2
Private Const INTERNET_DEFAULT_FTP_PORT = 21
Private Const INTERNET_SERVICE_FTP = 1
Private Const INTERNET_FLAG_PASSIVE = &H8000000
Private Const INTERNET_FLAG_RELOAD = &H80000000
Private Const INTERNET_OPEN_TYPE_PRECONFIG = 0
Private Const INTERNET_OPEN_TYPE_DIRECT = 1
Private Const INTERNET_OPEN_TYPE_PROXY = 3
Private Const INTERNET_OPEN_TYPE_PRECONFIG_WITH_NO_AUTOPROXY = 4
Private Const INTERNET_FLAG_EXISTING_CONNECT = &H20000000
Private Const FILE_ATTRIBUTE_DIRECTORY = &H10
Private Const FILE_ATTRIBUTE_ARCHIVE = &H20
Private Const FILE_ATTRIBUTE_COMPRESSED = &H800
Private Const FILE_ATTRIBUTE_HIDDEN = &H2
Private Const FILE_ATTRIBUTE_NORMAL = &H80
Private Const FILE_ATTRIBUTE_READONLY = &H1
Private Const FILE_ATTRIBUTE_SYSTEM = &H4
Private Const MAX_PATH = 260
Private Const PassiveConnection As Boolean = True

Private Type FILETIME
    dwLowDateTime As Long
    dwHighDateTime As Long
End Type

Private Type SYSTEMTIME
    wYear As Integer
    wMonth As Integer
    wDayOfWeek As Integer
    wDay As Integer
    wHour As Integer
    wMinute As Integer
    wSecond As Integer
    wMilliseconds As Integer
End Type
```

Listing 14.5 (Forts.)
Beispiele\14_Netzwerk-Internet\14_03_Internet.xls\clsInternet

```
Private Type WIN32_FIND_DATA
    dwFileAttributes As Long
    ftCreationTime As FILETIME
    ftLastAccessTime As FILETIME
    ftLastWriteTime As FILETIME
    nFileSizeHigh As Long
    nFileSizeLow As Long
    dwReserved0 As Long
    dwReserved1 As Long
    cFileName As String * MAX_PATH
    cAlternate As String * 14
End Type

Private Declare Function InternetReadFile _
   Lib "wininet" ( _
   ByVal hFile As Long, _
   ByVal sBuffer As String, _
   ByVal lNumBytesToRead As Long, _
   lNumberOfBytesRead As Long _
   ) As Integer

Private Declare Function InternetOpenUrl _
   Lib "wininet" Alias "InternetOpenUrlA" ( _
   ByVal hInternetSession As Long, _
   ByVal lpszUrl As String, _
   ByVal lpszHeaders As String, _
   ByVal dwHeadersLength As Long, _
   ByVal dwFlags As Long, _
   ByVal dwContext As Long _
   ) As Long

Private Declare Function InternetCloseHandle _
   Lib "wininet" ( _
   ByRef hInet As Long _
   ) As Long

Private Declare Function InternetConnect _
   Lib "wininet.dll" Alias "InternetConnectA" ( _
   ByVal hInternetSession As Long, _
   ByVal sServerName As String, _
   ByVal nServerPort As Integer, _
   ByVal sUserName As String, _
   ByVal sPassword As String, _
   ByVal lService As Long, _
   ByVal lFlags As Long, _
   ByVal lContext As Long _
   ) As Long

Private Declare Function InternetOpen _
   Lib "wininet.dll" Alias "InternetOpenA" ( _
   ByVal sAgent As String, _
   ByVal lAccessType As Long, _
   ByVal sProxyName As String, _
   ByVal sProxyBypass As String, _
   ByVal lFlags As Long _
   ) As Long
```

Listing 14.5 (Forts.)
Beispiele\14_Netzwerk-Internet\14_03_Internet.xls\clsInternet

```
Private Declare Function FtpSetCurrentDirectory _
   Lib "wininet.dll" Alias "FtpSetCurrentDirectoryA" ( _
   ByVal hFtpSession As Long, _
   ByVal lpszDirectory As String _
   ) As Boolean

Private Declare Function FtpGetCurrentDirectory _
   Lib "wininet.dll" Alias "FtpGetCurrentDirectoryA" ( _
   ByVal hFtpSession As Long, _
   ByVal lpszCurrentDirectory As String, _
   lpdwCurrentDirectory As Long _
   ) As Long

Private Declare Function FtpCreateDirectory _
   Lib "wininet.dll" Alias "FtpCreateDirectoryA" ( _
   ByVal hFtpSession As Long, _
   ByVal lpszDirectory As String _
   ) As Boolean

Private Declare Function FtpRemoveDirectory _
   Lib "wininet.dll" Alias "FtpRemoveDirectoryA" ( _
   ByVal hFtpSession As Long, _
   ByVal lpszDirectory As String _
   ) As Boolean

Private Declare Function FtpDeleteFile _
   Lib "wininet.dll" Alias "FtpDeleteFileA" ( _
   ByVal hFtpSession As Long, _
   ByVal lpszFileName As String _
   ) As Boolean

Private Declare Function FtpRenameFile _
   Lib "wininet.dll" Alias "FtpRenameFileA" ( _
   ByVal hFtpSession As Long, _
   ByVal lpszExisting As String, _
   ByVal lpszNew As String _
   ) As Boolean

Private Declare Function FtpGetFile _
   Lib "wininet.dll" Alias "FtpGetFileA" ( _
   ByVal hConnect As Long, _
   ByVal lpszRemoteFile As String, _
   ByVal lpszNewFile As String, _
   ByVal fFailIfExists As Long, _
   ByVal dwFlagsAndAttributes As Long, _
   ByVal dwFlags As Long, _
   ByRef dwContext As Long _
   ) As Boolean

Private Declare Function FtpPutFile _
   Lib "wininet.dll" Alias "FtpPutFileA" ( _
   ByVal hConnect As Long, _
   ByVal lpszLocalFile As String, _
   ByVal lpszNewRemoteFile As String, _
   ByVal dwFlags As Long, _
   ByVal dwContext As Long _
   ) As Boolean
```

Listing 14.5 (Forts.)
Beispiele\14_Netzwerk-Internet\14_03_Internet.xls\clsInternet

```
Private Declare Function InternetGetLastResponseInfo _
   Lib "wininet.dll" Alias "InternetGetLastResponseInfoA" ( _
   lpdwError As Long, _
   ByVal lpszBuffer As String, _
   lpdwBufferLength As Long _
   ) As Boolean

Private Declare Function FtpFindFirstFile _
   Lib "wininet.dll" Alias "FtpFindFirstFileA" ( _
   ByVal hFtpSession As Long, _
   ByVal lpszSearchFile As String, _
   lpFindFileData As WIN32_FIND_DATA, _
   ByVal dwFlags As Long, _
   ByVal dwContent As Long _
   ) As Long

Private Declare Function InternetFindNextFile _
   Lib "wininet.dll" Alias "InternetFindNextFileA" ( _
   ByVal hFind As Long, _
   lpvFindData As WIN32_FIND_DATA _
   ) As Long

Private Declare Function FileTimeToSystemTime _
   Lib "kernel32" ( _
   lpFileTime As FILETIME, _
   lpSystemTime As SYSTEMTIME _
   ) As Long

Private Declare Function FileTimeToLocalFileTime _
   Lib "kernel32" ( _
   lpFileTime As FILETIME, _
   lpLocalFileTime As FILETIME _
   ) As Long

Private mstrFTP         As String
Private mstrUser        As String
Private mstrPass        As String
Private mlngFTPHandle   As Long
Private mstrPath        As String
Private mstrProxy       As String
Private mlngNetOpen     As Long
Private mstrAgent       As String
Private mstrLastError   As String

Public Function LocalToFtp( _
   strFileLocal As String, _
   strFileRemote As String _
   ) As Boolean

   ' Kein FTP-Handle, verlassen
   If mlngFTPHandle = 0 Then Exit Function

   ' Datei ins Remoteverzeichnis kopieren
   LocalToFtp = FtpPutFile(mlngFTPHandle, strFileLocal, _
      strFileRemote, FTP_TRANSFER_TYPE_UNKNOWN, 0)

End Function
```

Listing 14.5 (Forts.)
Beispiele\14_Netzwerk-Internet\14_03_Internet.xls\clsInternet

```
Public Function FtpToLocal( _
   strFileRemote As String, _
   strFileLocal As String, _
   Optional blnReplace As Boolean _
   ) As Boolean

   blnReplace = Not (blnReplace)

   ' Kein FTP-Handle, verlassen
   If mlngFTPHandle = 0 Then Exit Function

   ' Datei vom Remoteverzeichnis kopieren
   FtpToLocal = FtpGetFile( _
      mlngFTPHandle, strFileRemote, _
      strFileLocal, blnReplace, _
      FILE_ATTRIBUTE_NORMAL, _
      FTP_TRANSFER_TYPE_UNKNOWN, 0)

End Function

Public Function GetCurDir() As String
   Dim blnOk As Boolean

   ' Kein FTP-Handle, verlassen
   If mlngFTPHandle = 0 Then Exit Function

   ' Aktuelles Verzeichnis liefern
   mstrPath = String(MAX_PATH, 0)

   ' Aktuelles Verzeichnis abfragen
   blnOk = FtpGetCurrentDirectory( _
      mlngFTPHandle, mstrPath, Len(mstrPath))

   If blnOk Then GetCurDir = ApiStringTrim(mstrPath)

End Function

Public Function ChangeDir(strFolder As String) As Boolean
   Dim blnOk As Boolean

   ' Kein FTP-Handle, verlassen
   If mlngFTPHandle = 0 Then Exit Function

   ' Verzeichnis wechseln
   If FtpSetCurrentDirectory(mlngFTPHandle, strFolder) Then

      ChangeDir = True

      mstrPath = String(MAX_PATH, 0)

      ' Aktuelles Verzeichnis abfragen
      blnOk = FtpGetCurrentDirectory( _
         mlngFTPHandle, mstrPath, Len(mstrPath))

      If blnOk Then mstrPath = ApiStringTrim(mstrPath)

   End If
End Function
```

Listing 14.5 (Forts.)
Beispiele\14_Netzwerk-Internet\
14_03_Internet.xls\clsInternet

```
Public Function CreateDir(strFolder As String) As Boolean

   ' Kein FTP-Handle, verlassen
   If mlngFTPHandle = 0 Then Exit Function

   ' Verzeichnis erzeugen
   If FtpCreateDirectory(mlngFTPHandle, strFolder) Then _
      CreateDir = True
End Function

Public Function DeleteDir(strFolder As String) As Boolean

   ' Kein FTP-Handle, verlassen
   If mlngFTPHandle = 0 Then Exit Function

   ' Verzeichnis löschen
   If FtpRemoveDirectory(mlngFTPHandle, strFolder) Then _
      DeleteDir = True
End Function

Public Function DeleteFile(strFile As String) As Boolean

   ' Kein FTP-Handle, verlassen
   If mlngFTPHandle = 0 Then Exit Function

   ' Datei löschen
   If FtpDeleteFile(mlngFTPHandle, strFile) Then _
      DeleteFile = True

End Function

Public Function RenameFile( _
   strFileNameOld As String, _
   strFileNameNew As String) _
   As Boolean

   ' Kein FTP-Handle, verlassen
   If mlngFTPHandle = 0 Then Exit Function

   ' Datei umbenennen
   If FtpRenameFile(mlngFTPHandle, strFileNameOld, _
      strFileNameNew) Then RenameFile = True

End Function

Public Function CreateFileList() As Collection
   Dim strFileName      As String
   Dim FF               As Long
   Dim astrData(1 To 4) As String
   Dim colFile          As New Collection
   Dim udtFindData      As WIN32_FIND_DATA

   ' Kein FTP-Handle, verlassen
   If mlngFTPHandle = 0 Then Exit Function
```

Listing 14.5 (Forts.)
Beispiele\14_Netzwerk-Internet\14_03_Internet.xls\clsInternet

```
    ' Dateiliste erzeugen
    With udtFindData

        ' Suchhandle holen
        FF = FtpFindFirstFile( _
            mlngFTPHandle, "*", udtFindData, 0, 0)

        Do While FF <> 0

            ' Dateiname aus Buffer extrahieren
            strFileName = ApiStringTrim(.cFileName)

            ' Diese Verzeichnisse brauchen wir nicht
            If (strFileName <> ".") And (strFileName <> "..") Then

                astrData(1) = strFileName

                ' Überprüfen, ob Verzeichnis
                If (.dwFileAttributes And FILE_ATTRIBUTE_DIRECTORY) _
                    = FILE_ATTRIBUTE_DIRECTORY Then

                    astrData(2) = "Directory"

                Else

                    astrData(2) = "Datei"

                End If

                ' Dateigröße
                astrData(3) = .nFileSizeLow

                ' Letzter Zugriff
                astrData(4) = Format(ChangeTime(.ftLastWriteTime), _
                    "DD.MM.YYYY hh:nn:ss")

                ' Zur Collection hinzufügen
                colFile.Add astrData, mstrPath & "/" & strFileName

            End If

            ' Nächste Datei holen
            If InternetFindNextFile(FF, udtFindData) = False _
                Then Exit Do
        Loop
        ' Suchhandle schließen
        InternetCloseHandle FF
    End With
    Set CreateFileList = colFile
End Function
```

Listing 14.5 (Forts.)
Beispiele\14_Netzwerk-Internet\14_03_Internet.xls\clsInternet

```
Public Function UrlRead(strSource As String) As String
   Dim lngInetFile         As Long
   Dim strInetDummy        As String
   Dim lngInetBytesCount   As Long
   Dim lInetBytesWant      As Long

   ' Internetverbindung herstellen
   OpenInternet

   ' Handle zur URL holen
   lngInetFile = InternetOpenUrl(mlngNetOpen, strSource, _
      vbNullString, ByVal 0&, INTERNET_FLAG_RELOAD, ByVal 0&)

   If lngInetFile <> 0 Then ' Zugriff auf URL möglich

      Do

         'Buffer erzeugen
         lInetBytesWant = lInetBytesWant + 100000
         strInetDummy = String(lInetBytesWant, 0)

         ' Daten holen
         InternetReadFile lngInetFile, strInetDummy, _
            lInetBytesWant, lngInetBytesCount

         ' Wenn Buffer zu klein, dann Buffer größer machen
         ' und erneut lesen
         If lInetBytesWant <= lngInetBytesCount Then

            'Handle schließen
            InternetCloseHandle lngInetFile

            ' Handle zur URL holen
            lngInetFile = InternetOpenUrl(mlngNetOpen, strSource, _
               vbNullString, ByVal 0&, INTERNET_FLAG_RELOAD,_
               ByVal 0&)

         End If

      Loop While lInetBytesWant <= lngInetBytesCount

       ' Internethandle schließen
       InternetCloseHandle lngInetFile
       strSource = Left$(strInetDummy, lngInetBytesCount)

   Else

       InetError

   End If

   UrlRead = strSource

   ' Verbindung schließen
   InternetCloseHandle mlngNetOpen

End Function
```

Listing 14.5 (Forts.)
Beispiele\14_Netzwerk-Internet\14_03_Internet.xls\clsInternet

```
Public Property Get LastError() As String
   LastError = mstrLastError
   mstrLastError = ""
End Property

Public Property Get Agent() As String
   Agent = mstrAgent
End Property
Public Property Let Agent(ByVal vNewValue As String)
   mstrAgent = vNewValue
End Property

Public Property Get Proxy() As String
   Proxy = mstrProxy
End Property
Public Property Let Proxy(ByVal vNewValue As String)
   mstrProxy = vNewValue
End Property

Public Property Get FTPServer() As String
   FTPServer = mstrFTP
End Property
Public Property Let FTPServer(ByVal vNewValue As String)
   mstrFTP = vNewValue
End Property

Public Property Let Password(ByVal vNewValue As String)
   mstrPass = vNewValue
End Property

Public Property Get Username() As String
   Proxy = mstrUser
End Property
Public Property Let Username(ByVal vNewValue As String)
   mstrUser = vNewValue
End Property

Public Function CloseConnectionToServer() As Boolean

   If mlngFTPHandle = 0 Then Exit Function

   ' Internetverbindung beenden
   InternetCloseHandle mlngFTPHandle

   mlngFTPHandle = 0

   ' Ergebnis zurückgeben
   CloseConnectionToServer = True

End Function

Public Function OpenConnectionToServer() As Boolean

   'Eventuell offene Verbindungen schließen
   InternetCloseHandle mlngFTPHandle
   InternetCloseHandle mlngNetOpen
   mlngFTPHandle = 0
```

Listing 14.5 (Forts.)
Beispiele\14_Netzwerk-Internet\14_03_Internet.xls\clsInternet

```
    'Internetverbindung herstellen
    OpenInternet
    If mlngNetOpen = 0 Then Exit Function

    'Mit FTP-Server verbinden
    mlngFTPHandle = InternetConnect(mlngNetOpen, _
       mstrFTP, INTERNET_DEFAULT_FTP_PORT, _
       mstrUser, mstrPass, INTERNET_SERVICE_FTP, _
       INTERNET_FLAG_PASSIVE Or _
       INTERNET_FLAG_EXISTING_CONNECT, 0)

    If mlngFTPHandle = 0 Then InetError: Exit Function

    OpenConnectionToServer = True

End Function

Private Sub OpenInternet()

    'Verbindung ins Inet herstellen
    If mstrProxy = "" Then

        'Direkt, ohne Proxy
        mlngNetOpen = InternetOpen(mstrAgent, _
           INTERNET_OPEN_TYPE_DIRECT, vbNullString, vbNullString, 0)

    Else

        'Über Proxy
        mlngNetOpen = InternetOpen(mstrAgent, _
            INTERNET_OPEN_TYPE_PROXY, mstrProxy, vbNullString, 0)

    End If

    'Fehlertext vom letzten Fehler holen
    If mlngNetOpen = 0 Then InetError

End Sub

Private Function InetError() As String
    Dim lngErrNumber As Long
    Dim strErrString As String
    Dim lngBufferLen As Long

    'Notwendige Bufferlänge holen
    InternetGetLastResponseInfo lngErrNumber, _
       strErrString, lngBufferLen

    'Buffer erzeugen
    strErrString = String(lngBufferLen, 0)

    'Fehlertext holen
    InternetGetLastResponseInfo lngErrNumber, _
       strErrString, lngBufferLen

    'Rückgabewert der Funktion
    InetError = strErrString
```

Listing 14.5 (Forts.)
Beispiele\14_Netzwerk-Internet\14_03_Internet.xls\clsInternet

```
    'Interne Variable
    If strErrString <> "" Then mstrLastError = strErrString

End Function

Private Function ApiStringTrim(sApiNullString As String) As String
    ApiStringTrim = Mid$(sApiNullString, 1, _
       InStr(sApiNullString, Chr(0)) - 1)
End Function

Private Function ChangeTime(udtFiletime As FILETIME) As Date
    Dim tSysTime As SYSTEMTIME

    FileTimeToSystemTime udtFiletime, tSysTime

    If tSysTime.wYear >= 1900 Then

       ChangeTime = CDbl(DateSerial(tSysTime.wYear, _
          tSysTime.wMonth, tSysTime.wDay) + _
          TimeSerial(tSysTime.wHour, tSysTime.wMinute, _
          tSysTime.wSecond))

    Else

       ChangeTime = 0

    End If
End Function

Private Sub Class_Initialize()
    mstrAgent = "Schwimmer"
    mstrPath = "/"
End Sub

Private Sub Class_Terminate()

    'Eventuell offene Verbindungen schließen
    InternetCloseHandle mlngFTPHandle
    InternetCloseHandle mlngNetOpen

End Sub
```

LocalToFtp

Beim Aufruf dieser Funktion wird eine Datei als Kopie auf den Remoterechner übertragen.

Der Parameter `strFileLocal` enthält den Pfad zu einer Datei auf dem lokalen Rechner, die kopiert werden soll. Dabei ist es egal, ob ein Schrägstrich (/) oder ein Backslash (\) als Verzeichnistrennzeichen verwendet wird.

Der Parameter `strFileRemote` enthält den Pfad inklusive Zielnamen, unter dem die Datei auf dem Remoterechner gespeichert werden soll. Dabei ist es egal, ob ein Schrägstrich (/) oder ein Backslash (\) als Verzeichnistrennzeichen verwendet wird.

Zur Datenübertragung muss mit der API-Funktion `InternetConnect` eine Verbindung zum FTP-Server aufgebaut sein. Die Datenübertragung auf den Remoterechner wird mit der API-Funktion `FtpPutFile` angestoßen.

Als Funktionsergebnis liefert die Funktion `LocalToFtp` den Wahrheitswert `True` zurück, wenn die Aktion erfolgreich war.

FtpToLocal

Beim Aufruf dieser Funktion wird eine Datei als Kopie vom Remoterechner auf den lokalen Rechner übertragen.

Der Parameter `strFileRemote` enthält den Pfad zu einer Datei auf dem Remoterechner, die kopiert werden soll. Dabei ist es egal, ob ein Schrägstrich (/) oder ein Backslash (\) als Verzeichnistrennzeichen verwendet wird.

Der Parameter `strFileLocal` enthält den Pfad inklusive Zielnamen, unter dem die Datei auf dem lokalen Rechner gespeichert werden soll. Dabei ist es egal, ob ein Schrägstrich (/) oder ein Backslash (\) als Verzeichnistrennzeichen verwendet wird.

Der optionale Parameter `blnReplace` gibt an, ob eine eventuell vorhandene Datei mit gleichem Namen überschrieben werden soll.

Zur Datenübertragung muss mit der API-Funktion `InternetConnect` eine Verbindung zum FTP-Server aufgebaut sein. Die Datenübertragung auf den lokalen Rechner wird mit der API-Funktion `FtpGetFile` angestoßen.

Als Funktionsergebnis liefert die Funktion `FtpToLocal` den Wahrheitswert `True` zurück, wenn die Aktion erfolgreich war.

GetCurDir

Beim Aufruf dieser Funktion wird das aktuelle Verzeichnis des Remoterechners zurückgeliefert. Die API-Funktion `FtpGetCurrentDirectory` leistet dabei die eigentliche Arbeit.

ChangeDir

Beim Aufruf dieser Funktion wird das aktuelle Verzeichnis des Remoterechners geändert.

Der Parameter `strFolder` ist ein String mit dem Verzeichnis, in das gewechselt werden soll. Es kann ein vollständiger Pfad oder ein Pfad relativ zum aktuellen Verzeichnis angegeben werden. Dabei ist es egal, ob ein Schrägstrich (/) oder ein Backslash (\) als Verzeichnistrennzeichen verwendet wird.

Die API-Funktion `FtpSetCurrentDirectory` leistet die eigentliche Arbeit.

Zur Funktionsausführung muss mit der API-Funktion `InternetConnect` eine Verbindung zum FTP-Server aufgebaut sein.

Als Funktionsergebnis liefert die Funktion `ChangeDir` den Wahrheitswert `True` zurück, wenn die Aktion erfolgreich war.

CreateDir

Beim Aufruf dieser Funktion wird ein neues Verzeichnis auf dem Remoterechners angelegt.

Der Parameter `strFolder` ist ein String mit dem Verzeichnis, welches erzeugt werden soll. Es kann ein vollständiger Pfad oder ein Pfad relativ zum aktuellen Verzeichnis angegeben werden. Dabei ist es egal, ob ein Schrägstrich (/) oder ein Backslash (\) als Verzeichnistrennzeichen verwendet wird.

Die API-Funktion `FtpCreateDirectory` leistet die eigentliche Arbeit.

Zur Funktionsausführung muss mit der API-Funktion `InternetConnect` eine Verbindung zum FTP-Server aufgebaut sein.

Als Funktionsergebnis liefert die Funktion `CreateDir` den Wahrheitswert `True` zurück, wenn die Aktion erfolgreich war.

DeleteDir

Beim Aufruf dieser Funktion wird ein Verzeichnis auf dem Remoterechner gelöscht.

Der Parameter `strFolder` ist ein String mit dem Verzeichnis, welches gelöscht werden soll. Es kann ein vollständiger Pfad oder ein Pfad relativ zum aktuellen Verzeichnis angegeben werden. Dabei ist es egal, ob ein Schrägstrich (/) oder ein Backslash (\) als Verzeichnistrennzeichen verwendet wird.

Die API-Funktion `FtpRemoveDirectory` leistet die eigentliche Arbeit.

Zur Funktionsausführung muss mit der API-Funktion `InternetConnect` eine Verbindung zum FTP-Server aufgebaut sein.

Als Funktionsergebnis liefert die Funktion `DeleteDir` den Wahrheitswert `True` zurück, wenn die Aktion erfolgreich war.

DeleteFile

Beim Aufruf dieser Funktion wird eine Datei auf dem Remoterechner gelöscht.

Der Parameter `strFile` ist ein String mit der Datei, welche gelöscht werden soll. Es kann ein vollständiger Pfad oder ein Pfad relativ zum aktuellen Verzeichnis angegeben werden. Dabei ist es egal, ob ein Schrägstrich (/) oder ein Backslash (\) als Verzeichnistrennzeichen verwendet wird.

Die API-Funktion `FtpDeleteFile` leistet die eigentliche Arbeit.

Zur Funktionsausführung muss mit der API-Funktion `InternetConnect` eine Verbindung zum FTP-Server aufgebaut sein.

Als Funktionsergebnis liefert die Funktion `DeleteFile` den Wahrheitswert `True` zurück, wenn die Aktion erfolgreich war.

RenameFile

Beim Aufruf dieser Funktion wird eine Datei auf dem Remoterechner umbenannt.

Der Parameter `strFileNameOld` ist ein String mit der Datei, welche umbenannt werden soll. Es kann ein vollständiger Pfad oder ein Pfad relativ zum aktuellen Verzeichnis angegeben werden. Dabei ist es egal, ob ein Schrägstrich (/) oder ein Backslash (\) als Verzeichnistrennzeichen verwendet wird.

Der Parameter `strFileNameNew` ist ein String mit dem Dateinamen, in welche die Datei umbenannt werden soll. Es kann ein vollständiger Pfad oder ein Pfad relativ zum aktuellen Verzeichnis angegeben werden. Dabei ist es egal, ob ein Schrägstrich (/) oder ein Backslash (\) als Verzeichnistrennzeichen verwendet wird.

Die API-Funktion `FtpRenameFile` leistet die eigentliche Arbeit.

Zur Funktionsausführung muss mit der API-Funktion `InternetConnect` eine Verbindung zum FTP-Server aufgebaut sein.

Als Funktionsergebnis liefert die Funktion `RenameFile` den Wahrheitswert `True` zurück, wenn die Aktion erfolgreich war.

CreateFileList

Beim Aufruf dieser Funktion wird in einer Collection eine Dateiliste erzeugt.

Als Funktionsergebnis liefert die Funktion `CreateFileList` diese Collection zurück. Das Funktionsergebnis muss zum Auswerten mit der Anweisung `Set` einer Objektvariablen vom Typ `Collection` zugewiesen werden.

Zum Erzeugen der eigentlichen Dateiliste werden die API-Funktionen `FtpFindFirstFile` und `InternetFindNextFile` benutzt.

Die Funktion `FtpFindFirstFile` benötigt einen nullterminierten String, der einem gültigen Pfad auf dem Remoterechner entspricht. Der Dateiname in diesem String kann Wildcards wie * und ? enthalten. Wird eine Datei gefunden, die diesem Muster entspricht, wird eine Struktur vom Typ `Win32_Find_Data` ausgefüllt, welche Informationen zu der gefundenen Datei liefert. Das Funktionsergebnis ist ein gültiges `Suchhandle`, das Sie der Funktion `InternetFindNextFile` übergeben können. Damit wird die nächste Datei gesucht, die dem gleichen Suchmuster entspricht.

Ein Array mit vier Elementen, welches einer Collection als Element übergeben wird, enthält die Infos der Datei. Das erste Element des Arrays nimmt den Dateinamen auf, das zweite die Information, ob es sich um eine Datei oder ein Directory handelt. Das dritte Element nimmt die Dateigröße und das vierte den Zeitpunkt der letzten Änderung auf.

URLRead

Beim Aufruf dieser Funktion wird der Inhalt einer Internetseite geliefert.

Der Parameter `strSource` ist ein String mit der URL, deren Inhalt geliefert werden soll.

Mit der Funktion `OpenInternet` wird zu Beginn ein `Socket` für den laufenden Prozess initialisiert. Die Funktion `InternetOpenUrl` liefert ein Handle auf die angeforderte Datei. Danach wird ein Puffer erzeugt und dieser mit dem Inhalt der Datei gefüllt.

Die eingesetzte Funktion `InternetReadFile` liefert den Inhalt, und zwar maximal so viele Zeichen, wie im Parameter `lNumBytesToRead` angegeben. Da auch über den Parameter `lNumberOfBytesRead` die Anzahl der tatsächlich in den Puffer geschriebenen Zeichen zurückgegeben wird, vergrößern Sie den Puffer und den Parameter `lNumBytesToRead` so lange, bis der Puffer größer ist als die Anzahl der zurückgelieferten Zeichen. Erst dann ist die Datei komplett eingelesen.

LastError

Beim Aufruf dieser Funktion wird der zuletzt aufgetretene Fehler zurückgegeben. Es wird dazu die interne Variable `mstrLastError` ausgelesen.

Agent

Die Eigenschaft `Agent` gibt den Namen der Anwendung an, welche die Internetfunktionen benutzt.

Beim Internet Explorer würde beispielsweise »Microsoft Internet Explorer« passen. Es wird die interne Variable `mstrAgent` gesetzt.

Proxy

Die Eigenschaft `Proxy` gibt den Namen oder die IP des vorgeschalteten Proxyservers an.

Es wird die interne Variable `mstrProxy` gesetzt.

FTPServer

Die Eigenschaft `FTPServer` gibt den Namen oder die IP des FTP-Servers an.

Es wird die interne Variable `mstrFtp` gesetzt.

Password

Die Eigenschaft `Password` enthält das Passwort des Users zur Anmeldung an den FTP-Server.

Es wird die interne Variable `mstrPass` gesetzt.

Username

Die Eigenschaft `Username` enthält den Benutzernamen zur Anmeldung an den FTP-Server.

Es wird die interne Variable `mstrUser` gesetzt.

CloseConnectionToServer

Beim Aufruf dieser Funktion wird die Verbindung mit dem FTP-Server beendet, sofern überhaupt eine bestanden hat.

Um die Session zu beenden, wird die API-Funktion `InternetCloseHandle` benutzt.

Als Funktionsergebnis liefert die Funktion `CloseConnectionToServer` den Wahrheitswert `True` zurück, wenn die Aktion erfolgreich war.

OpenConnectionToServer

Beim Aufruf dieser Funktion wird eine Verbindung mit dem FTP-Server hergestellt.

Eine bestehende Session wird vorher beendet. Mit der Funktion `OpenInternet` wird zu Beginn ein `Socket` für den laufenden Prozess initialisiert. Die API-Funktion `InternetConnect` stellt dann eine Verbindung mit dem FTP-Server her.

Als Funktionsergebnis liefert die Funktion `OpenConnectionToServer` den Wahrheitswert `True` zurück, wenn die Aktion erfolgreich war.

OpenInternet

Beim Aufruf dieser Prozedur wird ein `Socket` für den laufenden Prozess initialisiert.

Wenn die Variable `mstrProxy` einen Leerstring enthält, wird die API-Funktion `InternetOpen` mit dem Flag `INTERNET_OPEN_TYPE_DIRECT` und dem auf `vbNullString` gesetzten Parameter `sProxyBypass` aufgerufen. Andernfalls mit dem Flag `INTERNET_OPEN_TYPE_PROXY_DIRECT` und dem auf `mstrProxy` gesetzten Parameter `sProxyBypass`.

InetError

Beim Aufruf dieser Funktion wird die interne Variable `mstrLastError` mit dem Fehlertext des letzten Fehlers gesetzt.

Dazu wird die API-Funktion `InternetGetLastResponseInfo` benutzt.

Als Funktionsergebnis liefert die Funktion `InetError` den Fehlertext des letzten Fehlers zurück.

ApiStringTrim

Diese Funktion kürzt einen String beim ersten Auftreten eines Zeichens mit dem ASCII-Code null und liefert diesen String als Funktionsergebnis zurück.

ChangeTime

Diese Funktion wandelt eine Struktur vom Typ `FileTime` in ein gültiges Datum um.

Dazu werden die API-Funktion `FileTimeToSystemTime` und die VBA-Funktionen `DateSerial` und `TimeSerial` benutzt.

Class_Terminate

In dieser Ereignisprozedur werden eventuell offene Verbindungen geschlossen.

Class_Initialize

In dieser Ereignisprozedur werden verschiedene Parameter initialisiert.

14.4.2 FTP-Übertragung

In der beiliegenden Arbeitsmappe können Sie die Parameter wie Remoterechner, Proxy, Passwort, User und noch einiges mehr aus einem Tabellenblatt heraus setzen. Es gibt im Internet auch ein paar freie FTP-Server, die meistens von Universitäten betrieben werden. Noch besser ist es aber, wenn Sie sich zum Testen einen FTP-Server auf Ihren Rechner installieren. Es gibt mittlerweile sehr gute Freeware-Programme.

Um die Klasse `clsInternet` in Bezug auf das FTP-Protokoll zu testen, können Sie folgenden Code benutzen:

Listing 14.6
Beispiele\14_Netzwerk-Internet\14_03_Internet.xls\mdlKlasseInternet

```
Sub Klassentest()
   Dim objInet As New clsInternet
   Dim colDummy As Collection
   Dim varItem As Variant
   Dim strFile As String

   On Error GoTo fehlerbehandlung

   With objInet

      ' Den Agent setzen (optional)
      .Agent = "Michael Schwimmer"

      ' FTP-Server auf dem eigenen Rechner
      .FTPServer = "laptop"

      ' Den Usernamen setzen. Hier einen nicht existierenden, um
      ' Fehlermeldungen zu testen
      .Username = "Anonymous123"

      ' Passwort
      .Password = ""

      ' Versuch, eine Verbindung herzustellen
      .OpenConnectionToServer

      ' Fehlermeldung ausgeben
      MsgBox .LastError, , "Fehlermeldung"

      ' Connection schließen
      .CloseConnectionToServer

      ' Richtigen Usernamen benutzen
      .Username = "Anonymous"
```

Listing 14.6 (Forts.)
Beispiele\14_Netzwerk-Internet\14_03_Internet.xls\mdlKlasseInternet

```
        ' Richtiges Passwort
        .Password = ""

        ' Versuch, eine Verbindung herzustellen
        .OpenConnectionToServer

        ' In ein anderes Verzeichnis wechseln
        .ChangeDir "ftp_test"

        ' Im Remoteverzeichnis ein Verzeichnis anlegen
        .CreateDir "remote"

        ' In dieses Verzeichnis wechseln
        .ChangeDir "remote"

        ' Das aktuelle Verzeichnis anzeigen
        MsgBox .GetCurDir, , _
           "Akt. Verzeichnis"

        ' File ins Remoteverzeichnis übertragen
        .LocalToFtp "c:\ftp_test\local\Blitz1.jpg", _
            "/ftp_test/remote/Blitz2.jpg"

        ' File vom Remoteverzeichnis holen
        .FtpToLocal "/ftp_test/remote/Blitz2.jpg", _
            "c:\ftp_test\local\Blitz3.jpg"

        ' Fileliste des Remoteverzeichnisses holen
        Set colDummy = .CreateFileList

        ' Alles nacheinander anzeigen
        For Each varItem In colDummy
           strFile = varItem(1) & vbCrLf ' strFilename
            strFile = strFile & varItem(2) & vbCrLf ' strFileart
            strFile = strFile & varItem(3) & vbCrLf ' Größe
            strFile = strFile & Format(varItem(4), _
               "DD.MM.YYYY hh:nn:ss") ' Letzter Zugriff
            MsgBox strFile
        Next

        ' File löschen
        .DeleteFile "/ftp_test/remote/Blitz2.jpg"

        ' In das übergeordnete Verzeichnis wechseln
        .ChangeDir ".."
        MsgBox .GetCurDir, , "Akt. Verzeichnis"

        ' Ein Verzeichnis löschen
        .DeleteDir "remote"

        ' Verbindung trennen
        .CloseConnectionToServer

    End With

fehlerbehandlung:
End Sub
```

14.4.3 URL lesen

Eine Internetseite können Sie mit folgendem Code auslesen und als Textdatei im *Temp*-Verzeichnis speichern:

Listing 14.7
Beispiele\14_Netzwerk-Internet\14_03_Internet.xls\mdlKlasseInternet

```
Private Declare Function GetTempPath _
   Lib "kernel32" Alias "GetTempPathA" ( _
   ByVal nBufferLength As Long, _
   ByVal lpBuffer As String _
   ) As Long

Private Sub cmdUrl_Click()
   Dim objInet    As New clsInternet
   Dim colDummy   As Collection
   Dim varItem    As Variant
   Dim strFile    As String
   Dim strSource  As String
   Dim strDest    As String
   Dim strText    As String
   Dim lngFF      As Long
   Dim lngRet     As Long

   On Error GoTo fehlerbehandlung

   ' URL der Internetseite
   strSource = "http://Michael-Schwimmer.de"

   ' Zieldatei
   strDest = String(255, 0)
   lngRet = GetTempPath(255, strDest)
   strDest = Left(strDest, lngRet) & "~~InetTest.htm"

   ' Den Quelltext einer Internetseite lesen
   strText = objInet.UrlRead(strSource)

   lngFF = FreeFile
   If Dir(strDest) <> "" Then
      If MsgBox("Datei existiert, löschen?", vbYesNo, "Speichern") _
         = vbNo Then Exit Sub
      Kill strDest
   End If

   ' Als Datei speichern
   Open strDest For Binary As lngFF
      Put lngFF, , strText
   Close

   Shell "notepad """ & strDest & """"
fehlerbehandlung:
End Sub
```

14.5 Net Send

Unter NT, 2000 und XP können Sie `Net Send` verwenden, um Meldungen an andere Rechner im Netzwerk zu verschicken. Das wird auch eventuell mittels `SHELL` funktionieren, sicherer ist aber die API-Funktion `NetMessageBufferSend`, die exakt das Gleiche macht wie `Net Send`.

Listing 14.8
Beispiele\14_Netzwerk-Internet\
14_04_NetSend.xls\mdlNetSend

```
Private Declare Function NetMessageBufferSend _
   Lib "NETAPI32.DLL" ( _
   yServer As Any, _
   yToName As Byte, _
   yFromName As Any, _
   yMsg As Byte, _
   ByVal lSize As Long _
   ) As Long

Private Const NERR_Success As Long = 0&

Public Function SendMyMessage( _
    strRecipient As String, _
    strSender As String, _
    strBody As String) As Boolean

   'Unter NT, 2000, XP
   Dim abytRecipient() As Byte
   Dim abytSender() As Byte
   Dim abytBody() As Byte

   abytRecipient = strRecipient & vbNullChar
   abytSender = strSender & vbNullChar
   abytBody = strBody & vbNullChar

   SendMyMessage = NetMessageBufferSend( _
       ByVal 0&, _
       abytRecipient(0), _
       abytSender(0), _
       abytBody(0), _
       UBound(abytBody) _
       ) = NERR_Success

End Function
```

15

Sonstiges

15.1 Was Sie in diesem Kapitel erwartet

In diesem Kapitel bekommen Sie einiges vorgestellt, was vom Thema her nicht in die anderen Kapitel hineinpasst, aber nicht so umfangreich ist, um ein eigenes anzulegen.

Dazu gehört als Thema die Umwandlung vom ANSI- in den ASCII-Zeichensatz und umgekehrt.

Ein anderes Thema ist beispielsweise die programmgesteuerte Änderung des aktuellen Druckers. Das ist gar nicht so einfach, denn es müssen einige unerwartete Hürden überwunden werden.

Ein weiteres Beispiel befasst sich mit dem Anlegen von Beschreibungen benutzerdefinierter Funktionen, die auch im Funktionsassistent angezeigt werden.

Wie ein anderer Abschnitt in diesem Kapitel zeigt, können Sie zu einem gegebenen Benutzernamen einen eindeutigen Freischaltcode erzeugen, der auf jedem Rechner gleich ist.

Auch die Erzeugung einer im Universum einmaligen Zeichenfolge, einer so genannten GUID, wird in diesem Kapitel beschrieben.

15.2 OEM/Char

Windows verwendet den ANSI-Zeichensatz. Unter DOS war der auf 8 Bit erweiterte ASCII-Zeichensatz üblich. Dabei sind die Zeichen ab der Position 128 unterschiedlich. Wenn Sie unter Windows Texte oder Dateien öffnen, die unter `ASCII(OEM)` erstellt worden sind, werden die Zeichen ab Position 128 falsch dargestellt. Aber sie werden nicht nur falsch dargestellt, sie sind für `ANSI(CHAR)` auch an der falschen Position. Suchen und ersetzen ist keine gute Lösung. Es ist zeitraubend und wenn Sie dies programmtechnisch lösen wollen, müssen Sie quasi eine komplette Umrechnungstabelle mit Zeichencodes über 128 führen.

Es gibt aber API-Funktionen, die das Umrechnen in einem Rutsch und mit großer Geschwindigkeit erledigen. Und das in beide Richtungen. Es handelt sich dabei um die beiden Funktionen `CharToOemA` und `OemToCharA`, die ich in diesem Beispiel einsetze.

Listing 15.1
Beispiele\15_Sonstiges\15_01_Sonstiges.xls\mdlOEMCHAR

```
Private Declare Function CharToOem _
   Lib "user32" Alias "CharToOemA" ( _
   ByVal lpszSrc As String, _
   ByVal lpszDst As String _
   ) As Long

Private Declare Function OemToChar _
   Lib "user32" Alias "OemToCharA" ( _
   ByVal lpszSrc As String, _
   ByVal lpszDst As String _
   ) As Long

Public Function ToChar(ByVal strSource As String) As String
   ' Wandelt einen String von OEM (DOS) nach Char
   Dim strDest As String

   ' Puffer anlegen
   strDest = String(Len(strSource), 0)

   ' Umwandeln
   OemToChar strSource, strDest

   ' Zurückgeben
   ToChar = strDest

End Function

Public Function ToOem(ByVal strSource As String) As String
   ' Wandelt einen String von Char nach OEM (DOS)
   Dim strDest As String

   ' Puffer anlegen
   strDest = String(Len(strSource), 0)

   ' Umwandeln
   CharToOem strSource, strDest

   ' Zurückgeben
   ToOem = strDest

End Function


Public Sub TextdateiNachCharUmwandeln()
   ' Konvertiert Textdatei nach Char
   Dim varFile    As Variant
   Dim lngFileLen As Long
   Dim lngText    As String
   Dim FF         As Long

   varFile = Application.GetOpenFileName("Textdateien (*.txt), *.txt")
```

Listing 15.1 (Forts.)
Beispiele\15_Sonstiges\
15_01_Sonstiges.xls\
mdlOEMCHAR

```
    If varFile = False Then Exit Sub

    FF = FreeFile
    Open varFile For Binary As FF

        lngFileLen = LOF(1)

        lngText = String(lngFileLen, 0)

        Get #1, , lngText

        lngText = ToChar(lngText)

        Put FF, 1, lngText

    Close FF

End Sub

Public Sub TextdateiNachOEMUmwandeln()
    ' Konvertiert Textdatei nach OEM
    Dim varFile    As Variant
    Dim lngFileLen As Long
    Dim lngText    As String
    Dim FF         As Long

    varFile = Application.GetOpenFileName("Textdateien (*.txt), *.txt")

    If varFile = False Then Exit Sub

    FF = FreeFile
    Open varFile For Binary As FF

        lngFileLen = LOF(1)

        lngText = String(lngFileLen, 0)

        Get FF, , lngText

        lngText = ToOem(lngText)

        Put FF, 1, lngText

    Close FF

End Sub
```

ToChar, ToOem

Die beiden Funktionen `NachChar` und `NachOem` sind für die Umwandlung der Strings zuständig. Es sind die Hüllroutinen für die eigentlichen API-Funktionen `CharToOemA` und `OemToCharA`. Beide API-Funktionen bekommen als ersten Parameter den Quelltext übergeben. Der zweite Parameter ist jeweils ein Stringpuffer in der Größe, dass er den umgewandelten String aufnehmen kann.

TextdateiNachCharUmwandeln

Die Funktion `TextdateiNachCharUmwandeln` dient dazu, eine über einen Dialog ausgewählte Textdatei von `OEM`- in das `CHAR`-Format umzuwandeln. Dabei wird mit dem Dialog `GetOpenFileName` ein Dateiname erfragt, diese Datei anschließend als Binärdatei geöffnet und der gesamte Dateiinhalt in einer Stringvariablen gespeichert. Der String in dieser Variablen wird umgewandelt und mit `Put` in die Datei zurückgeschrieben.

TextdateiNachOEMUmwandeln

Die Funktion `TextdateiNachOEMUmwandeln` dient dazu, eine über einen Dialog ausgewählte Textdatei von `CHAR` in das `OEM`- Format umzuwandeln. Wie die Funktion arbeitet, können Sie im vorangegangenen Absatz bei `TextdateiNachCharUmwandeln` nachlesen.

15.3 Drucker auflisten und auswählen

Es ist recht einfach, programmgesteuert den aktiven Drucker zu wechseln. Die Eigenschaft `Application.ActivePrinter` braucht nur auf den entsprechenden Druckernamen gesetzt zu werden.

Das große Problem dabei ist aber, den korrekten Namen des Druckers zu finden. Der benötigte String enthält nämlich nicht nur den Druckernamen selbst, was ja auch zu einfach wäre, sondern auch noch den Anschlussnamen am Schluss. Ein Beispiel dafür ist »hp psc 900 series auf Ne01:«.

Und dieser Anschlussname steht leider nicht fest. Je nachdem, welcher Benutzer gerade angemeldet ist und welche Drucker diesem zur Verfügung stehen, kann dieser unterschiedlich ausfallen. Beispielsweise kann das bei dem einen Benutzer der Anschluss `Ne01` und bei dem anderen `Ne02` für den gleichen Drucker sein.

Der hier vorliegende Code ermittelt die kompletten Namen aller aktuell verfügbaren Drucker. NT-Systeme müssen dabei anders behandelt werden als 9x-Systeme, deshalb wird vorher die Version ermittelt. Auf den 9x-Systemen kann die API-Funktion `EnumPrinters` verwendet werden und erhält aus der `Printer_Info_2`-Struktur die gesuchten Informationen.

Unter NT muss in der Registry nachgeschaut werden, da in der `Printer_Info_2`-Struktur als `pPortName` leider nicht der richtige Anschlussname für Excel erscheint.

Wenn alle verfügbaren Drucker ermittelt wurden, können Sie sich aus der Liste den gewünschten Drucker mit dem richtigen Anschlussnamen heraussuchen. Wenn Sie dabei mit `InStr` nur nach dem Druckernamen selbst suchen, spielen bei der Suche die unterschiedlichen Anschlüsse auch keine Rolle.

In diesem Beispiel werden die Drucker in einer Listbox angezeigt und Sie können durch einen Klick auf den Button oder einen Doppelklick auf einen Eintrag den aktiven Drucker wechseln.

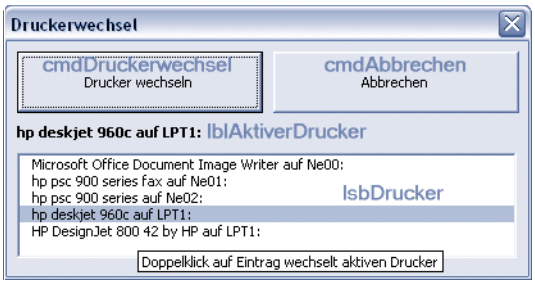


Abbildung 15.1
UserForm zum Wechseln des Druckers mit Steuerelementnamen

Listing 15.2
Beispiele\15_Sonstiges\15_01_Sonstiges.xls\ufChangePrinter

```
' Für Win 9x
Private Declare Function lstrcpy _
   Lib "kernel32.dll" Alias "lstrcpyA" ( _
   ByVal lpString1 As String, _
   ByVal lpString2 As Long _
   ) As Long

Private Declare Function lstrlen _
   Lib "kernel32.dll" Alias "lstrlenA" ( _
   ByVal lpString As Long _
   ) As Long

Private Declare Function EnumPrinters _
   Lib "winspool.drv" Alias "EnumPrintersA" ( _
   ByVal flags As Long, _
   ByVal name As String, _
   ByVal Level As Long, _
   pPrinterEnum As Long, _
   ByVal cdBuf As Long, _
   pcbNeeded As Long, _
   pcReturned As Long _
   ) As Long

Private Const PRINTER_ENUM_LOCAL = &H2
Private Const PRINTER_ENUM_NETWORK = &H40
Private Const PRINTER_ENUM_CONNECTIONS = &H4
Private Const PRINTER_ENUM_DEFAULT = &H1
Private Const PRINTER_ENUM_REMOTE = &H10
Private Const PRINTER_ENUM_SHARED = &H20

' Ab hier für NT
Private Declare Function RegEnumValue _
   Lib "advapi32.dll" Alias "RegEnumValueA" ( _
   ByVal hKey As Long, _
   ByVal dwIndex As Long, _
   ByVal lpValueName As String, _
   lpcbValueName As Long, _
   ByVal lngPtreserved As Long, _
   lpType As Long, _
   lpData As Byte, _
   lpcbData As Long _
   ) As Long
```

Listing 15.2 (Forts.)
Beispiele\15_Sonstiges\15_01_Sonstiges.xls\ufChangePrinter

```
Private Declare Function RegOpenKeyEx _
   Lib "advapi32.dll" Alias "RegOpenKeyExA" ( _
   ByVal hKey As Long, _
   ByVal lpSubKey As String, _
   ByVal ulOptions As Long, _
   ByVal samDesired As Long, _
   phkResult As Long _
   ) As Long

Private Declare Function RegCloseKey _
   Lib "advapi32.dll" ( _
   ByVal hKey As Long _
   ) As Long

Private Const KEY_ENUMERATE_SUB_KEYS = &H8
Private Const KEY_NOTIFY = &H10
Private Const KEY_QUERY_VALUE = &H1
Private Const READ_CONTROL = &H20000
Private Const KEY_READ = ( _
   READ_CONTROL Or _
   KEY_QUERY_VALUE Or _
   KEY_ENUMERATE_SUB_KEYS Or _
   KEY_NOTIFY)

Private Const HKEY_LOCAL_MACHINE = &H80000002
Private Const HKEY_CLASSES_ROOT = &H80000000
Private Const HKEY_CURRENT_USER = &H80000001
Private Const HKEY_CURRENT_CONFIG = &H80000005
Private Const HKEY_USERS = &H80000003
Private Const HKEY_PERFORMANCE_DATA = &H80000004
Private Const HKEY_DYN_DATA = &H80000006
Private Const constKeyName = _
    "Software\Microsoft\Windows NT\CurrentVersion\Devices\"

' Gemeinsamer Code
Private Type OSVersionInfo
   dwOSVersionInfoSize As Long
   dwMajorVersion As Long
   dwMinorVersion As Long
   dwBuildNumber As Long
   dwPlatformId As Long
   szCSDVersion As String * 128
End Type

Private Declare Function GetVersionEx _
   Lib "kernel32" Alias "GetVersionExA" ( _
   lpVersionInformation As OSVersionInfo _
   ) As Integer

Private mastrPrinter() As String

Private Sub Printerlist()
   Dim lngDummy      As Long
   Dim lngKeyHandle  As Long
   Dim lngIndex      As Long
   Dim lngPortLen    As Long
   Dim strField      As String
   Dim lngBuffLen    As Long
```

Listing 15.2 (Forts.)
Beispiele\15_Sonstiges\
15_01_Sonstiges.xls\
ufChangePrinter

```
    Dim strBuffer     As String
    Dim lngArrBuffLen As Long
    Dim abytBuffer()  As Byte

    ReDim mastrPrinter(1 To 500)

    ' Schlüssel öffnen
    lngDummy = RegOpenKeyEx(HKEY_CURRENT_USER, _
       constKeyName, 0&, KEY_READ, lngKeyHandle)

    If lngDummy <> 0 Then MsgBox "Falscher Schlüssel": Exit Sub

    ' Puffer für den Wertnamen erzeugen
    strField = String(1024, 0)
    lngBuffLen = 1023 'Länge Puffer Wertname

    ' Puffer für den Wert erzeugen
    ReDim abytBuffer(0 To 1024)
    lngArrBuffLen = 1024 'Länge Puffer Wert

    Do While RegEnumValue( _
       lngKeyHandle, lngIndex, strField, _
       lngBuffLen, 0&, ByVal 0&, _
       abytBuffer(0), lngArrBuffLen _
       ) = 0

       ' Bytearray in einen Unicodestring umwandeln
       strBuffer = (StrConv(abytBuffer, vbUnicode))

       ' Den String ab dem Komma bis einschließlich
       ' dem Doppelpunkt extrahieren
       ' strBuffer ist beispielsweise "winspool,LPT1:   "
       lngPortLen = InStr(1, strBuffer, ":") - _
           InStr(1, strBuffer, ",")
       strBuffer = Mid$(strBuffer, InStr(1, strBuffer, ":") _
           - 4, lngPortLen)

       ' Druckerstring zusammensetzen
       'strField ist beispielsweise "HP DesignJet 800 42 by HP"
       mastrPrinter(lngIndex + 1) = Left$(strField, lngBuffLen) & _
           " auf " & strBuffer

       lngIndex = lngIndex + 1

       lngArrBuffLen = 1024 ' Länge Puffer Wert

       ' Puffer für den Wertnamen erzeugen
       strField = String(1024, 0)
       lngBuffLen = 1023 ' Länge Puffer Wertname

    Loop ' Nächster Drucker

    ReDim Preserve mastrPrinter(1 To lngIndex + 1)

    lngDummy = RegCloseKey(lngKeyHandle)

End Sub
```

Listing 15.2 (Forts.)
Beispiele\15_Sonstiges\15_01_Sonstiges.xls\ufChangePrinter

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

Private Sub cmdDruckerwechsel_Click()
   On Error GoTo Fehlerbehandlung

   ' Aktiven Drucker ändern
   Application.ActivePrinter = lsbDrucker.List( _
      lsbDrucker.ListIndex)

   ' Name des aktiven Druckers ausgeben
   lblAktiverDrucker.Caption = Application.ActivePrinter

   Exit Sub

Fehlerbehandlung:
   MsgBox "Der Drucker kann nicht ausgewählt werden"
End Sub

Private Sub lsbDrucker_DblClick(ByVal Cancel As _
   MSForms.ReturnBoolean)

   cmbDruckerwechsel_Click

End Sub

Private Sub UserForm_Initialize()
   Dim lngPtr        As Long
   Dim strPtr        As String
   Dim lngActivePtr  As Long
   Dim udtOSVersion  As OSVersionInfo

   With udtOSVersion
      ' Version ermitteln
      .dwOSVersionInfoSize = Len(udtOSVersion)
      .szCSDVersion = Space$(128)
      GetVersionEx udtOSVersion

      ' Aktiver Drucker
      strPtr = Application.ActivePrinter

      If .dwPlatformId = 2 Then
         ' NT Familie
         Printerlist
      Else
         'Win 9xx Familie
         Printerlist9x
      End If
   End With

   For lngPtr = 1 To UBound(mastrPrinter)

       ' Aktiver Drucker
      If strPtr = mastrPrinter(lngPtr) Then _
        lngActivePtr = lngPtr - 1
```

Listing 15.2 (Forts.)
Beispiele\15_Sonstiges\
15_01_Sonstiges.xls\
ufChangePrinter

```
        ' Zur Liste hinzufügen
        lsbDrucker.AddItem mastrPrinter(lngPtr)

    Next

    ' Aktiven Drucker in Liste markieren
    lsbDrucker.ListIndex = lngActivePtr

    ' Name des aktiven Druckers ausgeben
    lblAktiverDrucker.Caption = Application.ActivePrinter

End Sub

'#############################################################
'Nur für Win 9x
'#############################################################
Private Sub Printerlist9x()
    Dim alngBuffer()  As Long
    Dim lngLen        As Long
    Dim lngIndex      As Long
    Dim lngCount      As Long
    Dim lngRet        As Long
    Dim lngBufferPtr  As Long
    Dim lngPtrType    As Long
    Dim strPtr        As String

    ReDim alngBuffer(1)

    lngPtrType = PRINTER_ENUM_CONNECTIONS Or _
        PRINTER_ENUM_LOCAL Or PRINTER_ENUM_NETWORK

    ' Benötigte Puffergröße ermitteln
    lngRet = EnumPrinters(lngPtrType, _
        vbNullString, 2, alngBuffer(0), 0&, _
        lngLen, lngCount)

    ' Puffer bereitstellen
    ReDim alngBuffer(0 To (lngLen + 3) \ 4)

    ' Infos über Drucker holen
    lngRet = EnumPrinters(lngPtrType, vbNullString, _
        2, alngBuffer(0), lngLen, lngLen, lngCount)

    If lngCount = 0 Then MsgBox "Keine Drucker verfügbar": Exit Sub

    ReDim mastrPrinter(1 To lngCount)

    For lngIndex = 1 To lngCount

        ' Printerinfostruktur(lngIndex)
        lngBufferPtr = lngBufferPtr + 1

        ' Printerinfostruktur, Element pPrinterName
        strPtr = StringVonPointer(alngBuffer(lngBufferPtr))
        lngBufferPtr = lngBufferPtr + 2
```

Listing 15.2 (Forts.)
Beispiele\15_Sonstiges\15_01_Sonstiges.xls\ufChangePrinter

```
        ' Printerinfostruktur, Element pPortName
        strPtr = strPtr & " auf "
        strPtr = strPtr & StringVonPointer(alngBuffer(lngBufferPtr))

        mastrPrinter(lngIndex) = strPtr

        lngBufferPtr = lngBufferPtr + 18

        ' Nächste Printerinfostruktur 2
    Next
End Sub

Private Function StringVonPointer(lngPtr As Long)
    Dim lngCount   As Long
    Dim strName    As String

    lngCount = lstrlen(lngPtr)

    strName = String(lngCount, 0)

    lstrcpy strName, lngPtr

    If InStr(1, strName, Chr(0)) <> 0 Then
        strName = Left$(strName, _
            InStr(1, strName, Chr(0)) - 1)
    End If

    StringVonPointer = strName

End Function
```

UserForm_Initialize

Beim Ausführen der Initialisierungsprozedur der UserForm wird zuerst die Windowsversion ermittelt. Dazu wird die API-Funktion `GetVersionEx` benutzt, die eine Struktur vom Typ `OSVersionInfo` mit den Informationen über das Betriebssystem füllt. Da für dieses Beispiel nur die Betriebssystemfamilie `NT/9X` wichtig ist, brauchen Sie nur das Element `dwPlatformId` der Struktur auswerten. Als Ergebnis wird eine der drei folgenden Konstanten geliefert:

```
Private Const VER_PLATFORM_WIN32_NT = 2
Private Const VER_PLATFORM_WIN32_WINDOWS = 1
Private Const VER_PLATFORM_WIN32s = 0
```

Ist das Element `dwPlatformId` gleich 2, wird die Prozedur `Printerlist` aufgerufen, ansonsten `Printerlist9x`. Diese Prozeduren füllen das Array `mastrPrinter` mit den kompletten Namen, die anschließend in einer Schleife in die Listbox `lsbDrucker` eingetragen werden. Beim Eintragen in die Listbox prüfen Sie gleichzeitig, welcher Eintrag der aktuelle Drucker ist, wählen diesen Eintrag aus und zeigen den Namen im Label `lblAktiverDrucker` an.

Printerlist

Diese Prozedur ermittelt die verfügbaren Drucker unter NT-Betriebssystemversionen wie NT, Windows 2000 und Windows XP.

Dazu werden die Werte im Schlüssel

HKEY_CURRENT_USER\Software\Microsoft\Windows NT\CurrentVersion\ Devices

ausgelesen. Diese enthalten als Wertnamen die Drucker und als Wert selbst den zugehörigen Anschluss.

Mit der API-Funktion `RegOpenKeyEx` wird der Schlüssel geöffnet und ein Handle darauf geliefert. Die Variable `lngKeyHandle`, die als Parameter mit übergeben wurde, nimmt dieses Handle auf.

Anschließend werden mit der Funktion `RegEnumValue` alle Werte dieses Schlüssels mit Wertnamen und Wert nacheinander ausgelesen. Dazu wird bei jedem Aufruf der Index `lngIndex` um eins erhöht, und das so lange, bis die Funktion `RegEnumValue` als Funktionsergebnis einen Wert ungleich null liefert.

Der Stringpuffer `strField` nimmt dabei den Feldnamen und das Bytearray `abytBuffer` den eigentlichen Wert auf. Das Bytearray wird dann noch mit `StrConv` in einen Unicodestring umgewandelt und der Anschluss aus diesem String herausgelesen. Dieser steht dort hinter dem Komma und wird mit einem Doppelpunkt abgeschlossen. Wertname und Anschluss werden zum Druckernamen zusammengesetzt und im Array `mastrPrinter` gespeichert. Zum Schluss wird mit `RegCloseKey` der anfangs geöffnete Schlüssel wieder geschlossen.

Druckerliste9x

Diese Prozedur ermittelt die verfügbaren Drucker unter 9x-Betriebssystemversionen wie Windows 95, 98 und ME.

Dazu wird die Funktion `EnumPrinters` benutzt, die einen Puffer mit einem Array von `PrinterInfo`-Strukturen füllt. Es werden Strukturen vom Typ zwei benötigt, deshalb wird der dritte Parameter von `EnumPrinters` auf zwei gesetzt.

Da zu Beginn noch nicht bekannt ist, wie viele Drucker vorhanden sind und Sie somit nichts über die benötigte Puffergröße wissen, wird diese Funktion mit dem auf null gesetzten Parameter `cbBuf` aufgerufen, der angibt, wie groß der übergebene Puffer ist. Die als Parameter `pcbNeeded` übergebene Variable `lngLen` wird, wenn der Puffer zu klein ist, mit der benötigten Puffergröße gefüllt und bei einer angegebenen Größe von null ist er meistens zu klein.

Jetzt wird ausgerechnet, wie viel Elemente das Longarray `alngBuffer` haben muss, um alle Bytes, die geliefert werden, auch aufzunehmen zu können und das Array wird entsprechend redimensioniert. Jetzt können Sie `EnumPrinters` mit allen wichtigen Parametern aufrufen und bekommen den Puffer `alngBuffer` mit den Daten gefüllt. Die als Parameter übergebene Variable `lngCount` liefert dabei die Anzahl der `Printer_Info_2`-Strukturen und somit auch die Anzahl der Drucker.

Eine Schleife wird anschließend so oft durchlaufen, wie Drucker vorhanden sind.

Die `Printer_Info_2`-Struktur besteht aus 21 Longwerten, wobei einige davon Zeiger auf Strings, Strukturen oder ganz einfach einen Wert darstellen. Der Longwert `pPrinterName` jeder `Printer_Info_2`-Struktur ist ein Zeiger auf einen Sting, der den Printernamen enthält. Dieser ist in der Struktur an zweiter Stelle, deshalb wird der Zeiger `lngBufferPtr` zu Beginn um eins erhöht. Um den vierten Wert `pPortName` auszulesen, wird `lngBufferPtr` um zwei erhöht und der nun gelieferte Wert zeigt auf einen String, welcher den Anschluss kennzeichnet. Diese beiden Zeiger werden jeweils an die Funktion `StringVonPointer` übergeben und liefern den an der entsprechenden Speicherstelle stehenden String zurück. Zusammengesetzt als vollständiger Druckername werden diese Strings im Array `masDrucker` gespeichert. Danach wird `lngBufferPtr` um 18 erhöht, um auf den Beginn der nächsten `Printer_Info_2`-Struktur zu zeigen.

Jetzt beginnt das Spiel von neuem, deshalb wird der Zeiger `lngBufferPtr` um eins erhöht und der Zeiger `pPrinterName` geholt, dieser an die Funktion `StringVonPointer` übergeben und so weiter und so fort.

StringVonPointer

Oft wird von einer API-Funktion nur ein Zeiger auf einen nullterminierten String geliefert. Die Funktion `StringVonPointer` holt sich den String von dieser Speicherstelle.

Dazu wird mit `lstrlen` die Länge des Strings im Speicher ermittelt. Anschließend wird ein String in dieser Länge erzeugt und mittels `lstrcpy` ab dem Pointer in den vorher angelegten kopiert.

cmdDruckerwechsel_Click

Bei einem Klick auf den Button CMDDRUCKERWECHSEL wird der aktive Drucker gewechselt. Dazu wird der Listindex der Listbox `lsbDrucker` ermittelt und der entsprechende Eintrag als neuer Druckername benutzt. Anschließend wird der neue Druckername im Label `lblAktiverDrucker` angezeigt.

lsbDrucker_DblClick

Bei einem Doppelklick in der Listbox `lsbDrucker` wird die Prozedur `cmbDruckerwechsel_Click` aufgerufen.

cmdAbbrechen_Click

Bei einem Klick auf den Button CMDABBRECHEN wird die UserForm entladen.

15.4 Beschreibung für eigene Funktionen

Um Funktionsbeschreibungen für benutzerdefinierte Funktionen zu realisieren, gibt es nicht sehr viele Möglichkeiten. Eine davon ist die, im Objektkatalog die Funktion herauszusuchen, zu markieren, mit der rechten Maustaste darauf zu klicken und unter EIGENSCHAFTEN | BESCHREIBUNG einen Hilfetext einzugeben.

Eine andere Möglichkeit ist, die Methode `MacroOptions` zu benutzen. Hier ein kleines Beispiel, näheres dazu finden Sie in der Hilfe.

Listing 15.3
Beispiele\15_Sonstiges\15_01_Sonstiges.xls\mdlDescriptions

```
Private Sub ZuweisenEinerKategorie()
    ' 1 Finanzmathematisch
    ' 2 Datum & Zeit
    ' 3 Math. & Trigonom.
    ' 4 Statistik
    ' 5 Matrix
    ' 6 Datenbank
    ' 7 Text
    ' 8 Logik
    ' 9 Information
    ' 10 Befehle
    ' 11 Benutzerorientiert
    ' 12 Makrosteuerung
    ' 13 DDE/Extern
    ' Ab 14 Benutzerdefiniert
   Application.MacroOptions Macro:="Functionstest", _
      Description:="Meine eigene Beschreibung", _
      Category:=9
End Sub
```

Eine andere Möglichkeit ist die, eine Vorgängerfunktion zu benutzen. Die Funktion `Register` diente in früheren Versionen (Excel4) einmal dazu, fremde Coderessourcen verfügbar zu machen, deshalb wird sie auch mit `Application.ExecuteExcel4Macro` aufgerufen. Es wird ein String gebraucht, weshalb der größte Teil des Makros sich damit beschäftigt, diesen String zu erzeugen.

Der String muss folgendermaßen aufgebaut sein:

Listing 15.4
Beispiele\15_Sonstiges\15_01_Sonstiges.xls\mdlDescriptions

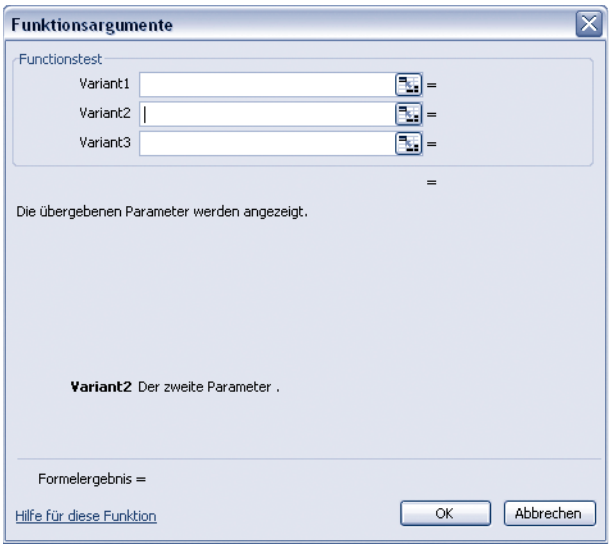
```
REGISTER(
"Vorhandene dll",
"Vorhandene Funktion in dieser dll",
"",
"Name der zu beschreibenden Funktion",
"Parametername 1,Parametername 2,Parametername 3, etc.",
"1",
"Neuer/Vorhandener Kategoriename",,,
"Funktionsbeschreibung",
"Parameterbeschreibung 1",
"Parameterbeschreibung 2",
"Parameterbeschreibung 3.",
"etc. ")
```

Nachfolgend der Code dazu:

```
Public Function Functionstest(a, b, c) As String
    ' Dummy-Funktion, um das Prinzip zu zeigen
   Functionstest = "Argument a=" & a & _
       ", Argument b=" & b & ", Argument c=" & c
End Function

Public Sub Funktionsbeschreibung()
    ' Für jede Beschreibung muss eine andere
    ' API-Funktion benutzt werden. Die Beschreibungen
    ' gehen beim Beenden verloren. Daher beim Öffnen
    ' der Mappe automatisch starten lassen.
   Dim strFunktionsname                As String
   Dim strParameter                    As String
   Dim strDummy                        As String
   Dim strKategorie                    As String
   Dim strBeschreibung                 As String
   Dim strArgumentbeschreibung(1 To 3) As String
   Dim strAnf                          As String
   Dim strÜbergabe                     As String
   Dim strKomma                        As String

   strAnf = """": strKomma = ","

   ' Jede beliebige API-Funktion
   ' kann benutzt werden. Sie muss nur
   ' in der entsprechenden .dll vorhanden
   ' sein. Groß- und Kleinschreibung beachten.
   strDummy = "kernel32" & strAnf & "," _
       & strAnf & "GetACP"

   ' Der Name der zu beschreibenden Funktion
   strFunktionsname = "Functionstest"

   ' Die zukünftigen Parameternamen
   strParameter = "Variant1,Variant2,Variant3"

   ' Die Kategorie
   strKategorie = "Neuer Eintrag"

   ' Hier kommt die allgemeine Beschreibung hin
   strBeschreibung = "Die übergebenen Parameter werden angezeigt"

   ' Die Beschreibung der einzelnen Parameter
   strArgumentbeschreibung(1) = _
       "Der erste Parameter "
   strArgumentbeschreibung(2) = _
       "Der zweite Parameter "
   strArgumentbeschreibung(3) = _
       "Der dritte Parameter "

   ' Der String für das Excel4-Makro wird erstellt
   strÜbergabe = "REGISTER(" & strAnf & _
       strDummy & strAnf
   strÜbergabe = strÜbergabe & strKomma & _
       strAnf & strAnf
   strÜbergabe = strÜbergabe & strKomma & _
       strAnf & strFunktionsname & strAnf
```

```
    strÜbergabe = strÜbergabe & strKomma & _
        strAnf & strParameter & strAnf
    strÜbergabe = strÜbergabe & strKomma & _
        strAnf & "1" & strAnf
    strÜbergabe = strÜbergabe & strKomma & _
        strAnf & "" & strKategorie & strAnf
    strÜbergabe = strÜbergabe & strKomma & _
        strKomma
    strÜbergabe = strÜbergabe & strKomma & _
        strAnf & strBeschreibung & strAnf
    strÜbergabe = strÜbergabe & strKomma & _
        strAnf & strArgumentbeschreibung(1) _
        & strAnf
    strÜbergabe = strÜbergabe & strKomma & _
        strAnf & strArgumentbeschreibung(2) _
        & strAnf
    strÜbergabe = strÜbergabe & strKomma & _
        strAnf & strArgumentbeschreibung(3) _
        & ". " & strAnf
    strÜbergabe = strÜbergabe & ")"

    ' Das Excel4-Makro wird gestartet
    Application.ExecuteExcel4Macro strÜbergabe

End Sub

Public Sub UnregisterFunctionsbeschreibung()
    Dim strFunktionsname    As String
    Dim strAnf              As String
    Dim strÜbergabe         As String
    Dim strKomma            As String
    Dim strDummy            As String

    ' Hier wird die Beschreibung gelöscht

    strAnf = """": strKomma = ","

    strDummy = "kernel32" & strAnf & "," & _
        strAnf & "GetACP"

    strFunktionsname = "Functionstest"

    strÜbergabe = "REGISTER(" & strAnf & _
        strDummy & strAnf
    strÜbergabe = strÜbergabe & strKomma & _
        strAnf & strAnf
    strÜbergabe = strÜbergabe & strKomma & _
        strAnf & strFunktionsname & strAnf
    strÜbergabe = strÜbergabe & strKomma & _
        strKomma & "0)"

    Application.ExecuteExcel4Macro strÜbergabe

    strÜbergabe = "UNREGISTER(" & strFunktionsname & ")"
    Application.ExecuteExcel4Macro strÜbergabe

End Sub
```

Abbildung 15.2
Der Funktionsassistent nach der Änderung



Wenn Sie mehr Informationen über solche, immer noch funktionierende Excel4-Makros benötigen, suchen Sie einfach mal bei Microsoft nach *macrofun.exe*.

15.5 Freischaltcode erzeugen

Die Bibliothek `oleaut32` bietet mit der Funktion `LhashValOfNameSys` die Möglichkeit, aus einem String einen eindeutigen 32-Bit-Wert zu erzeugen. Dieser gelieferte Wert ist auf jedem Rechner gleich, wenn der übergebene String auch gleich ist. Das nennt man einen *Hashwert*.

Mit solch einem Wert können Sie beispielsweise zusammen mit einem eindeutigen Benutzernamen eine Seriennummer erzeugen, die Sie an einen Kunden zum Freischalten einer Anwendung senden. Mit dem Benutzernamen und der zugesendeten Seriennummer kann dieser nun die Version freischalten. Diese Kombination kann zwar auf beliebigen Rechnern benutzt werden, aber wenn solch eine Kombination im Netz herumgeistert kennen Sie wenigstens den Urheber der Kopien.

Denkbar und von verschiedenen Firmen auch angewendet, ist das Verwenden von Hardware-Informationen wie die Seriennummer der Festplatte oder die Mac-Adresse der Netzwerkkarte. Diese Informationen werden dann an der Stelle des Benutzernamens verwendet und die Seriennummer funktioniert anschließend nur noch auf diesem Rechner.

Zum Erstellen des Freischaltcodes braucht der Verkäufer aber diese Hardware-Informationen, die ihm nach einer entsprechenden Rückfrage am besten programmgesteuert zugesendet werden. Wenn Sie aber zu solch harten Mitteln greifen, wird Ihnen Ihr Kunde möglicherweise nicht mehr lange treu bleiben,

spätestens dann, wenn er das erste Mal eine defekte Netzwerkkarte wechseln muss, kommt Freude auf.

Damit das ganze Verfahren überhaupt funktioniert, ist es notwendig, dass beim Kunden und bei Ihnen das gleiche Verfahren zum Erzeugen der Seriennummer angewendet wird. Nur darf der Kunde freilich nicht wissen, wie die Seriennummer erzeugt wird, sonst könnte er sich selbst zu einem frei wählbaren Benutzernamen eine passende Seriennummer erzeugen. Am besten wird dieser Code also in eine *.dll* oder als geschütztes Add-In ausgelagert, wobei erwähnt werden muss, dass der Schutz in Excel nicht besonders sicher ist. Der Schutz eines VBA-Projektes ist beispielsweise in weniger als einer Minute aufgehoben.

Diese Seriennummer kann der Anwender dann zusammen mit den zugehörigen Benutzernamen in einer *.ini*-Datei, in der Registry oder als `CustomDocumentProperties` speichern. Beim Öffnen der Datei wird aus dem Benutzernamen oder der Hardware die Seriennummer erzeugt und mit der gespeicherten verglichen. Bei Übereinstimmung passt der Schlüssel ins Schloss und Sie können dem Anwender das uneingeschränkte Benutzen erlauben.

Das alles ist natürlich kein hundertprozentiger Schutz, denn bei dieser Methode muss das Ausführen von VBA-Code schließlich erlaubt sein, was Sie ja ohne Probleme beim Starten verhindern können. Um das zu erschweren, können Sie aber wichtige Tabellenblätter mit VBA ausblenden (`Visible=xlVeryHidden`), die anschließend nicht mehr per Hand sichtbar gemacht werden können.

Um diese Blätter einzublenden, muss die Ausführung von VBA aktiviert sein, unterbindet der Nutzer dies, wird auch das notwendige Einblenden dieser Mappen nicht mehr durchgeführt. Selbstverständlich können Sie auch diese Methode aushebeln. Leider ist es ganz einfach so, dass nahezu alle Schutzmechanismen von Excel nicht wirklich sicher sind. Die einzig zuverlässige Methode ist nur der Arbeitsmappenschutz mit einem langen, aus Buchstaben und Zahlen zufällig zusammengesetzten Passwort. Dieser Schutz ist meines Wissens ohne `Brute-Force`-Angriff nicht zu knacken.

Aber für den unbedarften Anwender ist die Methode mit dem Ausblenden von Tabellenblättern sicher ausreichend und wer wirklich so gewieft ist, dass er so etwas Aushebeln kann, erstellt sich meiner Ansicht nach schneller eine eigene, auf seine Bedürfnisse zugeschnittene Mappe, bevor er mühselig eine fremde knackt und anpasst.

Ein Hashwert sollte mehrere Kriterien erfüllen:

Es soll nahezu unmöglich sein, zwei Wörter oder Zeichenfolgen zu finden, die das gleiche Ergebnis liefern. Zweitens sollte es schwer sein, aus einem gegebenen `Hash`-Wert den String zu rekonstruieren, aus dem dieser Wert erzeugt wurde.

Die Funktion `LhashValOfNameSys` liefert einen 32-Bit-Longwert. Dieser kann, als unsigned (nicht vorzeichenbehaftet) interpretiert, Werte bis ca. 4,2 Milliarden annehmen. Es ist somit theoretisch möglich, dass zwei oder mehr verschiedene Strings den gleichen Longwert liefern, aber es ist nicht sehr wahrscheinlich, dass Sie zufällig damit konfrontiert werden.

Aus solch einem Hashwert auf den ursprünglichen String zu schließen ist trotzdem nahezu unmöglich, da ja theoretisch unendlich viele Strings dieses eine Ergebnis liefern können. Mittels `Brute-Force` können Sie zwar einen oder mehrere Strings finden, die das richtige Ergebnis liefern. Wenn Sie dies aber künstlich verzögern, indem Sie bei jedem Versuch die Wartezeit bis zum nächsten erhöhen, sollte es in annehmbarer Zeit kaum möglich sein, einen solchen auch tatsächlich zu finden, außer natürlich, man kennt den zugrunde liegenden Algorithmus.

Die Funktionalität können Sie wie folgt benutzen:

Listing 15.5
Beispiele\15_Sonstiges\
15_01_Sonstiges.xls\Tabelle3

```
Private Sub cmdErzeugen_Click()
   Dim strUsername   As String
   Dim strKey        As String

   strUsername = Me.Range("B6")

   strKey = CreateUnlockKey(strUsername)

   Me.Range("B7") = strKey

End Sub

Private Sub cmdSpeichern_Click()
   Dim strKey As String
   On Error Resume Next
   strKey = Me.Range("B7")

   ' Dokumenteigenschaft löschen
   ThisWorkbook.CustomDocumentProperties.Item( _
      "Freischaltschlüssel").Delete

   ' Dokumenteigenschaft hinzufügen
   ThisWorkbook.CustomDocumentProperties.Add _
      name:="Freischaltschlüssel", _
      LinkToContent:=False, _
      Type:=msoPropertyTypeString, _
      Value:=strKey, _
      LinkSource:=False

End Sub

Private Sub cmdVerifizieren_Click()
   Dim strUsername As String

   strUsername = Me.Range("B6")

   If Verify(strUsername) Then
      MsgBox "Richtiger Benutzer"
   Else
      MsgBox "Falscher Benutzer"
   End If

End Sub
```

cmdErzeugen_Click

In dieser Ereignisprozedur wird aus der Zelle B6 der Benutzername geholt und dieser als Parameter an die Funktion `CreateUnlockKey` übergeben. Der zurückgegebene Code wird in die Zelle B7 eingetragen.

cmdSpeichern_Click

In dieser Ereignisprozedur wird aus der Zelle B7 der `Freischaltkey` geholt und dieser als `CustomDocumentProperties`-»Freischaltschlüssel« gespeichert.

cmdVerifizieren_Click

In dieser Ereignisprozedur wird aus der Zelle B6 der Benutzername geholt und dieser als Parameter an die Funktion `Verify` übergeben. Das Ergebnis dieser Abfrage wird in einer Messagebox ausgegeben.

Listing 15.6
Beispiele\15_Sonstiges\
15_01_Sonstiges.xls\
mdlHashcode

```
Private Declare Function LHashValOfNameSys _
   Lib "oleaut32" ( _
   ByVal syskind As Long, _
   ByVal lcid As Long, _
   ByVal szName As Long _
   ) As Long

Private Const SYS_WIN16 = 0
Private Const SYS_WIN32 = 1
Private Const SYS_MAC = 2
Private Const LCidGermany1 As Long = 1031 'Deutschland
Private Const LCidGermany2 As Long = 2055 'Schweiz
Private Const LCidGermany3 As Long = 3079 'Österreich
Private Const LCidGermany4 As Long = 4103 'Luxemburg
Private Const LCidGermany5 As Long = 5127 'Liechtenstein

' In einem geschützten Projekt
Public Function CreateUnlockKey( _
   strUsername As String _
   ) As String

   Dim i             As Long
   Dim lngHash       As Long
   Dim strUnlockKey  As String

   strUnlockKey = strUsername

   ' Irgendeine Rechenvorschrift zum Erzeugen eines eindeutigen
   ' Keys. Nicht zu einfach, damit man nicht leicht auf
   ' den Algorithmus schließen kann.
   For i = 1 To 5

      lngHash = CStr(LHashValOfNameSys(SYS_WIN32, _
         LCidGermany1, strPtr(strUnlockKey)))

      CreateUnlockKey = CreateUnlockKey & _
         Format(CLng(Mid(lngHash, 2, 3)), "000") & "-"
```

Listing 15.6 (Forts.)
Beispiele\15_Sonstiges\15_01_Sonstiges.xls\mdlHashcode

```
        strUnlockKey = CreateUnlockKey

    Next

    CreateUnlockKey = Left(CreateUnlockKey, _
        Len(CreateUnlockKey) - 1)

End Function

Public Function Verify(strUsername As String) As Boolean
    On Error GoTo Fehlerbehandlung
    If ThisWorkbook.CustomDocumentProperties( _
        "Freischaltschlüssel") = _
        CreateUnlockKey(strUsername) Then _
        Verify = True
Fehlerbehandlung:
End Function
```

CreateUnlockKey

In dieser Funktion wird aus einem als Parameter übergebenen String ein `Freischaltkey` erzeugt.

Die API-Funktion `LhashValOfNameSys` erzeugt aus diesem übergebenen String einen Longwert, der in einen String umgewandelt wird. Aus diesem werden drei Ziffern extrahiert und an den zunächst leeren Freischaltcode mit einem abschließenden Bindestrich gehängt. Dieser Freischaltcode wird in einer Schleife mehrmals an `LhashValOfNameSys` übergeben und jeweils drei zusätzliche Ziffern aus dem daraus erzeugten Hashwert an den Freischaltcode gehängt. Zuletzt wird noch der abschließende Bindestrich entfernt und fertig ist der Code.

Verify

In dieser Funktion wird aus einem als Parameter übergebenen Benutzernamen mithilfe der Funktion `CreateUnlockKey` ein Freischaltkey erzeugt. Dieser wird mit dem in der Auflistung `CustomDocumentProperties` als Element »Freischaltschlüssel« hinterlegten Schlüssel verglichen und bei Übereinstimmung wird von der Funktion der boolesche Wahrheitswert `WAHR` zurückgeliefert.

15.6 GUID

Manchmal wird ein wirklich eindeutiger Wert benötigt, der auch sonst nirgendwo mehr vorkommen darf.

Ein mögliches Einsatzgebiet wären beispielsweise Kundennummern. Jetzt könnte man sagen, dass man ja nachschauen kann, ob eine solche Nummer bereits existiert und dann einfach eine andere benutzen. Das wird aber spätestens dann problematisch, wenn auf hunderten von Laptops ohne Verbindung untereinander sofort eine eindeutige Kundennummer generiert werden soll.

Ähnlich ist der Fall bei OLE. Jedes Objekt besitzt eine eindeutige Kennzeichnung, die eine Anwendung in die Lage versetzt, in der Registry nachzuschauen, welche andere Anwendung ein eingefügtes OLE-Objekt – beispielsweise ein

Dokument – bearbeiten kann. Und das auf jedem Rechner, auf dem das Fremdprogramm registriert ist. Diese Kennung ist die so genannte *GUID* (`Globally Unique Identifier`).

Wenn Sie sich die Registry anschauen, entdecken Sie Unmengen von solchen GUIDs. Es gibt noch weitere Kennungen, die im Zusammenhang mit OLE eingesetzt werden, beispielsweise CLSID, UUID und IID. Das sind aber auch im Prinzip GUIDs, die alle folgendermaßen aussehen (natürlich nur als Schema):

```
{B97FF9DB-A2BE-4BC3-B4AF-B1F0BDEA5D6B}
```

Ist einem Objekt einmal solch ein Wert zugewiesen worden, muss sichergestellt sein, dass niemals ein anderes Objekt den gleichen Wert zugewiesen bekommt. Ein einmal festgelegter Wert für ein Objekt darf auch nie mehr geändert werden, es wird notfalls ein anderes Objekt angelegt. Die Ähnlichkeit zu den zuerst erwähnten Kundennummern liegt darin, dass Tausende Programmierer unabhängig voneinander, ohne eine Registrierungsstelle zu bemühen, so viel Objekte, Klassen und Interfaces anlegen können, wie sie möchten, ohne jemals eine bereits existierende `GUID` doppelt zu benutzen.

Bei der Erzeugung einer GUID mit der API-Funktion `CoCreateGuid` ist tatsächlich sichergestellt, dass diese wirklich einmalig im Universum ist. Zum einen wird bei der Erzeugung die eindeutige Netzwerkkartenadresse mit einbezogen und selbst wenn nur der eine Benutzer mit der gleichen Netzwerkkarte sein Leben lang mit höchster Geschwindigkeit GUIDs produziert, wird er niemals eine doppelte produzieren. Eine GUID ist eine 16-Byte-Zahl und bei dieser extrem hohen Anzahl von Möglichkeiten ist die Chance, zufällig eine gleiche zu produzieren, mikroskopisch klein.

Listing 15.7
Beispiele\15_Sonstiges\
15_01_Sonstiges.xls\mdlGuid

```
Private Declare Function StringFromGUID2 _
   Lib "OLE32.dll" ( _
   lpGUID As Long, _
   ByVal lpszString As Long, _
   ByVal lMax As Long _
   ) As Long

Private Declare Function CoCreateGuid _
   Lib "OLE32.dll" ( _
   lpGUID As Long _
   ) As Long

Private Sub Test()
   MsgBox CreateGuid, , "GUID"
End Sub

Public Function CreateGuid() As String
   Dim alngGUID(1 To 4) As Long

   CoCreateGuid alngGUID(1)

   CreateGuid = String(38, 0)

   StringFromGUID2 alngGUID(1), strPtr(CreateGuid), 39

End Function
```

CreateGuid

Die API-Funktion `CoCreateGuid` erzeugt eine 128 Bit große Zahl, die das Longarray `alngGUID` mit vier Elementen aufnimmt. Anschließend wird ein Stringpuffer mit 38 Zeichen erzeugt, der die `GUID` aufnehmen soll. Dazu wird die Funktion `StringFromGUID2` benutzt, der als Parameter ein Pointer auf die 128-Bit-Zahl und ein Pointer auf den Stringpuffer übergeben wird.

Der Pointer auf die 128-Bit-Zahl wird übergeben, indem Sie das erste Element des Arrays als Referenz übergeben. Der Pointer auf den String wird mit der undokumentierten Funktion `StrPtr` ermittelt und übergeben. Der letzte Parameter ist die Größe des Stringpuffers inklusive abschließender Null.

Stichwortverzeichnis

A

ActiveConnection 119
ActivePrinter 468
ActiveX Data Objects 115
adAffectAll 122
adAffectCurrent 121–122
adAffectGroup 121–122
Add 415
AddressOf 39, 187, 193
adFilterAffectedRecords 122
adFilterConflictingRecords 122
adFilterFetchedRecords 122
adFilterNone 122
adFilterPendingRecords 122
AdjustTokenPrivileges 325
adLockBatchOptimistic 119
adLockOptimistic 119
adLockPessimistic 119
adLockReadOnly 119
ADO 115
adOpenDynamic 119
adOpenForwardOnly 119
adOpenKeyset 119
adOpenStatic 119
ADOX 113
Adressraum 142
adSchemaColumns 120
adSchemaProviderSpecific 120
adSchemaTables 120
adUseClient 118
adUseServer 118
Alias 147, 355
Analyse-Add-In 288
And 47
AND NOT 382
ANSI 148, 465
API 142
  Präfix 154
Append 243
Application Programming Interface 142
Arccos 292
Arrays 162
ASCII 148, 465
Attributes-Eigenschaft 240
Auflistungen 73
  Element 74
  For Each 74–75
  Item 75
  Löschen 76

B

Base64 148
Baumstruktur 420
Bedingte Kompilierung 71
Beep-API 352
Beep-Frequenz 352
Beep-VBA 351
bEnable 391
BFFM_INITIALIZED 217
BFFM_SELCHANGED 217
BFFM_SETSELECTION 217
BFFM_SETSTATUSTEXT 217
Bilderschau 295
Binary 243
Bit 148
BitsPerSample 368
BITSPIXEL 311
Blackbox 92
Breitengrad 292
BSTR 155
Bubblesort 66
BuiltInDialog 209
Buß- und Bettag 287
ByRef 41
ByVal 41

C

Callback-Funktion 39, 193
cAlternate 237
Cancel 384
Caption 176
Cards.dll 325
cbBuf 475
CBool 26
cbRemoteName 276
CByte 26
CCur 26
CDate 26
CDbl 26
CDS_UPDATEREGISTRY 320
CF_Text 204
cFileName 237
ChangeDisplaySettings 320
ChangeShutdownPrivileges 324
Char 465
CharLowerBuff 147
CharToOemA 466
ChDir 209, 220
CHOOSEFONT 208
ChooseFontA 208
CHOOSEMYCOLOR 203
CInt 26
Class_Initialize 82
Clientfenster 309
CLng 26
Close 395
CloseClipboard 312, 333
CloseHandle 269, 408
CLSID 485
clsInternet 445
clsTracert 431
CoCreateGuid 485–486
Collection 79
  Add 80, 83
  after 80
  Anlegen 81
  before 80
  Datentyp 79
  Einfügen 79
  Element 79
  Löschen 79
  Objekt 79
  Schlüsselname 79
  Speicherplatz 81
COLUMN_NAME 120
ColumnCount 319
CombineRgn 390
comdlg32.dll 197
Compiler 17
Compilerprogramm 16
COMSPEC 221
Connection-Objekt 118
Container 420
CopyMemory 151, 158
CopyTextToClip 204
cpl 183
CreateBitmap 311
CreateCompatibleDC 311
CreateFile 252, 269
CreateFontIndirect 344
CreateObject 116, 395
CreateRectRgn 390
CreateRoundRectRgn 390
CreateSolidBrush 311, 345
CreateToolhelp32Snapshot 408
Criteria 120
CSng 26
CStr 26
CurDir 220
CursorLocation 118
CursorType 119
CustomDocumentProperties 481
CVar 26

D

DateCreated 240
DateLastAccessed 240
DateLastModified 240
Datenfelder 79, 162
Datentypen 23
  Boolean 23
  Byte 23
  Currency 23
  Date 23
  Decimal 23
  Double 23
  Integer 23
  Long 23
  Object 24
  Single 23
  String 23
  Variant 24
DateSerial 237, 460
dBase 136

DDE 394
Debugging 17
Declare 146
DefType 30
  DefBool 31
  DefByte 31
  DefCur 31
  DefDate 31
  DefDbl 31
  DefInt 31
  DefLng 31
  DefSng 31
  DefStr 31
  DefVar 31
Deklination 292
DeleteObject 311
DestroyIcon 312
DeviceContext 177, 309, 311
DEVMODE 319
Dim 32
Dir 220, 222
DISP_CHANGE_RESTART 320
DISP_CHANGE_SUCCESSFUL 320
DoEvents 36
DOS 465
DrawIconEx 311–312
DrawMenuBar 384
DrawText 345
DRIVE_CDROM 276
DRIVE_FIXED 276
DRIVE_RAMDISK 276
DRIVE_REMOTE 276
DRIVE_REMOVABLE 276
Drive-Objekt 272
Drives 272
dwFileAttributes 241
DWORD 150
dwPlatformId 325, 474
dwUsage 428
Dynamic Data Exchange 394
Dynamisches Datenfeld 162

E

Early Binding 116, 395
EmptyClipboard 312, 333
EN 28601 288
EnableEvents 38, 57
EnableWindow 391
Enum 35
EnumDisplaySettings 319
EnumPrinters 468, 475
Error
  Clear 83
  Number 83
ERROR_BAD_FORMAT 396
ERROR_FILE_NOT_FOUND 396
ERROR_PATH_NOT_FOUND 396
Event 109
EWX_FORCE 324
EWX_FORCEIFHUNG 324
EWX_LOGOFF 323
EWX_POWEROFF 324
EWX_REBOOT 324
EWX_SHUTDOWN 324
EXCEL4 344
Execute 227
ExecuteExcel4Macro 477
ExitWindowsEx 324
Explore 397
ExtractIcon 310–312

F

Fakultät 61
Farbtiefe 311
Feiertage 285
Fenster 175
Fensterhandle 175
Fenstertext 176
File Transfer Protocol 444
FILE_ATTRIBUTE_ARCHIVE 241
FILE_ATTRIBUTE_DIRECTORY 237, 241
FILE_ATTRIBUTE_HIDDEN 241
FILE_ATTRIBUTE_NORMAL 241
FILE_ATTRIBUTE_READONLY 241
FILE_ATTRIBUTE_SYSTEM 241
FILE_ATTRIBUTE_TEMPORARY 241
FileDateTime 245
Filename 226
Files 240
FileSearch-Objekt 220, 225
FileSystemObject 220
FileTime 237, 252, 460
FileTimeToLocalFileTime 237, 253
FileTimeToNormalTime 252
FileTimeToSystemTime 237, 460
FillRect 311, 345
FindClose 231
FindControl 345

FindExecutable 408, 410
FindFirstFile 231
FindNextFile 231
FindWindow 175, 390
FindWindowA 382
FOF_NOERRORUI 256
Folder-Objekt 240
FoundFiles 228
FrameRect 345
Frühlingsvollmond 285
FTP 444
FtpCreateDirectory 457
FtpDeleteFile 457
FtpFindFirstFile 458
FtpGetCurrentDirectory 456
FtpGetFile 456
FtpRemoveDirectory 457
FtpRenameFile 458
FtpSetCurrentDirectory 456
FW_BOLD 346
FW_LIGHT 346
FW_MEDIUM 346
FW_NORMAL 346
FW_THIN 346

G

GDI 333
GetAttr 220, 239
GetCommandLine 157
GetCommandLineA 156
GetCreateTime 252
GetCurrentProcess 325
GetDC 311
GetDeviceCaps 311, 319
GetDiskFreeSpaceEx 274
  lpFreeBytesAvailableToCaller 274
  lpRootPathName 274
  lpTotalNumberOfBytes 274
  lpTotalNumberOfFreeBytes 275
GetDriveType 276
GetExitCodeProcess 405
GetFileAttributes 241, 244
GetFolder 240
GetHostbyAddr 441
GetHostByName 442
gethostname 442
GetLogicalDrives 275
GetLongPathName 261
GetObject 332, 395
GetOpenFileName 209, 211–212
GetSaveAsFileName 209, 212
GetShortPathName 261, 301
GetTempPath 243
GetTextColor 344
GetVersionEx 324, 474
GetVolumeInformation 273
  lpMaximumComponentLength 274
  lpRootPathName 273
  lpVolumeNameBuffer 273–274
  lpVolumeSerialNumber 274
  nVolumeNameSize 274
GetWindow 194
GetWindowLong 382
GetWindowRect 312, 344, 390
GetWindowText 194
Global 32
GlobalAlloc 204, 209
GlobalLock 204, 209
Globally Unique Identifier 485
GlobalUnlock 204
GoTo 50
Grundton 371
GUID 484–485
Gültigkeitsbereich 32
GW_CHILD 194
GW_HWNDNEXT 194
GWL_STYLE 382, 384

H

hAddrList 442
Hashwert 480
Heaps 408
Hexziffer 150
Hostent 441
HTTP 444

I

ICMP 431
icmp.dll 428
IcmpCreateFile 443
IcmpSendEcho 443
Icons 301
IconToClip 311
If 42
If Then Else 46
imagehlp.dll 230
Index 73
inet_addr 441

inet_ntoa 441
InetError 460
InputBox-Funktion 194
InputBox-Methode 194
Instanzhandle 309
IntelliSense 54
Internet Control Message Protocol 431
INTERNET_OPEN_TYPE_DIRECT 460
InternetCloseHandle 460
InternetConnect 456–458, 460
InternetFindNextFile 458
InternetGetLastResponseInfo 460
InternetOpen 460
InternetOpenUrl 459
InternetReadFile 459
Interpreter 16
IP_ECHO_REPLY 443
IP_OPTION_INFORMATION 443
IsFeiertag 287
Item 73

K

Kalenderwoche 288
Kammerton A 371
Kill 220, 255
KillTimer 193
Klassen 91
  Instanzierung 93
  Property 93
  Überladen 94
Klassenname 176
Kommentare 15
Kompilieren 16
Konstanten 34
Kontextmenü 301
Kurvenform 356

L

Längengrad 292
LastModified 227
  msoLastModifiedAnyTime 227
  msoLastModifiedLastMonth 227
  msoLastModifiedLastWeek 227
  msoLastModifiedThisMonth 227
  msoLastModifiedThisWeek 227
  msoLastModifiedToday 227
  msoLastModifiedYesterday 227
Late Binding 116
Lebensdauer 32
LhashValOfNameSys 480–481, 484
Lib 147
List 319
lNumBytesToRead 459
LoadBitmapBynum 332–333
LoadLibrary 332
LocalFileTimeToFileTime 253
LockAspectRatio 333
LockType 119
LOGFONT 209, 344
Logische Verknüpfungen 59
LookIn 226
LookupPrivilegeValue 325
lpcCount 427
lpExitCode 405
lphEnum 427
lpNetResource 427
LPSTR 155
lpszLocalName 276
lpszRemoteName 276
LSet 442
lstrcpy 157, 428, 441, 476
lstrlen 157, 428, 476

M

Mac-Adresse 480
macrofun.exe 480
MacroOptions 477
MakeSureDirectoryPathExists 254
Maschinencode 16
MatchTextExactly 227
Matrixfunktion 16
mciSendString 353
Menühandles 177
MessageboxA 183
MessageLoop 175
Metonischer Zyklus 286
Microsoft Scripting Runtime 272
midiOutClose 376
midiOutOpen 376
midiOutShortMsg 376
Mischen 70
MkDir 220, 254
modal 186
modeless 391
Module 408
MSADOX 113
MsgBox 183
MsoFileType 226
msoFileTypeOfficeFiles 227
MyNetwork 426

N

Name 220
Namenskonventionen 20
nAvgBytesPerSec 368
nBlockAlign 368
nChannels 368
nCombineMode 390
Net Send 464
NetMessageBufferSend 464
NETRESOURCE 427
Netzwerkressourcen 419
New 83
NewSearch 226
nFileSizeLow 237
Nibbles 149
NormalTimeToFileTime 252
Nothing 34
nSamplesPerSec 368

O

Object Linking and Embedding 394
Objektkatalog 182
Oem 465
OemToCharA 466
OFStrukt 211
Oktaven 370
oleaut32 480
OLE-Automatisierung 394
OnTime 38, 186
Open 356, 397
Open-Anweisung 243
OpenClipboard 312, 333
Open-Methode 118
OpenProcess 405, 408
OpenProcessToken 325
OpenSchema 120
Option Base 162
Option Compare Binary 65
Option Compare Text 65
Option Explicit 29
Options 120
Or 48
Oracle 138
Ostern 285
OSVERSIONINFO 324
OSVersionInfo 474
Output 243

P

Parameterübergabe 145
Path 230
Pause 356
pcbNeeded 475
Performance 42
Pivotelement 67
Pixel 177
Play 356
PostMessage 194
Potenzieren 61
pPortName 468
pPrinterName 476
Präfix 21–22
Presentations 415
Print 397
Printer_Info_2 468
Private 32
PROCESS_ALL_ACCESS 403
PROCESS_QUERY_INFORMATION 403
Process32First 408
Process32Next 408
PROCESSENTRY32 408
PROGRAM_CHANGE 376
Proxyserver 459
Public 32
Puffer 165
Punkt 177

Q

QueryClose 384
QueryType 120
Quicksort 67
Quit 395

R

RAM 142
Random 243
Randomize 70, 87
Reboot 320
Recordset 118
  AddNew 120
  BOF 123
  Delete 121
  EOF 123
  Filter 122
  GetRows 123
  GetString 124
  MoveFirst 121

Recordset (Forts.)
MoveLast 121
MoveNext 121
MovePrevious 121
RecordCount 127
Update 121
UpdateBatch 122
Rect 390
ReDim 162
ReDim Preserve 80, 162, 237
RegCloseKey 475
RegEnumValue 475
Region 177
Register 477
RegOpenKeyEx 475
Rekursionen 61, 420
Relationale Datenbank 112
ReleaseCapture 391
ReleaseDC 312
Repaint 309
Request for Comments 431, 444
RESOURCEUSAGE_CONTAINER 428
RFC 431, 444
RGB-Farben 197
RGN_DIFF 390
RmDir 220, 255
Rnd 70, 87
Root 420

S

Safearrays 167
SaveCopyAs 444
SC_CLOSE 194
SchemaID 120
Screen 311
ScreenUpdating 54
Scripting.FileSystemObject 220, 228
ScrRun.Dll 220
SE_ERR_ACCESSDENIED 396
SE_ERR_ASSOCINCOMPLETE 396
SE_ERR_DDEBUSY 396
SE_ERR_DDEFAIL 396
SE_ERR_DDETIMEOUT 396
SE_ERR_DLLNOTFOUND 396
SE_ERR_NOASSOC 396
SE_ERR_OOM 396
SE_ERR_SHARE 396
SE_PRIVILEGE_ENABLED 325
SE_SHUTDOWN_NAME 325
SearchSubFolders 226
SearchTreeForFile 230
Seek 356
Select 57
Select Case 42, 44
SelectObject 311, 332, 343
SendMessage 391
SendMessageString 217
Set 83
Set Door Closed 356
Set Door Open 356
SetAttr 220, 240
SetClipboardData 204, 312, 333
SetEndOfFile 269
SetFileAttributes 241, 244
SetFilePointer 269
SetTextColor 343–344
SetTimer 193
SetWindowLong 384
SetWindowPos 193
SetWindowRgn 390
SetWindowText 194
Shapes 332
SHBrowseForFolder 213
ShellExecute 269, 395
SHFileOperation 255
SHFILEOPSTRUCT
FO_COPY 255
FO_DELETE 255
FO_MOVE 255
FO_RENAME 255
FOF_ALLOWUNDO 256
FOF_FILESONLY 256
FOF_MULTIDESTFILES 256
FOF_NOCONFIRMATION 256
FOF_NOCONFIRMMKDIR 256
FOF_NOCOPYSECURITYATTRIBS 256
FOF_RENAMEONCOLLISION 256
FOF_SILENT 256
FOF_SIMPLEPROGRESS 256
FOF_WANTMAPPINGHANDLE 256
SHGetPathFromIDList 217
SHGetSpecialFolderPath 264
SHNAMEMAPPING 257
ShortName 230
ShortPath 230
Show 397
signed 150
Sleep 376, 405

SND_ASYNC 352
SND_LOOP 352
SND_PURGE 352
SND_SYNC 352
sndPlaySound 352
SNDREC32.EXE 352
Socket 441, 459
SOCKET_ERROR 442
Sonnenaufgang 292
Sonnenuntergang 292
SortBy 227
  msoSortByFileName 227
  msoSortByFileType 227
  msoSortByLastModified 227
  msoSortByNone 227
  msoSortBySize 227
SortOrder 228
  msoSortOrderAscending 228
  msoSortOrderDescending 228
Source 119
sProxyBypass 460
SQL 127
  ALL 127
  ASC 128
  DESC 128
  DISTINCT 127
  FROM 127
  GROUP BY 128
  HAVING 128
  ORDER BY 128
  SELECT 127
  TOP 127
  WHERE 128
Stack 42, 143
Stack-Pointer 144
Static 33
STILL_ACTIVE 403, 405
strComment 427
StrConv 156, 166, 344
strDisplayType 426
StringFromGUID2 486
strLocalName 427
strParent 426
strProvider 427
StrPtr 486
strRemoteName 427
strUsage 427
SubFolders 230
SW_HIDE 397
SW_MAXIMIZE 397
SW_MINIMIZE 397
SW_RESTORE 397
SW_SHOW 397
SW_SHOWDEFAULT 397
SW_SHOWMAXIMIZED 398
SW_SHOWMINIMIZED 398
SW_SHOWMINNOACTIVE 398
SW_SHOWNA 398
SW_SHOWNOACTIVATE 398
SW_SHOWNORMAL 398
Swapdatei 143
Synodischer Monat 286
Systemeinstellungen 183
SYSTEMTIME 237, 252
SystemTimeToFileTime 252–253
szExeFile 408

T

TABLE_CATALOG 120
TABLE_NAME 120
TABLE_SCHEMA 120
TABLE_TYPE 120
tapiRequestMakeCall 417
Target 416
TaskID 395, 403, 405, 408
TCP/IP 444
TerminateProcess 405, 408
Thread 408
ThunderDFrame 176
ThunderXFrame 176
Time To Live 441
Timeout 186
Timerereignis 193
TimeSerial 237, 460
TipGetLpfnOfFunctionId 187
TOKEN_ADJUST_PRIVILEGES 325
TOKEN_PRIVILEGES 325
TOKEN_QUERY 325
Tonleiter 370
TPLuid 325
Tracert 428
TTL-Zähler 431, 441
Twips 177
Type 196
Typenkennzeichen 25
Typenumwandlung 25

U

Überlauf 25
Unicode 149
unsigned 143
Unterstrich 23
Username 459
UUID 485

V

Variablen 17
Variablendeklaration 26
  erforderlich 29
Variablennamen 17
Variant 26
Variantvariablen 59
VarPtr 169, 428
VarPtrArray 170
VarType 26
  vbArray 27
  vbBoolean 27
  vbByte 27
  vbCurrency 27
  vbDataObject 27
  vbDate 27
  vbDecimal 27
  vbDouble 27
  vbEmpty 27
  vbError 27
  vbInteger 27
  vbLong 27
  vbNull 27
  vbObject 27
  vbSingle 27
  vbString 27
  vbVariant 27
VBALISTE.XLS 20
vbNullString 176, 382
vbUnicode 156
vbYesNo 194
Verweis 146
Vollkreis 369
Vollständige Referenzierung 52
Vorzeichenbit 150

W

Wait 415
Wav 356
waveformat 368
Wavehandle 369
WaveHdr 369
Waveheader 368
waveOutOpen 368
waveOutPrepareHeader 369
waveOutSetVolume 367
waveOutWrite 369
wFormatTag 368
WHDR_BEGINLOOP 369
WHDR_ENDLOOP 369
WIN32_FIND_DATA 237, 241, 458
Windows 175
Windows Script Host 186
winmm.dll 370
With 52
WithEvents 103
WM_ENDSESSION 324
WM_LBUTTONDOWN 194
WM_LBUTTONUP 194
WM_QUERYENDSESSION 324
WnetEnumResource 427
WNetGetConnection 275
WNetOpenEnum 427
Word 150
Worksheet_BeforeDoubleClick 416
Worksheet_SelectionChange 57
WS_DLGFRAME 383
WS_MAXIMIZEBOX 382
WS_MINIMIZEBOX 383
WS_SYSMENU 382
WS_THICKFRAME 383
WSACleanup 442
WSAData 442
WSAStartup 442
WScript.Network 281
WSH 186
wsock32.dll 428
Wurzel 420

X

XLMAIN 344
xlVeryHidden 481

Z

Zählvariablen 18
Zufallszahlen 84

Copyright

Zusatzdaten

Möglicherweise liegt dem gedruckten Buch eine CD-ROM mit Zusatzdaten bei. Die Zurverfügungstellung dieser Daten auf der Website ist eine freiwillige Leistung des Verlags. Der Rechtsweg ist ausgeschlossen.

Hinweis

Dieses und andere eBooks können Sie rund um die Uhr und legal auf unserer Website

(http://www.informit.de)

herunterladen